mlrun

Release UNKNOWN

Iguazio

Apr 11, 2023

MLRUN BASICS

CHAPTER

ONE

USING MLRUN

MLRun is an open MLOps platform for quickly building and managing continuous ML applications across their lifecycle. MLRun integrates into your development and CI/CD environment and automates the delivery of production data, ML pipelines, and online applications. MLRun significantly reduces engineering efforts, time to production, and computation resources. With MLRun, you can choose any IDE on your local machine or on the cloud. MLRun breaks the silos between data, ML, software, and DevOps/MLOps teams, enabling collaboration and fast continuous improvements.

Get started with MLRun ***Tutorials and examples***, ***Installation and setup guide***, or read about ***Using MLRun***.

This page explains how MLRun addresses the ***MLOps tasks*** and presents the ***MLRun core components***.

1.1 MLOps tasks

Project management and CI/CD automation Ingest and process data Develop and train models Deploy models and apps Monitor and alert The **MLOps development workflow** section describes the different tasks and stages in detail. MLRun can be used to automate and orchestrate all the different tasks or just specific tasks (and integrate them with what you have already deployed).

Project management and CI/CD automation

In MLRun the assets, metadata, and services (data, functions, jobs, artifacts, models, secrets, etc.) are organized into projects. Projects can be imported/exported as a whole, mapped to git repositories or IDE projects (in PyCharm, VSCode, etc.), which enables versioning, collaboration, and CI/CD. Project access can be restricted to a set of users and roles. more…

Docs: Projects and automation CI/CD integration , **Tutorials:** quick start Automated ML pipeline , **Videos:** Quick start

Ingest and process data

MLRun provides abstract interfaces to various offline and online **data sources**, supports batch or realtime data processing at scale, data lineage and versioning, structured and unstructured data, and more. In addition, the MLRun **Feature store** automates the collection, transformation, storage, catalog, serving, and monitoring of data features across the ML lifecycle and enables feature reuse and sharing. more…

Docs: Feature store Data & artifacts , **Tutorials:** quick start Feature store

Develop and train models

MLRun allows you to easily build ML pipelines that take data from various sources or the Feature Store and process it, train models at scale with multiple parameters, test models, track each experiment, and register, version and deploy models, etc. MLRun provides scalable built-in or custom model training services that integrate with any framework and can work with 3rd party training/auto-ML services. You can also bring your own pre-trained model and use it in the pipeline. more...

Docs: Model training and tracking Batch runs and workflows , **Tutorials:** Train & eval models Automated ML pipeline , **Videos:** Train & compare models

Deploy models and applications

MLRun rapidly deploys and manages production-grade real-time or batch application pipelines using elastic and resilient serverless functions. MLRun addresses the entire ML application: intercepting application/user requests, running data processing tasks, inferencing using one or more models, driving actions, and integrating with the application logic. more...

Docs: Realtime pipelines Batch inference , **Tutorials:** Realtime serving Batch inference Advanced pipeline , **Videos:** Serve pre-trained models

Monitor and alert

Observability is built into the different MLRun objects (data, functions, jobs, models, pipelines, etc.), eliminating the need for complex integrations and code instrumentation. With MLRun, you can observe the application/model resource usage and model behavior (drift, performance, etc.), define custom app metrics, and trigger alerts or retraining jobs. more...

Docs: Model monitoring overview , **Tutorials:** Model monitoring & drift detection

1.2 MLRun core components

MLRun includes the following major components:

Project management & automation (SDK, API, etc.) Serverless functions
Data & artifacts Feature store Batch runs & workflows Real-time
pipelines Monitoring ***Project management*****:** A service (API, SDK, DB, UI) that manages the different project assets (data, functions, jobs, workflows, secrets, etc.) and provides central control and metadata layer.

Serverless functions**:** An automatically deployed software package with one or more methods and runtime-specific attributes (such as image, libraries, command, arguments, resources, etc.).

Data & artifacts**:** Glueless connectivity to various data sources, metadata management, catalog, and versioning for structured/unstructured artifacts.

Feature store**:** Automatically collects, prepares, catalogs, and serves production data features for development (offline) and real-time (online) deployment using minimal engineering effort.

Batch Runs & workflows**:** Execute one or more functions with specific parameters and collect, track, and compare all their results and artifacts.

Real-time serving pipeline**:** Rapid deployment of scalable data and ML pipelines using real-time serverless technology, including API handling, data preparation/enrichment, model serving, ensembles, driving and measuring actions, etc.

***Real-time monitoring*:** Monitors data, models, resources, and production components and provides a feedback loop for exploring production data, identifying drift, alerting on anomalies or data quality issues, triggering retraining jobs, measuring business impact, etc.

1.2.1 MLRun architecture

MLRun started as a community effort to map the different components in the ML project lifecycle, provide a common metadata layer, and automate the operationalization process (a.k.a MLOps).

Instead of a siloed, complex, and manual process, MLRun enables production pipeline design using a modular strategy, where the different parts contribute to a continuous, automated, and far simpler path from research and development to scalable production pipelines without refactoring code, adding glue logic, or spending significant efforts on data and ML engineering.

MLRun uses **Serverless Function** technology: write the code once, using your preferred development environment and simple "local" semantics, and then run it as-is on different platforms and at scale. MLRun automates the build process, execution, data movement, scaling, versioning, parameterization, output tracking, CI/CD integration, deployment to production, monitoring, and more.

Those easily developed data or ML "functions" can then be published or loaded from a hub and used later to form offline or real-time production pipelines with minimal engineering efforts.

MLRun deployment

MLRun has two main components, the service and the client (SDK):

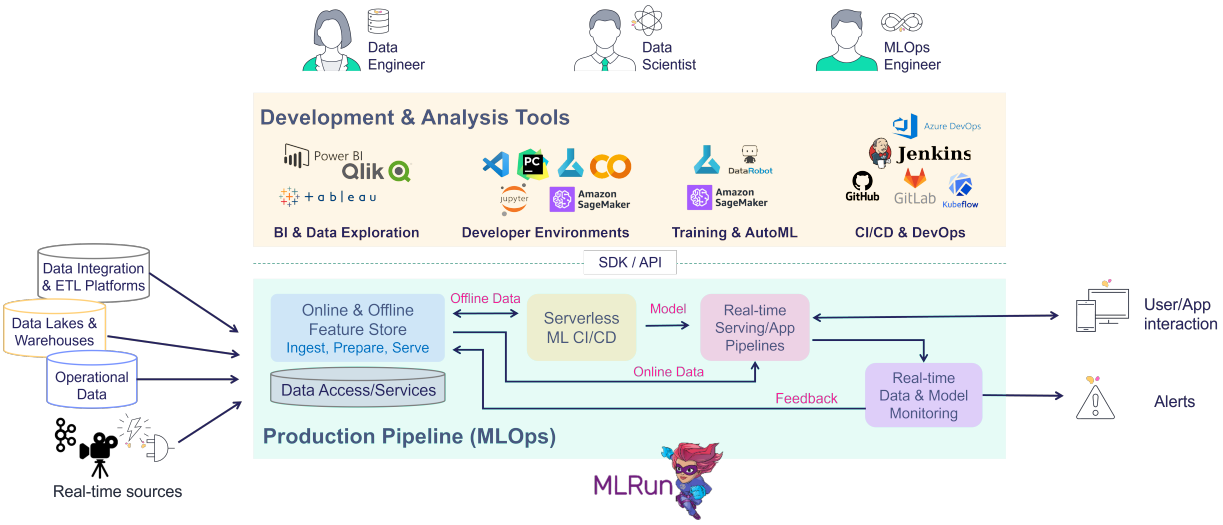
- The MLRun service runs over Kubernetes (can also be deployed using local Docker for demo and test purposes). It can orchestrate and integrate with other open source open source frameworks, as shown in the following diagram.
- The MLRun client SDK is installed in your development environment and interacts with the service using REST API calls.

MLRun: an integrated and open approach

Data preparation, model development, model and application delivery, and end to end monitoring are tightly connected: they cannot be managed in silos. This is where MLRun MLOps orchestration comes in. ML, data, and DevOps/MLOps teams collaborate using the same set of tools, practices, APIs, metadata, and version control.

MLRun provides an open architecture that supports your existing development tools, services, and practices through an open API/SDK and pluggable architecture.

MLRun simplifies & accelerates the time to production !



While each component in MLRun is independent, the integration provides much greater value and simplicity. For example:

- The training jobs obtain features from the feature store and update the feature store with metadata, which will be used in the serving or monitoring.
- The real-time pipeline enriches incoming events with features stored in the feature store. It can also use feature metadata (policies, statistics, schema, etc.) to impute missing data or validate data quality.
- The monitoring layer collects real-time inputs and outputs from the real-time pipeline and compares them with the features data/metadata from the feature store or model metadata generated by the training layer. Then, it writes all the fresh production data back to the feature store so it can be used for various tasks such as data analysis, model retraining (on fresh data), and model improvements.

When one of the components detailed above is updated, it immediately impacts the feature generation, the model serving pipeline, and the monitoring. MLRun applies versioning to each component, as well as versioning and rolling upgrades across components.

CHAPTER

TWO

MLOPS DEVELOPMENT WORKFLOW

ML applications require you to implement the following stages in a scalable and reproducible way:

1. *Ingest and process data*
2. *Develop and train models*
3. *Deploy models and applications*
4. *Monitor and alert*

MLRun automates the MLOps work. It simplifies & accelerates the time to production

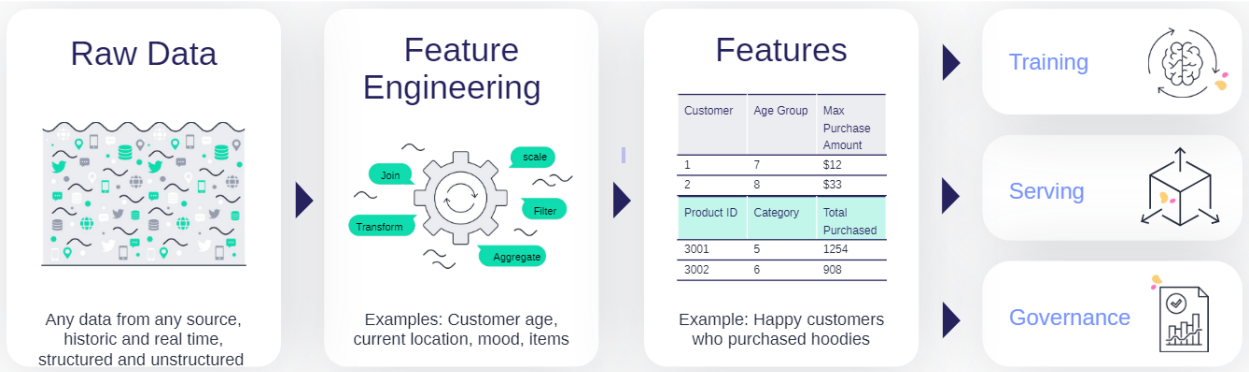
2.1 Ingest and process data

There is no ML without data. Before everything else, ML teams need access to historical and/or online data from multiple sources, and they must catalog and organize the data in a way that allows for simple and fast analysis (for example, by storing data in columnar data structures, such as Parquet).

In most cases, the raw data cannot be used as-is for machine learning algorithms for various reasons such as:

- The data is low quality (missing fields, null values, etc.) and requires cleaning and imputing.
- The data needs to be converted to numerical or categorical values which can be processed by algorithms.
- The data is unstructured in text, json, image, or audio formats, and needs to be converted to tabular or vector formats.
- The data needs to be grouped or aggregated to make it meaningful.
- The data is encoded or requires joins with reference information.
- The ML process starts with manual exploratory data analysis and feature engineering on small data extractions. In order to bring accurate models into production, ML teams must work on larger datasets and automate the process of collecting and preparing the data.

Furthermore, batch collection and preparation methodologies such as ETL, SQL queries, and batch analytics don't work well for operational or real-time pipelines. As a result, ML teams often build separate data pipelines which use stream processing, NoSQL, and containerized micro- services. 80% of data today is unstructured, so an essential part of building operational data pipelines is to convert unstructured textual, audio, and visual data into machine learning- or deep learning-friendly data organization.



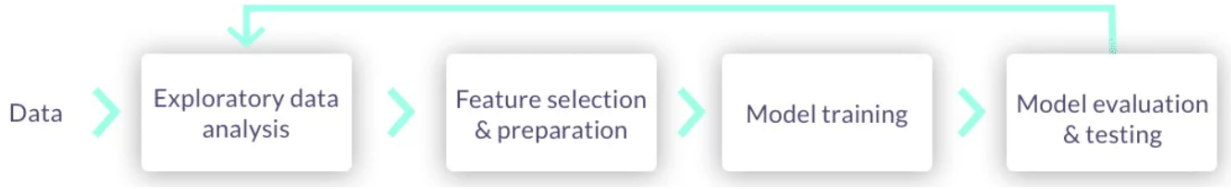
MLOps solutions should incorporate a feature store that defines the data collection and transformations just once for both batch and real-time scenarios, processes features automatically without manual involvement, and serves the features from a shared catalog to training, serving, and data governance applications. Feature stores must also extend beyond traditional analytics and enable advanced transformations on unstructured data and complex layouts.

2.2 Develop and train models

Whether it's deep learning or machine learning, MLRun allows you to train your models at scale and capture all the relevant metadata for experiments tracking and lineage.

With MLOps, ML teams build machine learning pipelines that automatically collect and prepare data, select optimal features, run training using different parameter sets or algorithms, evaluate models, and run various model and system tests. All the executions, along with their data, metadata, code and results must be versioned and logged, providing quick results visualization, to compare them with past results and understand which data was used to produce each model.

Pipelines can be more complex—for example, when ML teams need to develop a combination of models, or use Deep Learning or NLP.



ML pipelines can be triggered manually, or preferably triggered automatically when:

- The code, packages or parameters change
- The input data or feature engineering logic changes
- Concept drift is detected, and the model needs to be re-trained with fresh data

ML pipelines:

- Are built using micro-services (containers or serverless functions), usually over Kubernetes.
- Have all their inputs (code, package dependencies, data, parameters) and the outputs (logs, metrics, data/features, artifacts, models) tracked for every step in the pipeline, in order to reproduce and/or explain the experiment results.
- Use versioning for all the data and artifacts used throughout the pipeline.

- Store code and configuration in versioned Git repositories.
- Use Continuous Integration (CI) techniques to automate the pipeline initiation, test automation, and for the review and approval process.

Pipelines should be executed over scalable services or functions, which can span elastically over multiple servers or containers. This way, jobs complete faster, and computation resources are freed up once they complete, saving significant costs.

The resulting models are stored in a versioned model repository along with metadata, performance metrics, required parameters, statistical information, etc. Models can be loaded later into batch or real-time serving micro-services or functions.

2.3 Deploy models and applications

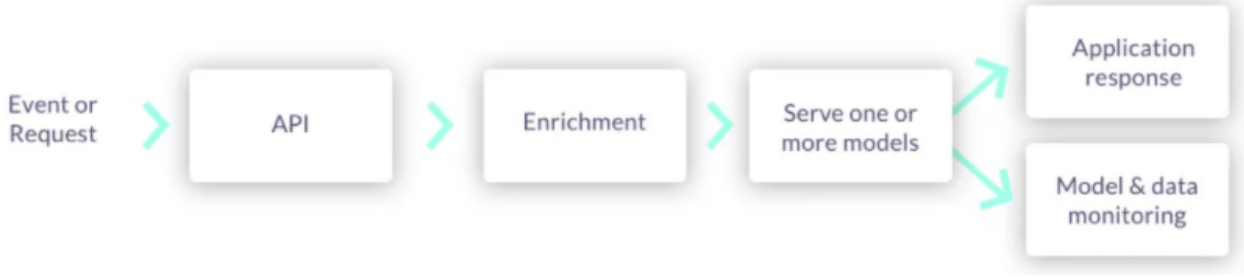
With MLRun, in addition to a batch inference, you can deploy a robust and scalable *real-time pipeline* for more complex and online scenarios. MLRun uses Nuclio, an open source serverless framework for creating real-time pipelines for model deployment.

Once an ML model has been built, it needs to be integrated with real-world data and the business application or front-end services. The entire application, or parts thereof, need to be deployed without disrupting the service. Deployment can be extremely challenging if the ML components aren't treated as an integral part of the application or production pipeline.

Production pipelines usually consist of:

- Real-time data collection, validation, and feature engineering logic
- One or more model serving services
- API services and/or application integration logic
- Data and model monitoring services
- Resource monitoring and alerting services
- Event, telemetry, and data/features logging services

The different services are interdependent. For example, if the inputs to a model change, the feature engineering logic must be upgraded along with the model serving and model monitoring services. These dependencies require online production pipelines (graphs) to reflect these changes.



Production pipelines can be more complex when using unstructured data, deep learning, NLP or model ensembles, so having flexible mechanisms to build and wire up the pipeline graphs is critical.

Production pipelines are usually interconnected with fast streaming or messaging protocols, so they should be elastic to address traffic and demand fluctuations, and they should allow non-disruptive upgrades to one or more elements of the pipeline. These requirements are best addressed with fast serverless technologies.

Production pipeline development and deployment flow:

1. Develop production components:
    - API services and application integration logic
    - Feature collection, validation, and transformation
    - Model serving graphs
2. Test online pipelines with simulated data
3. Deploy online pipelines to production
4. Monitor models and data and detect drift
5. Retrain models and re-engineer data when needed
6. Upgrade pipeline components (non-disruptively) when needed

2.4 Monitor and alert

Once the model is deployed, use MLRun to track the operational statistics as well as identify drift. When drift is identified, MLRun can trigger the training pipeline to train a new model.

AI services and applications are becoming an essential part of any business. This trend brings with it liabilities, which drive further complexity. ML teams need to add data, code and experiment tracking, monitor data to detect quality problems, and monitor models to detect concept drift and improve model accuracy through the use of AutoML techniques and ensembles, and so on.

Nothing lasts forever, not even carefully constructed models that have been trained using mountains of well-labeled data. ML teams need to react quickly to adapt to constantly changing patterns in real-world data. Monitoring machine learning models is a core component of MLOps to keep deployed models current and predicting with the utmost accuracy, and to ensure they deliver value long-term.

CHAPTER

THREE

TUTORIALS AND EXAMPLES

The following tutorials provide a hands-on introduction to using MLRun to implement a data science workflow and automate machine-learning operations (MLOps).

- **Quick-start Tutorial** (**watch video**)
- ***Targeted Tutorials***
- ***End to End Demos***

Make sure you start with the Quick start tutorial to understand the basics Introduction to MLRun - Use serverless functions to train and deploy models

3.1 Quick start tutorial

Introduction to MLRun - Use serverless functions to train and deploy models

This notebook provides a quick overview of developing and deploying machine learning applications using the MLRun MLOps orchestration framework.

Tutorial steps:

- ***Install MLRun***
- ***Define the MLRun project and ML functions***
- ***Run the data processing function and log artifacts***
- ***Use the MLRun built-in Function Hub functions for training***
- ***Build, test, and deploy model serving functions***

Watch the video tutorial.

3.1.1 Install MLRun

MLRun has a backend service that can run locally or over Kubernetes (preferred). See the instructions for installing it locally using Docker or over Kubernetes Cluster. Alternatively, you can use Iguazio's managed MLRun service.

Before you start, make sure the MLRun client package is installed and configured properly:

This notebook uses sklearn. If it is not installed in your environment run `!pip install scikit-learn~=1.0`.

```
# Install MLRun and sklearn, run this only once (restart the notebook after the install !!)
%pip install mlrun scikit-learn~=1.0
```

Restart the notebook kernel after the pip installation.

```
import mlrun
```

Configure the client environment

MLRun client connects to the local or remote MLRun service/cluster using a REST API. To configure the service address, credentials, and default settings, you use the `mlrun.set_environment()` method, or environment variables, (see details in Set up your client environment.)

You can skip this step when using MLRun Jupyter notebooks or Iguazio's managed notebooks.

3.1.2 Define MLRun project and ML functions

MLRun **Project** is a container for all your work on a particular activity or application. Projects host `functions`, `workflow`, `artifacts`, `secrets`, and more. Projects have access control and can be accessed by one or more users. They are usually associated with a GIT and interact with CI/CD frameworks for automation. See the MLRun Projects documentation.

Create a new project

```
project = mlrun.get_or_create_project("quick-tutorial", "./", user_project=True)
```

```
> 2022-09-20 13:19:49,414 [info] loaded project quick-tutorial from MLRun DB
```

MLRun serverless functions specify the source code, base `image`, extra package `requirements`, runtime engine `kind` (batch `job`, real-time `serving`, `spark`, `dask`, etc.), and desired resources (cpu, gpu, mem, storage, ...). The runtime engines (local, job, Nuclio, Spark, etc.) automatically transform the function code and spec into fully managed and elastic services that run over Kubernetes. Function source code can come from a single file (.py, .ipynb, etc.) or a full archive (git, zip, tar). MLRun can execute an entire file/notebook or specific function classes/handlers.

Note

The `@mlrun.handler` is a decorator that logs the returning values to MLRun as configured. This example uses the default settings so that it logs a dataset (`pd.DataFrame`) and a string value by getting the returned objects types. In addition to logging outputs, the decorator can parse incoming inputs to the required type. For more info, see the `mlrun.handler` documentation.

Function code

Run the following cell to generate the data prep file (or copy it manually):

```
%%writefile data-prep.py

import pandas as pd
from sklearn.datasets import load_breast_cancer

import mlrun


@mlrun.handler(outputs=["dataset", "label_column"])
def breast_cancer_generator():
    """
    A function which generates the breast cancer dataset
    """
    breast_cancer = load_breast_cancer()
    breast_cancer_dataset = pd.DataFrame(
        data=breast_cancer.data, columns=breast_cancer.feature_names
    )
    breast_cancer_labels = pd.DataFrame(data=breast_cancer.target, columns=["label"])
    breast_cancer_dataset = pd.concat(
        [breast_cancer_dataset, breast_cancer_labels], axis=1
    )

    return breast_cancer_dataset, "label"
```

```
Overwriting data-prep.py
```

Create a serverless function object from the code above, and register it in the project

```
data_gen_fn = project.set_function("data-prep.py", name="data-prep", kind="job", image=
↪"mlrun/mlrun", handler="breast_cancer_generator")
project.save()  # save the project with the latest config
```

```
<mlrun.projects.project.MlrunProject at 0x7ff72063d460>
```

3.1.3 Run your data processing function and log artifacts

Functions are executed (using the CLI or SDK **run** command) with an optional `handler`, various `params`, `inputs`, and resource requirements. This generates a `run` object that can be tracked through the CLI, UI, and SDK. Multiple functions can be executed and tracked as part of a multi-stage pipeline (workflow).

Note

When a function has additional package `requirements`, or needs to include the content of a `source` archive, you must first build the function using the `project.build_function()` method.

The `local` flag indicates if the function is executed **locally** or "teleported" and executed in the **Kubernetes cluster**. The execution progress and results can be viewed in the UI (see hyperlinks below).

Run using the SDK

```
gen_data_run = project.run_function("data-prep", local=True)
```

```
> 2022-09-20 13:22:59,351 [info] starting run data-prep-breast_cancer_generator␣
↪uid=1ea3533192364dbc8898ce328988d0a3 DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-09-20 13:22:59,693 [info] run executed, status=completed
```

Print the run state and outputs

```
gen_data_run.state()
```

```
'completed'
```

```
gen_data_run.outputs
```

```
{'label_column': 'label',
 'dataset': 'store://artifacts/quick-tutorial-iguazio/data-prep-breast_cancer_generator_
↪dataset:1ea3533192364dbc8898ce328988d0a3'}
```

Print the output dataset artifact (`DataItem` object) as dataframe

```
gen_data_run.artifact("dataset").as_df().head()
```

```
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          0.10960
3        11.42         20.38           77.58      386.1          0.14250
4        20.29         14.34          135.10     1297.0          0.10030

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419
1           0.07864          0.0869              0.07017         0.1812
2           0.15990          0.1974              0.12790         0.2069
3           0.28390          0.2414              0.10520         0.2597
4           0.13280          0.1980              0.10430         0.1809

   mean fractal dimension  ...  worst texture  worst perimeter  worst area  \
0                 0.07871  ...          17.33           184.60      2019.0
1                 0.05667  ...          23.41           158.80      1956.0
2                 0.05999  ...          25.53           152.50      1709.0
3                 0.09744  ...          26.50            98.87       567.7
4                 0.05883  ...          16.67           152.20      1575.0
```

(continues on next page)

(continued from previous page)

```
   worst smoothness  worst compactness  worst concavity  worst concave points  \
0            0.1622             0.6656           0.7119                0.2654
1            0.1238             0.1866           0.2416                0.1860
2            0.1444             0.4245           0.4504                0.2430
3            0.2098             0.8663           0.6869                0.2575
4            0.1374             0.2050           0.4000                0.1625

   worst symmetry  worst fractal dimension  label
0          0.4601                  0.11890      0
1          0.2750                  0.08902      0
2          0.3613                  0.08758      0
3          0.6638                  0.17300      0
4          0.2364                  0.07678      0

[5 rows x 31 columns]
```

3.1.4 Train a model using an MLRun built-in Function Hub

MLRun provides a **Function Hub** that hosts a set of pre-implemented and validated ML, DL, and data processing functions.

You can import the `auto-trainer` hub function that can: train an ML model using a variety of ML frameworks; generate various metrics and charts; and log the model along with its metadata into the MLRun model registry.

```
# Import the function
trainer = mlrun.import_function('hub://auto_trainer')
```

See the `auto_trainer` function usage instructions in the Function Hub or by typing `trainer.doc()`

Run the function on the cluster (if there is)

```
trainer_run = project.run_function(trainer,
    inputs={"dataset": gen_data_run.outputs["dataset"]},
    params = {
        "model_class": "sklearn.ensemble.RandomForestClassifier",
        "train_test_split_size": 0.2,
        "label_columns": "label",
        "model_name": 'cancer',
    },
    handler='train',
)
```

```
> 2022-09-20 13:23:14,811 [info] starting run auto-trainer-train␣
↪uid=84057e1510174611a5d2de0671ee803e DB=http://mlrun-api:8080
> 2022-09-20 13:23:14,970 [info] Job is running in the background, pod: auto-trainer-
↪train-dzjwz
Matplotlib created a temporary config/cache directory at /tmp/matplotlib-3pzdch1o␣
↪because the default path (/.config/matplotlib) is not a writable directory; it is␣
↪highly recommended to set the MPLCONFIGDIR environment variable to a writable␣
↪directory, in particular to speed up the import of Matplotlib and to better support␣
↪multiprocessing.
```

(continues on next page)

(continued from previous page)

```
> 2022-09-20 13:23:20,953 [info] Sample set not given, using the whole training set as␣
↪the sample set
> 2022-09-20 13:23:21,143 [info] training 'cancer'
> 2022-09-20 13:23:22,561 [info] run executed, status=completed
final state: completed
```

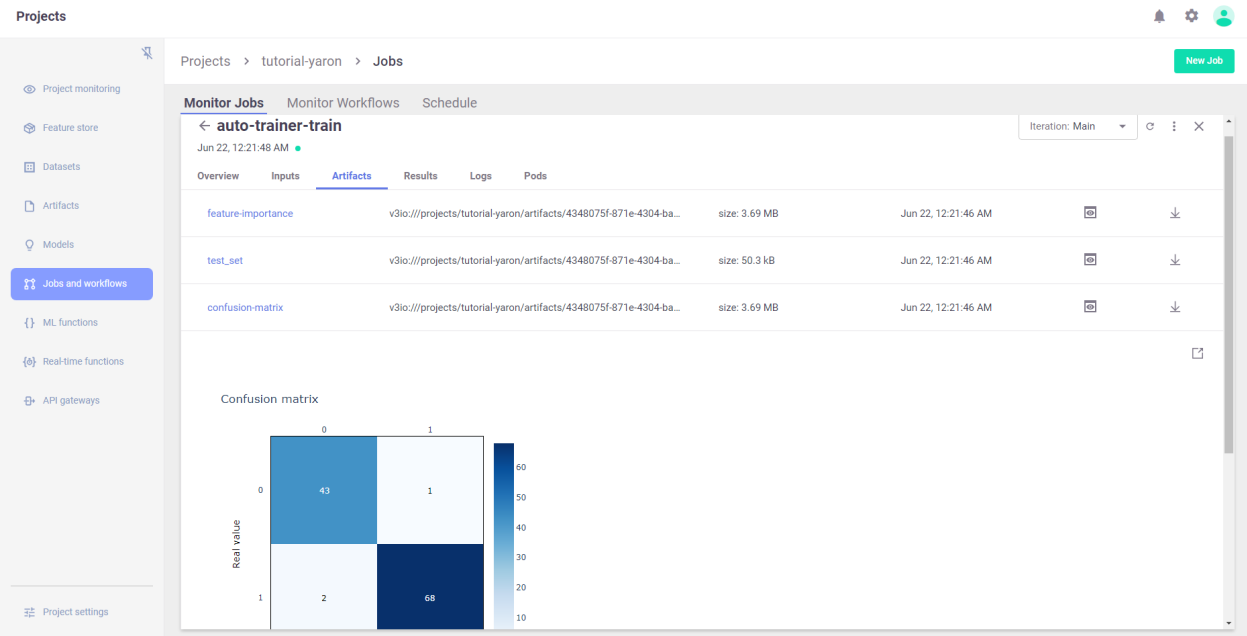
```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-09-20 13:23:24,216 [info] run executed, status=completed
```

View the job progress results and the selected run in the MLRun UI



Results (metrics) and artifacts are generated and tracked automatically by MLRun

```
trainer_run.outputs
```

```
{'accuracy': 0.956140350877193,
 'f1_score': 0.967741935483871,
 'precision_score': 0.9615384615384616,
 'recall_score': 0.974025974025974,
 'feature-importance': 'v3io:///projects/quick-tutorial-iguazio/artifacts/auto-trainer-
↪train/0/feature-importance.html',
 'test_set': 'store://artifacts/quick-tutorial-iguazio/auto-trainer-train_test_
↪set:84057e1510174611a5d2de0671ee803e',
 'confusion-matrix': 'v3io:///projects/quick-tutorial-iguazio/artifacts/auto-trainer-
↪train/0/confusion-matrix.html',
 'roc-curves': 'v3io:///projects/quick-tutorial-iguazio/artifacts/auto-trainer-train/0/
↪roc-curves.html',
```

(continues on next page)

(continued from previous page)

```
 'calibration-curve': 'v3io:///projects/quick-tutorial-iguazio/artifacts/auto-trainer-
↪train/0/calibration-curve.html',
 'model': 'store://artifacts/quick-tutorial-iguazio/
↪cancer:84057e1510174611a5d2de0671ee803e'}
```

```
# Display HTML output artifacts
trainer_run.artifact('confusion-matrix').show()
```

```
<IPython.core.display.HTML object>
```

3.1.5 Build, test, and deploy the model serving functions

MLRun serving can produce managed, real-time, serverless, pipelines composed of various data processing and ML tasks. The pipelines use the Nuclio real-time serverless engine, which can be deployed anywhere. For more details and examples, see MLRun serving graphs.

Create a model serving function

```
serving_fn = mlrun.new_function("serving", image="python:3.8", kind="serving",␣
↪requirements=["mlrun[complete]", "scikit-learn==1.1.2"])
```

Add a model

The basic serving topology supports a router with multiple child models attached to it. The `function.add_model()` method allows you to add models and specify the `name`, `model_path` (to a model file, dir, or artifact), and the serving `class` (built-in or user defined).

```
serving_fn.add_model('cancer-classifier',model_path=trainer_run.outputs["model"], class_
↪name='mlrun.frameworks.sklearn.SklearnModelServer')
```

```
<mlrun.serving.states.TaskStep at 0x7ff6da1ac190>
```

```
# Plot the serving graph topology
serving_fn.spec.graph.plot(rankdir="LR")
```

```
<graphviz.graphs.Digraph at 0x7ff6da1acaf0>
```

Simulating the model server locally

```
# Create a mock (simulator of the real-time function)
server = serving_fn.to_mock_server()
```

```
> 2022-09-20 13:24:24,867 [warning] run command, file or code were not specified
> 2022-09-20 13:24:25,240 [info] model cancer-classifier was loaded
> 2022-09-20 13:24:25,241 [info] Loaded ['cancer-classifier']
```

Test the mock model server endpoint

- List the served models

```
server.test("/v2/models/", method="GET")
```

```
{'models': ['cancer-classifier']}
```

- Infer using test data

```
my_data = {"inputs"
           :[[
               1.371e+01, 2.083e+01, 9.020e+01, 5.779e+02, 1.189e-01, 1.645e-01,
               9.366e-02, 5.985e-02, 2.196e-01, 7.451e-02, 5.835e-01, 1.377e+00,
               3.856e+00, 5.096e+01, 8.805e-03, 3.029e-02, 2.488e-02, 1.448e-02,
               1.486e-02, 5.412e-03, 1.706e+01, 2.814e+01, 1.106e+02, 8.970e+02,
               1.654e-01, 3.682e-01, 2.678e-01, 1.556e-01, 3.196e-01, 1.151e-01]
            ]
}
server.test("/v2/models/cancer-classifier/infer", body=my_data)
```

```
X does not have valid feature names, but RandomForestClassifier was fitted with feature␣
↪names
```

```
{'id': '27d3f10a36ce465f841d3e19ca404889',
 'model_name': 'cancer-classifier',
 'outputs': [0]}
```

- Read the model name, ver and schema (input and output features)

Deploy a real-time serving function (over Kubernetes or Docker)

This section requires Nuclio to be installed (over k8s or Docker).

Use the mlrun `deploy_function()` method to build and deploy a Nuclio serving function from your serving-function code. You can deploy the function object (`serving_fn`) or reference pre-registered project functions.

```
project.deploy_function(serving_fn)
```

```
> 2022-09-20 13:24:34,823 [info] Starting remote function deploy
2022-09-20 13:24:35  (info) Deploying function
2022-09-20 13:24:35  (info) Building
2022-09-20 13:24:35  (info) Staging files and preparing base images
2022-09-20 13:24:35  (info) Building processor image
2022-09-20 13:25:35  (info) Build complete
2022-09-20 13:26:05  (info) Function deploy complete
> 2022-09-20 13:26:06,030 [info] successfully deployed function: {'internal_invocation_
↪urls': ['nuclio-quick-tutorial-iguazio-serving.default-tenant.svc.cluster.local:8080'],
↪ 'external_invocation_urls': ['quick-tutorial-iguazio-serving-quick-tutorial-iguazio.
↪default-tenant.app.alexp-edge.lab.iguazeng.com/']}
```

```
DeployStatus(state=ready, outputs={'endpoint': 'http://quick-tutorial-iguazio-serving-
↪quick-tutorial-iguazio.default-tenant.app.alexp-edge.lab.iguazeng.com/', 'name':
↪'quick-tutorial-iguazio-serving'})
```

- Test the live endpoint

```
serving_fn.invoke("/v2/models/cancer-classifier/infer", body=my_data)
```

```
> 2022-09-20 13:26:06,094 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-quick-tutorial-iguazio-serving.default-tenant.svc.cluster.local:8080/v2/models/
↪cancer-classifier/infer'}
```

```
{'id': '2533b72a-6d94-4c51-b960-02a2deaf84b6',
 'model_name': 'cancer-classifier',
 'outputs': [0]}
```

3.1.6 Done!

Congratulation! You've completed Part 1 of the MLRun getting-started tutorial. Proceed to **Part 2: Train, Track, Compare, and Register Models** to learn how to train an ML model.

3.2 Train, compare, and register models

This notebook provides a quick overview of training ML models using MLRun MLOps orchestration framework.

Make sure you reviewed the basics in MLRun **Quick Start Tutorial**.

Tutorial steps:

- ***Define an MLRun project and a training functions***
- ***Run the function, log the artifacts and model***
- ***Hyper-parameter tuning and model/experiment comparison***
- ***Build and test the model serving functions***

Watch the video tutorial.

3.2.1 MLRun installation and configuration

Before running this notebook make sure `mlrun` and `sklearn` packages are installed (`pip install mlrun scikit-learn~=1.0`) and that you have configured the access to the MLRun service.

```
# Install MLRun if not installed, run this only once (restart the notebook after the␣
↪install !!!)
%pip install mlrun
```

3.2.2 Define MLRun project and a training functions

You should create, load, or use (get) an MLRun project that holds all your functions and assets.

Get or create a new project

The `get_or_create_project()` method tries to load the project from MLRun DB. If the project does not exist, it creates a new one.

```
import mlrun
project = mlrun.get_or_create_project("tutorial", context="./", user_project=True)
```

```
> 2022-09-20 13:55:10,543 [info] loaded project tutorial from None or context and saved␣
↪in MLRun DB
```

Add (auto) MLOps to your training function

Training functions generate models and various model statistics. You'll want to store the models along with all the relevant data, metadata, and measurements. MLRun can apply all the MLOps functionality automatically ("Auto-MLOps") by simply using the framework-specific `apply_mlrun()` method.

This is the line to add to your code, as shown in the training function below.

```
apply_mlrun(model=model, model_name="my_model", x_test=x_test, y_test=y_test)
```

`apply_mlrun()` manages the training process and automatically logs all the framework-specific model object, details, data, metadata, and metrics. It accepts the model object and various optional parameters. When specifying the `x_test` and `y_test` data it generates various plots and calculations to evaluate the model. Metadata and parameters are automatically recorded (from MLRun `context` object) and therefore don't need to be specified.

Function code

Run the following cell to generate the `trainer.py` file (or copy it manually):

```
%%writefile trainer.py

import pandas as pd

from sklearn import ensemble
from sklearn.model_selection import train_test_split

import mlrun
from mlrun.frameworks.sklearn import apply_mlrun


@mlrun.handler()
def train(
    dataset: pd.DataFrame,
    label_column: str = "label",
    n_estimators: int = 100,
    learning_rate: float = 0.1,
    max_depth: int = 3,
    model_name: str = "cancer_classifier",
):
    # Initialize the x & y data
    x = dataset.drop(label_column, axis=1)
```

(continues on next page)

(continued from previous page)

```
    y = dataset[label_column]

    # Train/Test split the dataset
    x_train, x_test, y_train, y_test = train_test_split(
        x, y, test_size=0.2, random_state=42
    )

    # Pick an ideal ML model
    model = ensemble.GradientBoostingClassifier(
        n_estimators=n_estimators, learning_rate=learning_rate, max_depth=max_depth
    )

    # ------------------- The only line you need to add for MLOps ----------------------
↪---
    # Wraps the model with MLOps (test set is provided for analysis & accuracy␣
↪measurements)
    apply_mlrun(model=model, model_name=model_name, x_test=x_test, y_test=y_test)
    # ----------------------------------------------------------------------------------
↪---

    # Train the model
    model.fit(x_train, y_train)
```

```
Overwriting trainer.py
```

Create a serverless function object from the code above, and register it in the project

```
trainer = project.set_function("trainer.py", name="trainer", kind="job", image="mlrun/
↪mlrun", handler="train")
```

3.2.3 Run the training function and log the artifacts and model

Create a dataset for training

```
import pandas as pd
from sklearn.datasets import load_breast_cancer
breast_cancer = load_breast_cancer()
breast_cancer_dataset = pd.DataFrame(data=breast_cancer.data, columns=breast_cancer.
↪feature_names)
breast_cancer_labels = pd.DataFrame(data=breast_cancer.target, columns=["label"])
breast_cancer_dataset = pd.concat([breast_cancer_dataset, breast_cancer_labels], axis=1)

breast_cancer_dataset.to_csv("cancer-dataset.csv", index=False)
```

Run the function (locally) using the generated dataset

```
trainer_run = project.run_function(
    "trainer",
    inputs={"dataset": "cancer-dataset.csv"},
```

(continues on next page)

(continued from previous page)

```
    params = {"n_estimators": 100, "learning_rate": 1e-1, "max_depth": 3},
    local=True
)
```

```
> 2022-09-20 13:56:57,630 [info] starting run trainer-train␣
↪uid=b3f1bc3379324767bee22f44942b96e4 DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-09-20 13:56:59,356 [info] run executed, status=completed
```

View the auto generated results and artifacts

```
trainer_run.outputs
```

```
{'accuracy': 0.956140350877193,
 'f1_score': 0.965034965034965,
 'precision_score': 0.9583333333333334,
 'recall_score': 0.971830985915493,
 'feature-importance': 'v3io:///projects/tutorial-iguazio/artifacts/trainer-train/0/
↪feature-importance.html',
 'test_set': 'store://artifacts/tutorial-iguazio/trainer-train_test_
↪set:b3f1bc3379324767bee22f44942b96e4',
 'confusion-matrix': 'v3io:///projects/tutorial-iguazio/artifacts/trainer-train/0/
↪confusion-matrix.html',
 'roc-curves': 'v3io:///projects/tutorial-iguazio/artifacts/trainer-train/0/roc-curves.
↪html',
 'calibration-curve': 'v3io:///projects/tutorial-iguazio/artifacts/trainer-train/0/
↪calibration-curve.html',
 'model': 'store://artifacts/tutorial-iguazio/cancer_
↪classifier:b3f1bc3379324767bee22f44942b96e4'}
```

```
trainer_run.artifact('feature-importance').show()
```

```
<IPython.core.display.HTML object>
```

Export model files + metadata into a zip (requires MLRun 1.1.0 and later)

You can `export()` the model package (files + metadata) into a zip, and load it on a remote system/cluster by running `model = project.import_artifact(key, path)`).

```
trainer_run.artifact('model').meta.export("model.zip")
```

3.2.4 Hyper-parameter tuning and model/experiment comparison

Run a `GridSearch` with a couple of parameters, and select the best run with respect to the `max accuracy`. (For more details, see MLRun Hyper-Param and Iterative jobs.)

For basic usage you can run the hyperparameters tuning job by using the arguments:

- `hyperparams` for the hyperparameters options and values of choice.
- `selector` for specifying how to select the best model.

Running a remote function

To run the hyper-param task over the cluster you need the input data to be available for the job, using object storage or the MLRun versioned artifact store.

The following line logs (and uploads) the dataframe as a project artifact:

```
dataset_artifact = project.log_dataset("cancer-dataset", df=breast_cancer_dataset,␣
↪index=False)
```

Run the function over the remote Kubernetes cluster (`local` is not set):

```
hp_tuning_run = project.run_function(
    "trainer",
    inputs={"dataset": dataset_artifact.uri},
    hyperparams={
        "n_estimators": [10, 100, 1000],
        "learning_rate": [1e-1, 1e-3],
        "max_depth": [2, 8]
    },
    selector="max.accuracy",
)
```

```
> 2022-09-20 13:57:28,217 [info] starting run trainer-train␣
↪uid=b7696b221a174f66979be01138797f19 DB=http://mlrun-api:8080
> 2022-09-20 13:57:28,365 [info] Job is running in the background, pod: trainer-train-
↪xfzfp
Matplotlib created a temporary config/cache directory at /tmp/matplotlib-zuih5pkq␣
↪because the default path (/.config/matplotlib) is not a writable directory; it is␣
↪highly recommended to set the MPLCONFIGDIR environment variable to a writable␣
↪directory, in particular to speed up the import of Matplotlib and to better support␣
↪multiprocessing.
> 2022-09-20 13:58:07,356 [info] best iteration=3, used criteria max.accuracy
> 2022-09-20 13:58:07,750 [info] run executed, status=completed
final state: completed
```

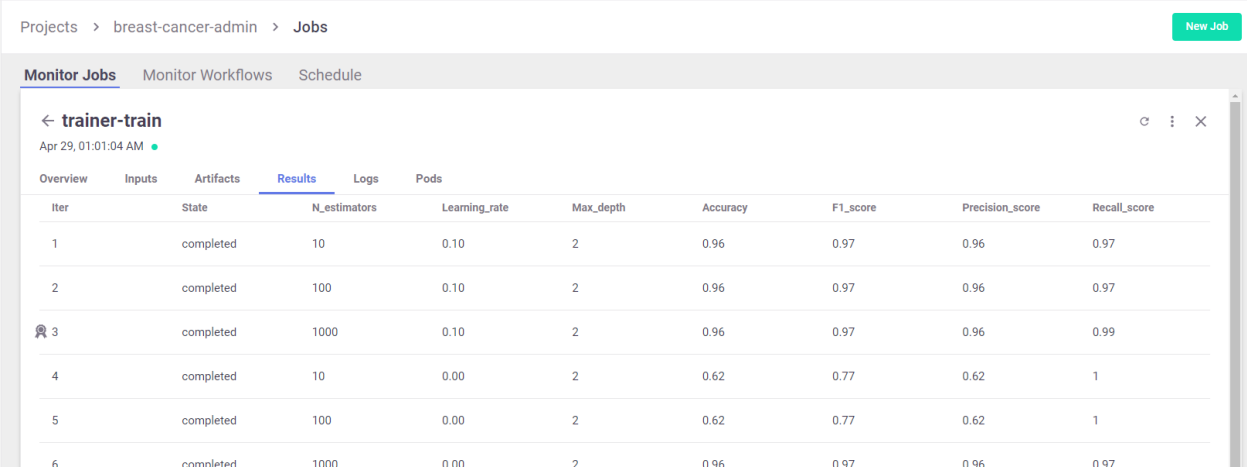
```
<IPython.core.display.HTML object>
```

```

```

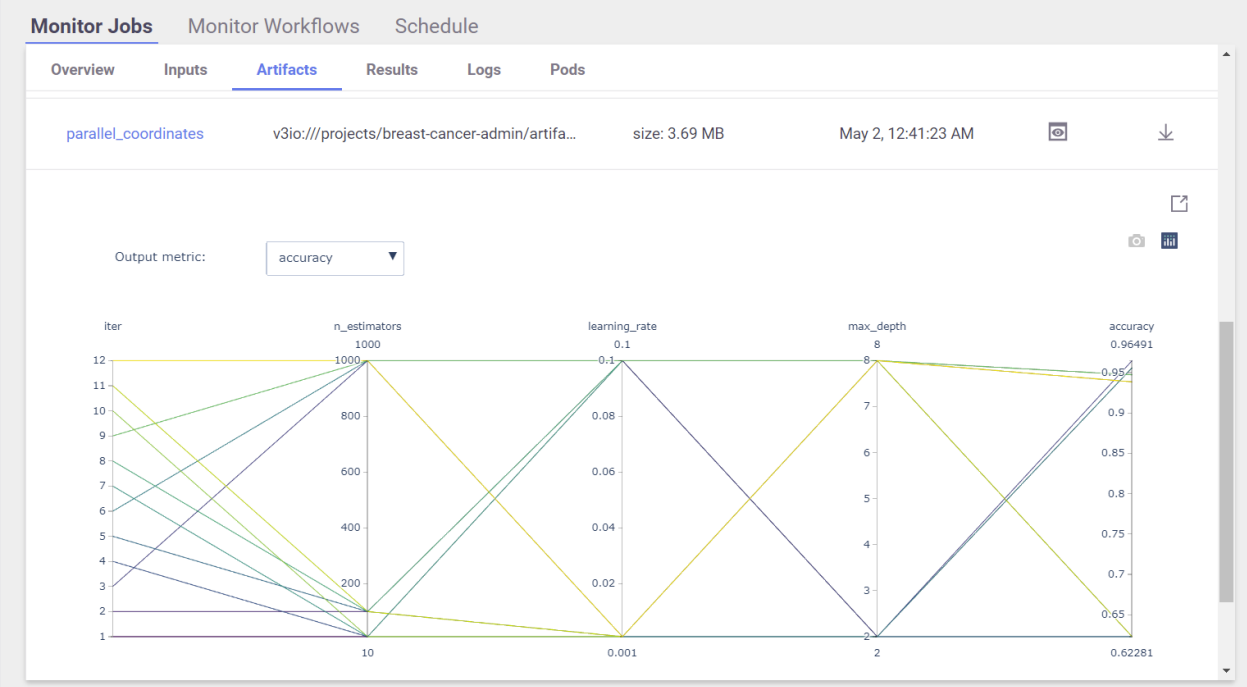
```
<IPython.core.display.HTML object>
```

```
> 2022-09-20 13:58:13,925 [info] run executed, status=completed
```

View Hyper-param results and the selected run in the MLRun UI



Interactive Parallel Coordinates Plot



List the generated models and compare the different runs

```
hp_tuning_run.outputs
```

```
{'best_iteration': 3,
 'accuracy': 0.9649122807017544,
 'f1_score': 0.9722222222222222,
 'precision_score': 0.958904109589041,
 'recall_score': 0.9859154929577465,
 'feature-importance': 'v3io:///projects/tutorial-iguazio/artifacts/trainer-train/3/
↪feature-importance.html',
 'test_set': 'store://artifacts/tutorial-iguazio/trainer-train_test_
↪set:b7696b221a174f66979be01138797f19',
 'confusion-matrix': 'v3io:///projects/tutorial-iguazio/artifacts/trainer-train/3/
↪confusion-matrix.html',
```

(continues on next page)

(continued from previous page)

```
 'roc-curves': 'v3io:///projects/tutorial-iguazio/artifacts/trainer-train/3/roc-curves.
↪html',
 'calibration-curve': 'v3io:///projects/tutorial-iguazio/artifacts/trainer-train/3/
↪calibration-curve.html',
 'model': 'store://artifacts/tutorial-iguazio/cancer_
↪classifier:b7696b221a174f66979be01138797f19',
 'iteration_results': 'v3io:///projects/tutorial-iguazio/artifacts/trainer-train/0/
↪iteration_results.csv',
 'parallel_coordinates': 'v3io:///projects/tutorial-iguazio/artifacts/trainer-train/0/
↪parallel_coordinates.html'}
```

```
# List the models in the project (can apply filters)
models = project.list_models()
for model in models:
    print(f"uri: {model.uri}, metrics: {model.metrics}")
```

```
uri: store://models/tutorial-iguazio/cancer_classifier
↪#0:b3f1bc3379324767bee22f44942b96e4, metrics: {'accuracy': 0.956140350877193, 'f1_score
↪': 0.965034965034965, 'precision_score': 0.9583333333333334, 'recall_score': 0.
↪971830985915493}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#1:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.956140350877193, 'f1_score
↪': 0.965034965034965, 'precision_score': 0.9583333333333334, 'recall_score': 0.
↪971830985915493}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#2:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.956140350877193, 'f1_score
↪': 0.965034965034965, 'precision_score': 0.9583333333333334, 'recall_score': 0.
↪971830985915493}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#3:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.9649122807017544, 'f1_
↪score': 0.9722222222222222, 'precision_score': 0.958904109589041, 'recall_score': 0.
↪9859154929577465}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#4:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.6228070175438597, 'f1_
↪score': 0.7675675675675676, 'precision_score': 0.6228070175438597, 'recall_score': 1.0}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#5:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.6228070175438597, 'f1_
↪score': 0.7675675675675676, 'precision_score': 0.6228070175438597, 'recall_score': 1.0}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#6:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.956140350877193, 'f1_score
↪': 0.965034965034965, 'precision_score': 0.9583333333333334, 'recall_score': 0.
↪971830985915493}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#7:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.9385964912280702, 'f1_
↪score': 0.951048951048951, 'precision_score': 0.9444444444444444, 'recall_score': 0.
↪9577464788732394}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#8:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.9473684210526315, 'f1_
↪score': 0.9577464788732394, 'precision_score': 0.9577464788732394, 'recall_score': 0.
↪9577464788732394}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#9:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.9473684210526315, 'f1_
↪score': 0.9577464788732394, 'precision_score': 0.9577464788732394, 'recall_score': 0.
↪9577464788732394}
```

(continues on next page)

(continued from previous page)

```
uri: store://models/tutorial-iguazio/cancer_classifier
↪#10:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.6228070175438597, 'f1_
↪score': 0.7675675675675676, 'precision_score': 0.6228070175438597, 'recall_score': 1.0}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#11:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.6228070175438597, 'f1_
↪score': 0.7675675675675676, 'precision_score': 0.6228070175438597, 'recall_score': 1.0}
uri: store://models/tutorial-iguazio/cancer_classifier
↪#12:b7696b221a174f66979be01138797f19, metrics: {'accuracy': 0.9385964912280702, 'f1_
↪score': 0.951048951048951, 'precision_score': 0.9444444444444444, 'recall_score': 0.
↪9577464788732394}
```

```
# To view the full model object use:
# print(models[0].to_yaml())
```

```
# Compare the runs (generate interactive parallel coordinates plot and a table)
project.list_runs(name="trainer-train", iter=True).compare()
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

3.2.5 Build and test the model serving functions

MLRun serving can produce managed, real-time, serverless, pipelines composed of various data processing and ML tasks. The pipelines use the Nuclio real-time serverless engine, which can be deployed anywhere. For more details and examples, see the MLRun Serving Graphs.

Create a model serving function from your code, and (view it here)

```
serving_fn = mlrun.new_function("serving", image="mlrun/mlrun", kind="serving")
serving_fn.add_model('cancer-classifier',model_path=hp_tuning_run.outputs["model"],␣
↪class_name='mlrun.frameworks.sklearn.SklearnModelServer')
```

```
<mlrun.serving.states.TaskStep at 0x7feb1f55faf0>
```

```
# Create a mock (simulator of the real-time function)
server = serving_fn.to_mock_server()

my_data = {"inputs"
           :[[
               1.371e+01, 2.083e+01, 9.020e+01, 5.779e+02, 1.189e-01, 1.645e-01,
               9.366e-02, 5.985e-02, 2.196e-01, 7.451e-02, 5.835e-01, 1.377e+00,
               3.856e+00, 5.096e+01, 8.805e-03, 3.029e-02, 2.488e-02, 1.448e-02,
               1.486e-02, 5.412e-03, 1.706e+01, 2.814e+01, 1.106e+02, 8.970e+02,
               1.654e-01, 3.682e-01, 2.678e-01, 1.556e-01, 3.196e-01, 1.151e-01]
            ]
}
server.test("/v2/models/cancer-classifier/infer", body=my_data)
```

```
> 2022-09-20 14:12:35,714 [warning] run command, file or code were not specified
> 2022-09-20 14:12:35,859 [info] model cancer-classifier was loaded
> 2022-09-20 14:12:35,860 [info] Loaded ['cancer-classifier']
```

```
/conda/envs/mlrun-extended/lib/python3.8/site-packages/sklearn/base.py:450: UserWarning:

X does not have valid feature names, but GradientBoostingClassifier was fitted with␣
↪feature names
```

```
{'id': 'd9aee47cadc042ebbd9474ec0179a446',
 'model_name': 'cancer-classifier',
 'outputs': [0]}
```

3.2.6 Done!

Congratulation! You've completed Part 2 of the MLRun getting-started tutorial. Proceed to **Part 3: Model serving** to learn how to deploy and serve your model using a serverless function.

3.3 Serving pre-trained ML/DL models

This notebook demonstrate how to serve standard ML/DL models using **MLRun Serving**.

Make sure you went over the basics in MLRun **Quick Start Tutorial**.

MLRun serving can produce managed real-time serverless pipelines from various tasks, including MLRun models or standard model files. The pipelines use the Nuclio real-time serverless engine, which can be deployed anywhere. Nuclio is a high-performance open-source "serverless" framework that's focused on data, I/O, and compute-intensive workloads.

MLRun serving supports advanced real-time data processing and model serving pipelines. For more details and examples, see the MLRun serving pipelines documentation.

Tutorial steps:

- ***Using pre-built MLRun serving classes and images***
- ***Create and test the serving function***
- ***Deploy the serving function***
- ***Build a custom serving class***
- ***Building advanced model serving graph***

Watch the video tutorial.

3.3.1 MLRun installation and configuration

Before running this notebook make sure the `mlrun` package is installed (`pip install mlrun`) and that you have configured the access to MLRun service.

```
# Install MLRun if not installed, run this only once. Restart the notebook after the␣
↪install!
%pip install mlrun
```

Get or create a new project

You should create, load or use (get) an MLRun Project. The `get_or_create_project()` method tries to load the project from the MLRun DB. If the project does not exist it creates a new one.

```
import mlrun
project = mlrun.get_or_create_project("tutorial", context="./", user_project=True)
```

```
> 2022-06-20 09:07:50,188 [info] loaded project tutorial from MLRun DB
```

3.3.2 Using pre-built MLRun serving classes and images

MLRun contains built-in serving functionality for the major ML/DL frameworks (Scikit-Learn, TensorFlow.Keras, ONNX, XGBoost, LightGBM, and PyTorch). In addition, MLRun provides a few container images with the required ML/DL packages pre-installed.

You can overwrite the packages in the images, or provide your own image. (You just need to make sure that the `mlrun` package is installed in it.)

The following table specifies, for each framework, the relevant pre-integrated image and the corresponding MLRun `ModelServer` serving class:

framework	image	serving class
SciKit-Learn	mlrun/mlrun	mlrun.frameworks.sklearn.SklearnModelServer
TensorFlow.Keras	mlrun/ml-models	mlrun.frameworks.tf_keras.TFKerasModelServer
ONNX	mlrun/ml-models	mlrun.frameworks.onnx.ONNXModelServer
XGBoost	mlrun/ml-models	mlrun.frameworks.xgboost.XGBoostModelServer
LightGBM	mlrun/ml-models	mlrun.frameworks.lgbm.LGBMModelServer
PyTorch	mlrun/ml-models	mlrun.frameworks.pytorch.PyTorchModelServer

For GPU support use the `mlrun/ml-models-gpu` image (adding GPU drivers and support)

Example using SKlearn and TF Keras models

See how to specify the parameters in the following two examples. These use standard pre-trained models (using the iris dataset) stored in MLRun samples repository. (You can use your own models instead.)

```
models_dir = mlrun.get_sample_path('models/serving/')

framework = 'sklearn'  # change to 'keras' to try the 2nd option
kwargs = {}
if framework == "sklearn":
    serving_class = 'mlrun.frameworks.sklearn.SklearnModelServer'
    model_path = models_dir + 'sklearn.pkl'
```

(continues on next page)

(continued from previous page)

```
    image = 'mlrun/mlrun'
else:
    serving_class = 'mlrun.frameworks.tf_keras.TFKerasModelServer'
    model_path = models_dir + 'keras.h5'
    image = 'mlrun/ml-models'  # or mlrun/ml-models-gpu when using GPUs
    kwargs['labels'] = {'model-format': 'h5'}
```

Log the model

The model and its metadata are first registered in MLRun's **Model Registry**. Use the `log_model()` method to specify the model files and metadata (metrics, schema, parameters, etc.).

```
model_object = project.log_model(f'{framework}-model', model_file=model_path, **kwargs)
```

3.3.3 Create and test the serving function

Create a new **`serving`** function, specify its `name` and the correct `image` (with your desired framework).

If you want to add specific packages to the base image, specify the `requirements` attribute, example:

```
serving_fn = mlrun.new_function("serving", image=image, kind="serving",␣
↪requirements=["tensorflow==2.8.1"])
```

The following example uses a basic topology of a model `router` and adds a single model behind it. (You can add multiple models to the same function.)

```
serving_fn = mlrun.new_function("serving", image=image, kind="serving", requirements={})
serving_fn.add_model(framework ,model_path=model_object.uri, class_name=serving_class,␣
↪to_list=True)

# Plot the serving topology input -> router -> model
serving_fn.plot(rankdir="LR")
```

```
<graphviz.dot.Digraph at 0x7fdd81da8750>
```

Simulate the model server locally (using the mock_server)

```
# Create a mock server that represents the serving pipeline
server = serving_fn.to_mock_server()
```

Test the mock model server endpoint

- List the served models

```
server.test("/v2/models/", method="GET")
```

```
{'models': ['sklearn']}
```

- Infer using test data

```
sample = {"inputs":[[5.1, 3.5, 1.4, 0.2],[7.7, 3.8, 6.7, 2.2]]}
server.test(path=f'/v2/models/{framework}/infer',body=sample)
```

```
{'id': '1da64557daa843c1a2d6719eea7d4361',
 'model_name': 'sklearn',
 'outputs': [0, 2]}
```

See more API options and parameters in Model serving API.

3.3.4 Deploy the serving function

Deploy the serving function and use `invoke` to test it with the provided `sample`.

```
project.deploy_function(serving_fn)
```

```
> 2022-06-20 09:07:56,977 [info] Starting remote function deploy
2022-06-20 09:07:57  (info) Deploying function
2022-06-20 09:07:57  (info) Building
2022-06-20 09:07:57  (info) Staging files and preparing base images
2022-06-20 09:07:57  (info) Building processor image
2022-06-20 09:08:32  (info) Build complete
2022-06-20 09:08:44  (info) Function deploy complete
> 2022-06-20 09:08:44,641 [info] successfully deployed function: {'internal_invocation_
↪urls': ['nuclio-tutorial-yaron-serving.default-tenant.svc.cluster.local:8080'],
↪'external_invocation_urls': ['tutorial-yaron-serving-tutorial-yaron.default-tenant.app.
↪yh43.iguazio-cd1.com/']}
```

```
DeployStatus(state=ready, outputs={'endpoint': 'http://tutorial-yaron-serving-tutorial-
↪yaron.default-tenant.app.yh43.iguazio-cd1.com/', 'name': 'tutorial-yaron-serving'})
```

```
serving_fn.invoke(path=f'/v2/models/{framework}/infer',body=sample)
```

```
> 2022-06-20 09:08:44,692 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-tutorial-yaron-serving.default-tenant.svc.cluster.local:8080/v2/models/sklearn/
↪infer'}
```

```
{'id': 'a16f00e8-663a-4031-a04e-e42a7d4dd697',
 'model_name': 'sklearn',
 'outputs': [0, 2]}
```

3.3.5 Build a custom serving class

Model serving classes implement the full model serving functionality, which include loading models, pre- and post-processing, prediction, explainability, and model monitoring.

Model serving classes must inherit from `mlrun.serving.V2ModelServer`, and at the minimum implement the `load()` (download the model file(s) and load the model into memory) and `predict()` (accept request payload and return prediction/inference results) methods.

For more detailed information on custom serving classes, see Build your own model serving class.

The following code demonstrates a minimal scikit-learn (a.k.a. sklearn) serving-class implementation:

```
from cloudpickle import load
import numpy as np
from typing import List
import mlrun

class ClassifierModel(mlrun.serving.V2ModelServer):
    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

    def predict(self, body: dict) -> List:
        """Generate model predictions from sample."""
        feats = np.asarray(body['inputs'])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()
```

In order to create a function that incorporates the code of the new class (in `serving.py`) use `code_to_function`:

```
serving_fn = mlrun.code_to_function('serving', filename='serving.py', kind='serving',
↪image='mlrun/mlrun')
serving_fn.add_model('my_model',model_path=model_file, class_name='ClassifierModel')
```

3.3.6 Build an advanced model serving graph

MLRun graphs enable building and running DAGs (directed acyclic graphs). Graphs are composed of individual steps. The first graph element accepts an `Event` object, transforms/processes the event and passes the result to the next step in the graph, and so on. The final result can be written out to a destination (file, DB, stream, etc.) or returned back to the caller (one of the graph steps can be marked with `.respond()`).

The serving graphs can be composed of pre-defined graph steps, block-type elements (model servers, routers, ensembles, data readers and writers, data engineering tasks, validators, etc.), custom steps, or from native python classes/functions. A graph can have data processing steps, model ensembles, model servers, post-processing, etc. Graphs can auto-scale and span multiple function containers (connected through streaming protocols).

See the Advanced Model Serving Graph Notebook Example.

3.3.7 Done!

Congratulations! You've completed Part 3 of the MLRun getting-started tutorial. Proceed to **Part 4: ML Pipeline** to learn how to create an automated pipeline for your project.

3.4 Projects and automated ML pipeline

This notebook demonstrates how to work with projects, source control (git), and automating the ML pipeline.

Make sure you went over the basics in MLRun **Quick Start Tutorial**.

MLRun Project is a container for all your work on a particular activity: all the associated code, functions, jobs, workflows, data, models, and artifacts. Projects can be mapped to `git` repositories to enable versioning, collaboration, and CI/CD.

You can create project definitions using the SDK or a yaml file and store those in the MLRun DB, a file, or an archive. Once the project is loaded you can run jobs/workflows that refer to any project element by name, allowing separation between configuration and code. See load projects for details.

Projects contain `workflows` that execute the registered functions in a sequence/graph (DAG), and that can reference project parameters, secrets and artifacts by name. MLRun currently supports two workflow engines, `local` (for simple tasks) and Kubeflow Pipelines (for more complex/advanced tasks). MLRun also supports a real-time workflow engine (see online serving pipelines (graphs).

An ML Engineer can gather the different functions created by the data engineer and data scientist and create this automated pipeline.

Tutorial steps:

- ***Set up the project and functions***
- ***Work with GIT and archives***
- ***Build and run automated ML pipelines and CI/CD***
- ***Test the deployed model endpoint***

3.4.1 MLRun installation and configuration

Before running this notebook make sure the `mlrun` package is installed (`pip install mlrun`) and that you have configured the access to MLRun service.

```
# Install MLRun if not installed, run this only once (restart the notebook after the␣
↪install !!!)
%pip install mlrun
```

3.4.2 Set up the project and functions

Get or create a project

There are three ways to create/load **MLRun projects**:

- `mlrun.projects.new_project()` — Create a new MLRun project and optionally load it from a yaml/zip/git template.
- `mlrun.projects.load_project()` — Load a project from a context directory or remote git/zip/tar archive.
- `mlrun.projects.get_or_create_project()` — Load a project from the MLRun DB if it exists, or from a specified context/archive.

Projects refer to a `context` directory that holds all the project code and configuration. The `context` dir is usually mapped to a `git` repository and/or to an IDE (PyCharm, VSCode, etc.) project.

```
import mlrun
project = mlrun.get_or_create_project("tutorial", context="./", user_project=True)
```

```
> 2022-09-20 14:59:47,322 [info] loaded project tutorial from MLRun DB
```

Register project functions

To run workflows, you must save the definitions for the functions in the project so that function objects are initialized automatically when you load a project or when running a project version in automated CI/CD workflows. In addition, you might want to set/register other project attributes such as global parameters, secrets, and data.

Functions are registered using the `set_function()` command, where you can specify the code, requirements, image, etc. Functions can be created from a single code/notebook file or have access to the entire project context directory. (By adding the `with_repo=True` flag, it guarantees that the project context is cloned into the function runtime environment).

Function registration examples:

```
    # Example: register a notebook file as a function
    project.set_function('mynb.ipynb', name='test-function', image="mlrun/mlrun",␣
↪handler="run_test")

    # Define a job (batch) function that uses code/libs from the project repo
    project.set_function(
        name="myjob", handler="my_module.job_handler",
        image="mlrun/mlrun", kind="job", with_repo=True,
    )
```

Function code

Run the following cell to generate the data prep file (or copy it manually):

```
%%writefile data-prep.py

import pandas as pd
from sklearn.datasets import load_breast_cancer

import mlrun
```

(continues on next page)

(continued from previous page)

```
@mlrun.handler(outputs=["dataset", "label_column"])
def breast_cancer_generator():
    """
    A function that generates the breast cancer dataset
    """
    breast_cancer = load_breast_cancer()
    breast_cancer_dataset = pd.DataFrame(
        data=breast_cancer.data, columns=breast_cancer.feature_names
    )
    breast_cancer_labels = pd.DataFrame(data=breast_cancer.target, columns=["label"])
    breast_cancer_dataset = pd.concat(
        [breast_cancer_dataset, breast_cancer_labels], axis=1
    )

    return breast_cancer_dataset, "label"
```

```
Overwriting data-prep.py
```

Register the function above in the project

```
project.set_function("data-prep.py", name="data-prep", kind="job", image="mlrun/mlrun",␣
↪handler="breast_cancer_generator")
```

```
<mlrun.runtimes.kubejob.KubejobRuntime at 0x7fd96c30a0a0>
```

Register additional project objects and metadata

You can define other objects (workflows, artifacts, secrets) and parameters in the project and use them in your functions, for example:

```
    # Register a simple named artifact in the project (to be used in workflows)
    data_url = 'https://s3.wasabisys.com/iguazio/data/iris/iris.data.raw.csv'
    project.set_artifact('data', target_path=data_url)

    # Add a multi-stage workflow (./workflow.py) to the project with the name 'main' and␣
↪save the project
    project.set_workflow('main', "./workflow.py")

    # Read env vars from dict or file and set as project secrets
    project.set_secrets({"SECRET1": "value"})
    project.set_secrets(file_path="secrets.env")

    project.spec.params = {"x": 5}
```

Save the project

```
# Save the project in the db (and into the project.yaml file)
project.save()
```

```
<mlrun.projects.project.MlrunProject at 0x7fd96c2fdb50>
```

When you save the project it stores the project definitions in the `project.yaml`. This allows reconstructing the project in a remote cluster or a CI/CD system.

See the generated project file: **project.yaml**.

3.4.3 Work with GIT and archives

Push the project code/metadata into an archive

Use standard git commands to push the current project tree into a git archive. Make sure you `.save()` the project before pushing it.

```
git remote add origin <server>
git commit -m "Commit message"
git push origin master
```

Alternatively, you can use MLRun SDK calls:

- `project.create_remote(git_uri, branch=branch)` — to register the remote Git path
- `project.push()` — save the project state and commit/push updates to the remote git repo

You can also save the project content and metadata into a local or remote `.zip` archive, for example:

```
project.export("../archive1.zip")
project.export("s3://my-bucket/archive1.zip")
project.export(f"v3io://projects/{project.name}/archive1.zip")
```

Load a project from local/remote archive

The project metadata and context (code and configuration) can be loaded and initialized using the *load_project()* method. When `url` (of the git/zip/tar) is specified, it clones a remote repo into the local `context` dir.

```
# Load the project and run the 'main' workflow
project = load_project(context="./", name="myproj", url="git://github.com/mlrun/project-
↪archive.git")
project.run("main", arguments={'data': data_url})
```

Projects can also be loaded and executed using the CLI:

```
mlrun project -n myproj -u "git://github.com/mlrun/project-archive.git" .
mlrun project -r main -w -a data=<data-url> .
```

```
# load the project in the current context dir
project = mlrun.load_project("./")
```

3.4.4 Build and run automated ML pipelines and CI/CD

A pipeline is created by running an MLRun **"workflow"**. The following code defines a workflow and writes it to a file in your local directory, with the file name **workflow.py**. The workflow describes a directed acyclic graph (DAG) which is executed using the `local`, `remote`, or `kubeflow` engines.

See running a multi-stage workflow. The defined pipeline includes the following steps:

- Generate/prepare the data (`ingest`).
- Train and the model (`train`).
- Deploy the model as a real-time serverless function (`serving`).

Note

A pipeline can also include continuous build integration and deployment (CI/CD) steps, such as building container images and deploying models.

```
%%writefile './workflow.py'

from kfp import dsl
import mlrun

# Create a Kubeflow Pipelines pipeline
@dsl.pipeline(name="breast-cancer-demo")
def pipeline(model_name="cancer-classifier"):
    # Run the ingestion function with the new image and params
    ingest = mlrun.run_function(
        "data-prep",
        name="get-data",
        outputs=["dataset"],
    )

    # Train a model using the auto_trainer hub function
    train = mlrun.run_function(
        "hub://auto_trainer",
        inputs={"dataset": ingest.outputs["dataset"]},
        params = {
            "model_class": "sklearn.ensemble.RandomForestClassifier",
            "train_test_split_size": 0.2,
            "label_columns": "label",
            "model_name": model_name,
        },
        handler='train',
        outputs=["model"],
    )

    # Deploy the trained model as a serverless function
    serving_fn = mlrun.new_function("serving", image="mlrun/mlrun", kind="serving")
    serving_fn.with_code(body=" ")
    mlrun.deploy_function(
        serving_fn,
        models=[
```

(continues on next page)

(continued from previous page)

```
            {
                "key": model_name,
                "model_path": train.outputs["model"],
                "class_name": 'mlrun.frameworks.sklearn.SklearnModelServer',
            }
        ],
    )
```

```
Writing ./workflow.py
```

Run the workflow

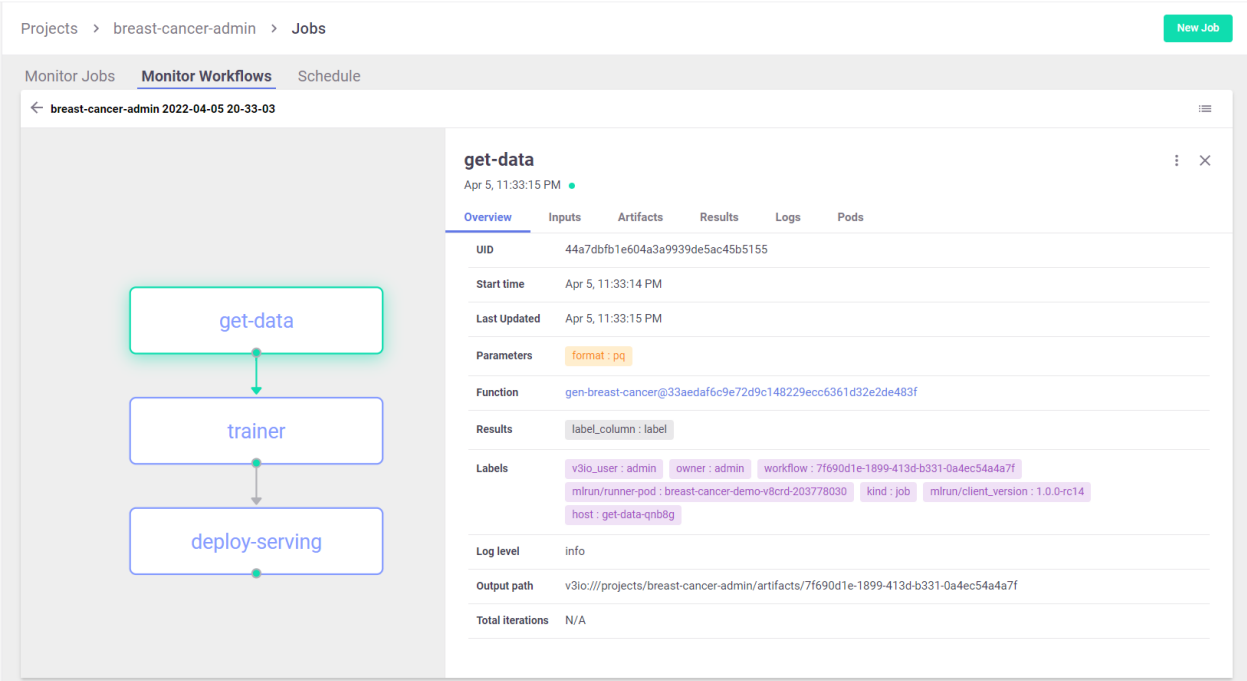
```
# Run the workflow
run_id = project.run(
    workflow_path="./workflow.py",
    arguments={"model_name": "cancer-classifier"},
    watch=True)
```

```
<IPython.core.display.HTML object>
```

```
<graphviz.graphs.Digraph at 0x7fd96f32d1c0>
```

```
<IPython.core.display.HTML object>
```

View the pipeline in MLRun UI



Run workflows using the CLI

With MLRun you can use a single command to load the code from local dir or remote archive (Git, zip, ...) and execute

a pipeline. This can be very useful for integration with CI/CD frameworks and practices. See CI/CD integration for more details.

The following command loads the project from the current dir (`.`) and executes the workflow with an argument, for running locally (without k8s).

```
mlrun project -r ./workflow.py -w -a model_name=classifier2 .!mlrun project -r ./
↪workflow.py -w -a model_name=classifier2 .
```

3.4.5 Test the deployed model endpoint

Now that your model is deployed using the pipeline, you can invoke it as usual:

```
serving_fn = project.get_function("serving")
```

```
# Create a mock (simulator of the real-time function)
my_data = {"inputs"
           :[[
               1.371e+01, 2.083e+01, 9.020e+01, 5.779e+02, 1.189e-01, 1.645e-01,
               9.366e-02, 5.985e-02, 2.196e-01, 7.451e-02, 5.835e-01, 1.377e+00,
               3.856e+00, 5.096e+01, 8.805e-03, 3.029e-02, 2.488e-02, 1.448e-02,
               1.486e-02, 5.412e-03, 1.706e+01, 2.814e+01, 1.106e+02, 8.970e+02,
               1.654e-01, 3.682e-01, 2.678e-01, 1.556e-01, 3.196e-01, 1.151e-01]
            ]
}
serving_fn.invoke("/v2/models/cancer-classifier/infer", body=my_data)
```

```
> 2022-09-20 15:09:02,664 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-tutorial-iguazio-serving.default-tenant.svc.cluster.local:8080/v2/models/cancer-
↪classifier/infer'}
```

```
{'id': '7ecaf987-bd79-470e-b930-19959808b678',
 'model_name': 'cancer-classifier',
 'outputs': [0]}
```

3.4.6 Done!

Congratulations! You've completed Part 4 of the MLRun getting-started tutorial. To continue, proceed to Part 5 Model monitoring and drift detection.

You might also want to explore the following demos:

- For an example of distributed training pipeline using TensorFlow, Keras, and PyTorch, see the **mask detection demo**.
- To learn more about deploying live endpoints and concept drift, see the **network-operations (NetOps) demo**.
- To learn about using the feature store to process raw transactions and events in real-time and respond and block transactions before they occur, see the **Fraud prevention demo**.
- For an example of a pipeline that summarizes and extracts keywords from a news article URL, see the **News article summarization and keyword extraction via NLP**.

3.5 Model monitoring and drift detection

This tutorial illustrates leveraging the model monitoring capabilities of MLRun to deploy a model to a live endpoint and calculate data drift.

Make sure you have reviewed the basics in MLRun **Quick Start Tutorial**.

Tutorial steps:

- ***Create an MLRun project***
- ***Log a model with a given framework and training set***
- ***Import and deploy serving function***
- ***Simulate production traffic***
- ***View drift calculations and status***
- ***View detailed drift dashboards***

3.5.1 MLRun installation and configuration

Before running this notebook make sure `mlrun` is installed and that you have configured the access to the MLRun service.

```
# Install MLRun if not installed, run this only once (restart the notebook after the␣
↪install !!!)
%pip install mlrun
```

3.5.2 Set up the project

First, import the dependencies and create an MLRun project. This contains all of the models, functions, datasets, etc:

```
import os

import mlrun
import pandas as pd
```

```
project = mlrun.get_or_create_project(name="tutorial", context="./", user_project=True)
```

```
> 2022-09-21 08:58:03,005 [info] loaded project tutorial from MLRun DB
```

Note

This tutorial does not focus on training a model. Instead, it starts with a trained model and its corresponding training dataset.

Next, log the following model file and dataset to deploy and calculate data drift. The model is a AdaBoostClassifier from sklearn, and the dataset is in `csv` format.

```
model_path = mlrun.get_sample_path('models/model-monitoring/model.pkl')
training_set_path = mlrun.get_sample_path('data/model-monitoring/iris_dataset.csv')
```

3.5.3 Log the model with training data

Log the model using MLRun experiment tracking. This is usually done in a training pipeline, but you can also bring in your pre-trained models from other sources. See Working with data and model artifacts and Automated experiment tracking for more information.

```
model_name = "RandomForestClassifier"
```

```
model_artifact = project.log_model(
    key=model_name,
    model_file=model_path,
    framework="sklearn",
    training_set=pd.read_csv(training_set_path),
    label_column="label"
)
```

```
# the model artifact unique URI
model_artifact.uri
```

```
'store://models/tutorial-nick/RandomForestClassifier#0:latest'
```

3.5.4 Import and deploy the serving function

Import the model server function from the MLRun Function Hub. Additionally, mount the filesytem, add the model that was logged via experiment tracking, and enable drift detection.

The core line here is `serving_fn.set_tracking()` that creates the required infrastructure behind the scenes to perform drift detection. See the Model monitoring overview for more info on what is deployed.

```
# Import the serving function from the Function Hub and mount filesystem
serving_fn = mlrun.import_function('hub://v2_model_server', new_name="serving")

# Add the model to the serving function's routing spec
serving_fn.add_model(model_name, model_path=model_artifact.uri)

# Enable model monitoring
serving_fn.set_tracking()
```

Deploy the serving function with drift detection

Deploy the serving function with drift detection enabled with a single line of code:

```
mlrun.deploy_function(serving_fn)
```

```
> 2022-09-21 08:58:08,053 [info] Starting remote function deploy
2022-09-21 08:58:09  (info) Deploying function
2022-09-21 08:58:09  (info) Building
2022-09-21 08:58:10  (info) Staging files and preparing base images
2022-09-21 08:58:10  (info) Building processor image
2022-09-21 08:58:55  (info) Build complete
2022-09-21 08:59:03  (info) Function deploy complete
> 2022-09-21 08:59:04,232 [info] successfully deployed function: {'internal_invocation_
↪urls': ['nuclio-tutorial-nick-serving.default-tenant.svc.cluster.local:8080'],
↪'external_invocation_urls': ['tutorial-nick-serving-tutorial-nick.default-tenant.app.
↪us-sales-350.iguazio-cd1.com/']}
```

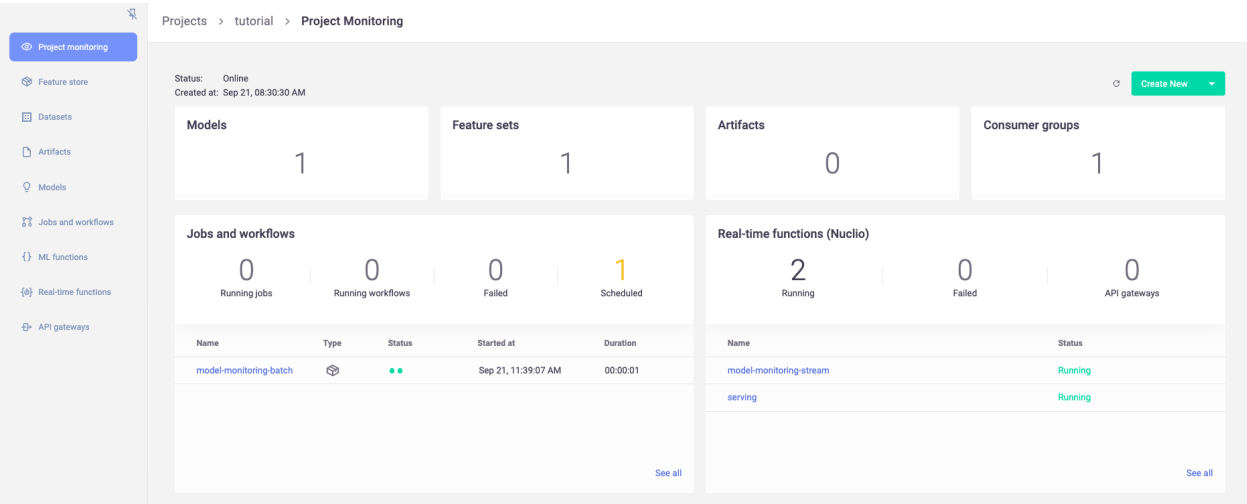
```
DeployStatus(state=ready, outputs={'endpoint': 'http://tutorial-nick-serving-tutorial-
↪nick.default-tenant.app.us-sales-350.iguazio-cd1.com/', 'name': 'tutorial-nick-serving
↪'})
```

3.5.5 View deployed resources

At this point, you should see the newly deployed model server, as well as a `model-monitoring-stream`, and a scheduled job (in yellow). The `model-monitoring-stream` collects, processes, and saves the incoming requests to the model server. The scheduled job does the actual calculation (by default every hour).

Note

You will not see `model-monitoring-batch` jobs listed until they actually run (by default every hour).



3.5.6 Simulate production traffic

Next, use the following code to simulate incoming production data using elements from the training set. Because the data is coming from the same training set you logged, you should not expect any data drift.

Note

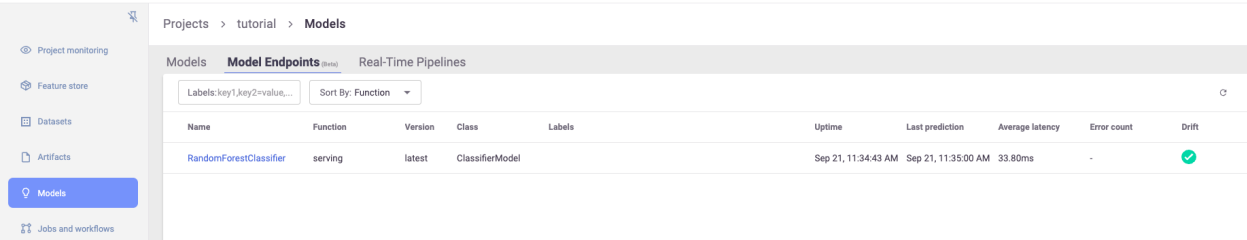
By default, the drift calculation starts via the scheduled hourly batch job after receiving 10,000 incoming requests.

```
import json
import logging
from random import choice, uniform
from time import sleep

from tqdm import tqdm

# Suppress print messages
logging.getLogger(name="mlrun").setLevel(logging.WARNING)

# Get training set as list
iris_data = pd.read_csv(training_set_path).drop("label", axis=1).to_dict(orient="split")[
↪"data"]

# Simulate traffic using random elements from training set
for i in tqdm(range(12_000)):
    data_point = choice(iris_data)
    serving_fn.invoke(f'v2/models/{model_name}/infer', json.dumps({'inputs': [data_
↪point]}))

# Resume normal logging
logging.getLogger(name="mlrun").setLevel(logging.INFO)
```

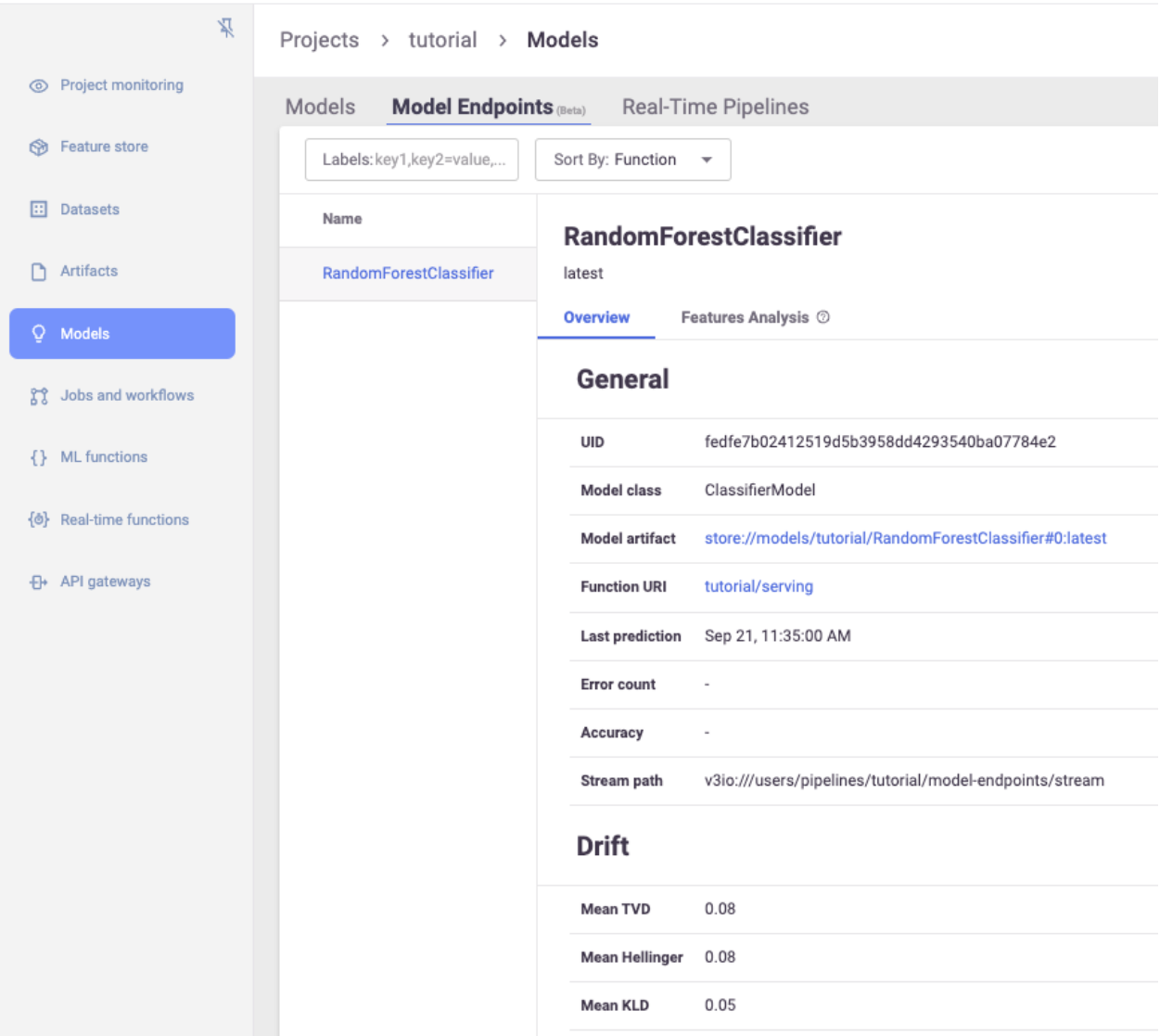
```
100%|| 12000/12000 [06:45<00:00, 29.63it/s]
```

3.5.7 View drift calculations and status

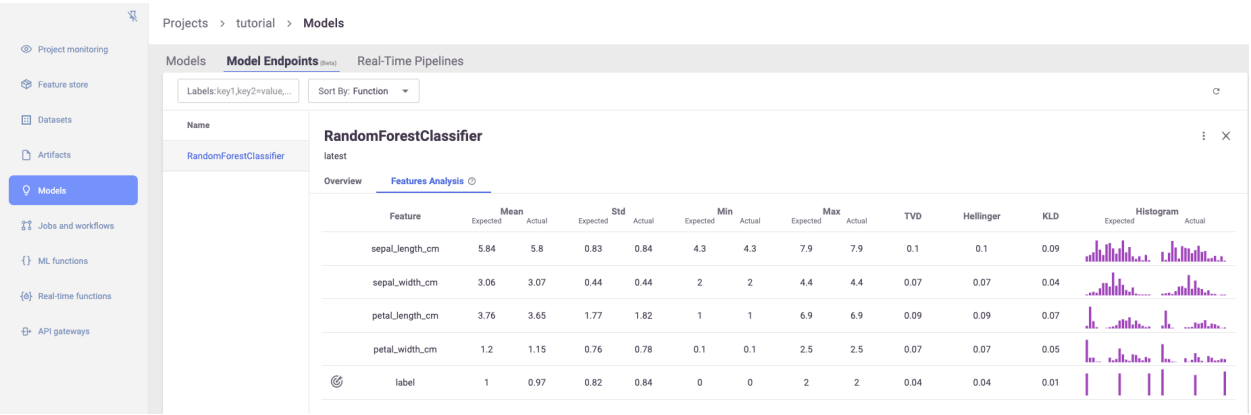
Once data drift has been calculated, you can view it in the MLRun UI. This includes a high level overview of the model status:



A more detailed view on model information and overall drift metrics:



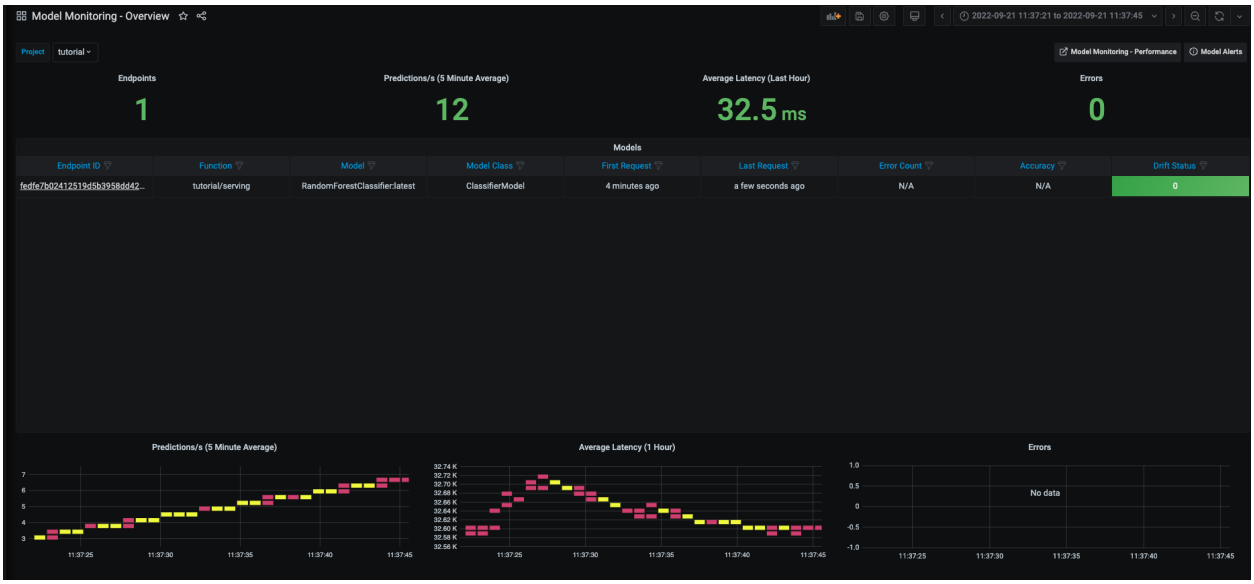
As well as a view for feature-level distributions and drift metrics:



3.5.8 View detailed drift dashboards

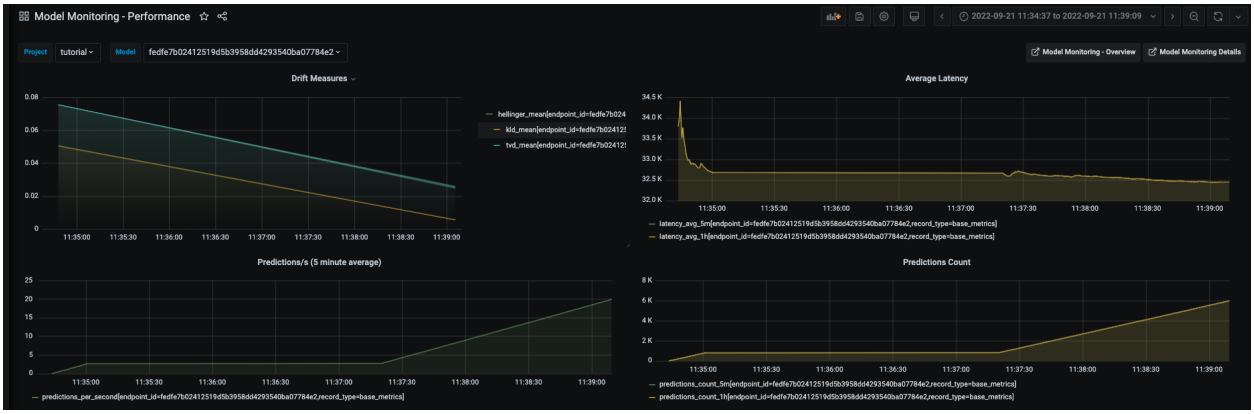
Finally, there are also more detailed Grafana dashboards that show additional information on each model in the project:

For more information on accessing these dashboards, see Model monitoring using Grafana dashboards.



Graphs of individual features over time:

As well as drift and operational metrics over time:



3.6 Add MLOps to existing code

This tutorial showcases how easy it is to apply MLRun on your existing code. With only 7 lines of code, you get:

- Experiment tracking — Track every single run of your experiment to learn what yielded the best results.
- Automatic Logging — Log datasets, metrics results and plots with one line of code. MLRun takes care for all the rest.
- Parameterization — Enable running your code with different parameters, run hyperparameters tuning and get the most out of your code.
- Resource management — Control the amount of resources available for your experiment.

Use this kaggle code by Sylas as an example, part of the competition New York City Taxi Fare Prediction.

Tutorial steps:

- ***Get the data***
- ***Code review***
- ***Run the script with MLRun***
- ***Review outputs***

3.6.1 Get the data

You can download the original data from kaggle. However, since the original data is 5.7GB in size, this demo uses sampled data. Since this demo uses MLRun's `DataItem` to pass the datasets, the sampled data is downloaded automatically. However, if you want to look at the data, you can download it: training set, and testing set.

3.6.2 Code review

Use the original code with the **minimum** changes required to apply MLRun to it. The code itself is straightforward:

1. Read the training data and perform feature engineering on it to preprocess it for training.
2. Train a LightGBM regression model using LightGBM's `train` function.
3. Read the testing data and save the contest expected submission file.

You can **Download the `script.py` file [Download here]**, or copy / paste it from here:

Show code

```
import gc

import lightgbm as lgbm
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

# [MLRun] Import MLRun:
import mlrun
from mlrun.frameworks.lgbm import apply_mlrun

# [MLRun] Get MLRun's context:
context = mlrun.get_or_create_ctx("apply-mlrun-tutorial")

# [MLRun] Reading train data from context instead of local file:
train_df = context.get_input("train_set", "./train.csv").as_df()
# train_df =  pd.read_csv('./train.csv')

# Drop rows with null values
train_df = train_df.dropna(how="any", axis="rows")


def clean_df(df):
    return df[
        (df.fare_amount > 0)
        & (df.fare_amount <= 500)
        &
        # (df.passenger_count >= 0) & (df.passenger_count <= 8)  &
        (
            (df.pickup_longitude != 0)
            & (df.pickup_latitude != 0)
            & (df.dropoff_longitude != 0)
            & (df.dropoff_latitude != 0)
        )
    ]


train_df = clean_df(train_df)
```

(continues on next page)

(continued from previous page)

```
# To Compute Haversine distance
def sphere_dist(pickup_lat, pickup_lon, dropoff_lat, dropoff_lon):
    """
    Return distance along great radius between pickup and dropoff coordinates.
    """
    # Define earth radius (km)
    R_earth = 6371
    # Convert degrees to radians
    pickup_lat, pickup_lon, dropoff_lat, dropoff_lon = map(
        np.radians, [pickup_lat, pickup_lon, dropoff_lat, dropoff_lon]
    )
    # Compute distances along lat, lon dimensions
    dlat = dropoff_lat - pickup_lat
    dlon = dropoff_lon - pickup_lon

    # Compute haversine distance
    a = (
        np.sin(dlat / 2.0) ** 2
        + np.cos(pickup_lat) * np.cos(dropoff_lat) * np.sin(dlon / 2.0) ** 2
    )
    return 2 * R_earth * np.arcsin(np.sqrt(a))


def sphere_dist_bear(pickup_lat, pickup_lon, dropoff_lat, dropoff_lon):
    """
    Return distance along great radius between pickup and dropoff coordinates.
    """
    # Convert degrees to radians
    pickup_lat, pickup_lon, dropoff_lat, dropoff_lon = map(
        np.radians, [pickup_lat, pickup_lon, dropoff_lat, dropoff_lon]
    )
    # Compute distances along lat, lon dimensions
    dlon = pickup_lon - dropoff_lon

    # Compute bearing distance
    a = np.arctan2(
        np.sin(dlon * np.cos(dropoff_lat)),
        np.cos(pickup_lat) * np.sin(dropoff_lat)
        - np.sin(pickup_lat) * np.cos(dropoff_lat) * np.cos(dlon),
    )
    return a


def radian_conv(degree):
    """
    Return radian.
    """
    return np.radians(degree)


def add_airport_dist(dataset):
    """
```

(continues on next page)

(continued from previous page)

```
    Return minumum distance from pickup or dropoff coordinates to each airport.
    JFK: John F. Kennedy International Airport
    EWR: Newark Liberty International Airport
    LGA: LaGuardia Airport
    SOL: Statue of Liberty
    NYC: Newyork Central
    """
    jfk_coord = (40.639722, -73.778889)
    ewr_coord = (40.6925, -74.168611)
    lga_coord = (40.77725, -73.872611)
    sol_coord = (40.6892, -74.0445)  # Statue of Liberty
    nyc_coord = (40.7141667, -74.0063889)

    pickup_lat = dataset["pickup_latitude"]
    dropoff_lat = dataset["dropoff_latitude"]
    pickup_lon = dataset["pickup_longitude"]
    dropoff_lon = dataset["dropoff_longitude"]

    pickup_jfk = sphere_dist(pickup_lat, pickup_lon, jfk_coord[0], jfk_coord[1])
    dropoff_jfk = sphere_dist(jfk_coord[0], jfk_coord[1], dropoff_lat, dropoff_lon)
    pickup_ewr = sphere_dist(pickup_lat, pickup_lon, ewr_coord[0], ewr_coord[1])
    dropoff_ewr = sphere_dist(ewr_coord[0], ewr_coord[1], dropoff_lat, dropoff_lon)
    pickup_lga = sphere_dist(pickup_lat, pickup_lon, lga_coord[0], lga_coord[1])
    dropoff_lga = sphere_dist(lga_coord[0], lga_coord[1], dropoff_lat, dropoff_lon)
    pickup_sol = sphere_dist(pickup_lat, pickup_lon, sol_coord[0], sol_coord[1])
    dropoff_sol = sphere_dist(sol_coord[0], sol_coord[1], dropoff_lat, dropoff_lon)
    pickup_nyc = sphere_dist(pickup_lat, pickup_lon, nyc_coord[0], nyc_coord[1])
    dropoff_nyc = sphere_dist(nyc_coord[0], nyc_coord[1], dropoff_lat, dropoff_lon)

    dataset["jfk_dist"] = pickup_jfk + dropoff_jfk
    dataset["ewr_dist"] = pickup_ewr + dropoff_ewr
    dataset["lga_dist"] = pickup_lga + dropoff_lga
    dataset["sol_dist"] = pickup_sol + dropoff_sol
    dataset["nyc_dist"] = pickup_nyc + dropoff_nyc

    return dataset


def add_datetime_info(dataset):
    # Convert to datetime format
    dataset["pickup_datetime"] = pd.to_datetime(
        dataset["pickup_datetime"], format="%Y-%m-%d %H:%M:%S UTC"
    )

    dataset["hour"] = dataset.pickup_datetime.dt.hour
    dataset["day"] = dataset.pickup_datetime.dt.day
    dataset["month"] = dataset.pickup_datetime.dt.month
    dataset["weekday"] = dataset.pickup_datetime.dt.weekday
    dataset["year"] = dataset.pickup_datetime.dt.year

    return dataset
```

(continues on next page)

(continued from previous page)

```
train_df = add_datetime_info(train_df)
train_df = add_airport_dist(train_df)
train_df["distance"] = sphere_dist(
    train_df["pickup_latitude"],
    train_df["pickup_longitude"],
    train_df["dropoff_latitude"],
    train_df["dropoff_longitude"],
)

train_df["bearing"] = sphere_dist_bear(
    train_df["pickup_latitude"],
    train_df["pickup_longitude"],
    train_df["dropoff_latitude"],
    train_df["dropoff_longitude"],
)
train_df["pickup_latitude"] = radian_conv(train_df["pickup_latitude"])
train_df["pickup_longitude"] = radian_conv(train_df["pickup_longitude"])
train_df["dropoff_latitude"] = radian_conv(train_df["dropoff_latitude"])
train_df["dropoff_longitude"] = radian_conv(train_df["dropoff_longitude"])


train_df.drop(columns=["key", "pickup_datetime"], inplace=True)

y = train_df["fare_amount"]
train_df = train_df.drop(columns=["fare_amount"])


print(train_df.head())

x_train, x_test, y_train, y_test = train_test_split(
    train_df, y, random_state=123, test_size=0.10
)

del train_df
del y
gc.collect()

params = {
    "boosting_type": "gbdt",
    "objective": "regression",
    "nthread": 4,
    "num_leaves": 31,
    "learning_rate": 0.05,
    "max_depth": -1,
    "subsample": 0.8,
    "bagging_fraction": 1,
    "max_bin": 5000,
    "bagging_freq": 20,
    "colsample_bytree": 0.6,
    "metric": "rmse",
    "min_split_gain": 0.5,
```

(continues on next page)

(continued from previous page)

```
    "min_child_weight": 1,
    "min_child_samples": 10,
    "scale_pos_weight": 1,
    "zero_as_missing": True,
    "seed": 0,
    "num_rounds": 50000,
}

train_set = lgbm.Dataset(
    x_train,
    y_train,
    silent=False,
    categorical_feature=["year", "month", "day", "weekday"],
)
valid_set = lgbm.Dataset(
    x_test,
    y_test,
    silent=False,
    categorical_feature=["year", "month", "day", "weekday"],
)

# [MLRun] Apply MLRun on the LightGBM module:
apply_mlrun(context=context)

model = lgbm.train(
    params,
    train_set=train_set,
    num_boost_round=10000,
    early_stopping_rounds=500,
    valid_sets=[valid_set],
)
del x_train
del y_train
del x_test
del y_test
gc.collect()

# [MLRun] Reading test data from context instead of local file:
test_df = context.get_input("test_set", "./test.csv").as_df()
# test_df =  pd.read_csv('./test.csv')
print(test_df.head())
test_df = add_datetime_info(test_df)
test_df = add_airport_dist(test_df)
test_df["distance"] = sphere_dist(
    test_df["pickup_latitude"],
    test_df["pickup_longitude"],
    test_df["dropoff_latitude"],
    test_df["dropoff_longitude"],
)

test_df["bearing"] = sphere_dist_bear(
    test_df["pickup_latitude"],
```

(continues on next page)

(continued from previous page)

```
    test_df["pickup_longitude"],
    test_df["dropoff_latitude"],
    test_df["dropoff_longitude"],
)
test_df["pickup_latitude"] = radian_conv(test_df["pickup_latitude"])
test_df["pickup_longitude"] = radian_conv(test_df["pickup_longitude"])
test_df["dropoff_latitude"] = radian_conv(test_df["dropoff_latitude"])
test_df["dropoff_longitude"] = radian_conv(test_df["dropoff_longitude"])


test_key = test_df["key"]
test_df = test_df.drop(columns=["key", "pickup_datetime"])

# Predict from test set
prediction = model.predict(test_df, num_iteration=model.best_iteration)
submission = pd.DataFrame({"key": test_key, "fare_amount": prediction})

# [MLRun] Log the submission instead of saving it locally:
context.log_dataset(key="taxi_fare_submission", df=submission, format="csv")
# submission.to_csv('taxi_fare_submission.csv',index=False)
```

This demo focuses on reviewing the changes / additions made to the original code so that you can apply MLRun on top of it. **Seven lines of code are added / replaced** as you can see in the sections below:

Initialization

Imports

On lines 9-10, add 2 imports:

- `mlrun` — Import MLRun of course.
- `apply_mlrun` — Use the `apply_mlrun` function from MLRun's frameworks, a sub-package for common ML/DL frameworks integrations with MLRun.

```
import mlrun
from mlrun.frameworks.lgbm import apply_mlrun
```

MLRun context

To get parameters and inputs into the code, you need to get MLRun's context. Use the function `get_or_create_ctx`.

Line 13:

```
context = mlrun.get_or_create_ctx("apply-mlrun-tutorial")
```

Get Training Set

In the original code the training set was read from a local file. Now you want to get it from the user who runs the code. Use the context to get the `"training_set"` input by using the `get_input` method. To maintain the original logic, include the default path for when the training set was not provided by the user.

Line 16:

```
train_df = context.get_input("train_set", "./train.csv").as_df()
# Instead of: `train_df =  pd.read_csv('./train.csv')`
```

Apply MLRun

Now use the `apply_mlrun` function from MLRun's LightGBM framework integration. MLRun automatically wraps the LightGBM module and enables automatic logging and evaluation.

Line 219:

```
apply_mlrun(context=context)
```

Logging the dataset

Similar to the way you got the training set, you get the test dataset as an input from the MLRun content.

Line 235:

```
test_df = context.get_input("test_set", "./test.csv").as_df()
# Instead of: `test_df =  pd.read_csv('./test.csv')`
```

Save the submission

Finally, instead of saving the result locally, log the submission to MLRun.

Line 267:

```
context.log_dataset(key="taxi_fare_submission", df=submission, format="csv")
# Instead of: `submission.to_csv('taxi_fare_submission.csv',index=False)`
```

3.6.3 Run the script with MLRun

Now you can run the script and see MLRun in action.

```
import mlrun
```

Create a project

Create a project using the function `get_or_create_project`. To read more about MLRun projects, see Projects.

```
project = mlrun.get_or_create_project(name="apply-mlrun-tutorial", context="./", user_
↪project=True)
```

```
> 2022-08-09 18:21:26,785 [info] loaded project apply-mlrun-tutorial from MLRun DB
```

Create a function

Create an MLRun function using the function `code_to_function`. To read more about MLRun functions, see *Serverless functions*.

```
script_function = mlrun.code_to_function(
    filename="./src/script.py",
    name="apply-mlrun-tutorial-function",
    kind="job",
    image="mlrun/ml-models"
)
```

```
<mlrun.runtimes.kubejob.KubejobRuntime at 0x7f20a5dfe250>
```

Run the function

Now you can run the function, providing it with the inputs you want. Use the datasets links to send them to the function. MLRun downloads and reads them into `pd.DataFrame` automatically.

```
script_run = script_function.run(
    inputs={
        "train_set": "https://s3.us-east-1.wasabisys.com/iguazio/data/nyc-taxi/train.csv
↪",
        "test_set": "https://s3.us-east-1.wasabisys.com/iguazio/data/nyc-taxi/test.csv"
    },
)
```

```
> 2022-08-09 18:21:26,851 [info] starting run apply-mlrun-tutorial-function␣
↪uid=8d82ef16a15d4151a16060c13b133170 DB=http://mlrun-api:8080
> 2022-08-09 18:21:27,017 [info] handler was not provided running main (./script.py)
> 2022-08-09 18:21:39,330 [info] logging run results to: http://mlrun-api:8080
   pickup_longitude  pickup_latitude  ...  distance   bearing
0         -1.288826         0.710721  ...  1.030764 -2.918897
1         -1.291824         0.710546  ...  8.450134 -0.375217
2         -1.291242         0.711418  ...  1.389525  2.599961
3         -1.291319         0.710927  ...  2.799270  0.133905
4         -1.290987         0.711536  ...  1.999157 -0.502703

[5 rows x 17 columns]
[LightGBM] [Warning] bagging_fraction is set=1, subsample=0.8 will be ignored. Current␣
↪value: bagging_fraction=1
[LightGBM] [Warning] Met categorical feature which contains sparse values. Consider␣
↪renumbering to consecutive integers started from zero
```

(continues on next page)

(continued from previous page)

```
[LightGBM] [Warning] bagging_fraction is set=1, subsample=0.8 will be ignored. Current␣
↪value: bagging_fraction=1
[LightGBM] [Warning] Auto-choosing col-wise multi-threading, the overhead of testing was␣
↪0.008352 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 55092
[LightGBM] [Info] Number of data points in the train set: 194071, number of used␣
↪features: 17
[LightGBM] [Warning] bagging_fraction is set=1, subsample=0.8 will be ignored. Current␣
↪value: bagging_fraction=1
[LightGBM] [Info] Start training from score 11.335573
                           key  ... passenger_count
0  2015-01-27 13:08:24.0000002  ...                1
1  2015-01-27 13:08:24.0000003  ...                1
2  2011-10-08 11:53:44.0000002  ...                1
3  2012-12-01 21:12:12.0000002  ...                1
4  2012-12-01 21:12:12.0000003  ...                1

[5 rows x 7 columns]
```

```
<IPython.core.display.HTML object>
```

```
> 2022-08-09 18:22:02,987 [info] run executed, status=completed
```

3.6.4 Review outputs

To view the outputs yielded by the MLRun automatic logging and evaluation, call the `outputs` property on the run object:

```
script_run.outputs
```

```
{'valid_0_rmse': 3.905279481685527,
 'valid_0_rmse_plot': 'v3io:///projects/apply-mlrun-tutorial-guyl/artifacts/apply-mlrun-
↪tutorial-function/0/valid_0_rmse_plot.html',
 'valid_0-feature-importance': 'v3io:///projects/apply-mlrun-tutorial-guyl/artifacts/
↪apply-mlrun-tutorial-function/0/valid_0-feature-importance.html',
 'valid_0': 'store://artifacts/apply-mlrun-tutorial-guyl/apply-mlrun-tutorial-function_
↪valid_0:8d82ef16a15d4151a16060c13b133170',
 'model': 'store://artifacts/apply-mlrun-tutorial-guyl/
↪model:8d82ef16a15d4151a16060c13b133170',
 'taxi_fare_submission': 'store://artifacts/apply-mlrun-tutorial-guyl/apply-mlrun-
↪tutorial-function_taxi_fare_submission:8d82ef16a15d4151a16060c13b133170'}
```

MLRun **automatically detects all the metrics calculated** and collects the data along with the training. Here there was one validation set named `valid_0` and the RMSE metric was calculated on it. You can see the RMSE values per iteration plot and the final score including the features importance plot.

You can explore the different artifacts by calling the `artifact` function like this:

```
script_run.artifact('valid_0_rmse_plot').show()
```

```
<IPython.core.display.HTML object>
```

```
script_run.artifact('valid_0-feature-importance').show()
```

```
<IPython.core.display.HTML object>
```

And of course, you can also see the submission that was logged:

```
script_run.artifact('taxi_fare_submission').show()
```

```
                            key  fare_amount
0     2015-01-27 13:08:24.0000002    10.281408
1     2015-01-27 13:08:24.0000003    11.019641
2     2011-10-08 11:53:44.0000002     4.898061
3     2012-12-01 21:12:12.0000002     7.758042
4     2012-12-01 21:12:12.0000003    15.298775
...                           ...          ...
9909  2015-05-10 12:37:51.0000002     9.117569
9910  2015-01-12 17:05:51.0000001    10.850885
9911  2015-04-19 20:44:15.0000001    55.048856
9912  2015-01-31 01:05:19.0000005    20.110280
9913  2015-01-18 14:06:23.0000006     7.081041

[9914 rows x 2 columns]
```

3.7 Batch inference and drift detection

This tutorial leverages a function from the MLRun Function Hub to perform batch inference using a logged model and a new prediction dataset. The function also calculates data drift by comparing the new prediction dataset with the original training set.

Make sure you have reviewed the basics in MLRun **Quick Start Tutorial**.

Tutorial steps:

- ***Set up an MLRun project***
- ***View the data***
- ***Log a model with a given framework and training set***
- ***Import and run the batch inference function***
- ***View predictions and drift status***

3.7.1 MLRun installation and Configuration

Before running this notebook make sure `mlrun` is installed and that you have configured the access to the MLRun service.

```
# Install MLRun if not installed, run this only once (restart the notebook after the␣
↪install !!!)
%pip install mlrun
```

3.7.2 Set up a project

First, import the dependencies and create an MLRun project. The project contains all of your models, functions, datasets, etc:

```
import mlrun
import os
import pandas as pd
```

```
project = mlrun.get_or_create_project("tutorial", context="./", user_project=True)
```

```
> 2022-09-20 19:15:36,113 [info] Created and saved project batch-predict: {'from_template
↪': None, 'overwrite': False, 'context': './', 'save': True}
```

Note

This tutorial does not focus on training a model. Instead, it starts with a trained model and its corresponding training and prediction dataset.

You will use the following model files and datasets to perform the batch prediction. The model is a DecisionTreeClassifier from sklearn and the datasets are in `parquet` format.

```
model_path = mlrun.get_sample_path('models/batch-predict/model.pkl')
training_set_path = mlrun.get_sample_path('data/batch-predict/training_set.parquet')
prediction_set_path = mlrun.get_sample_path('data/batch-predict/prediction_set.parquet')
```

3.7.3 View the data

The training data has 20 numerical features and a binary (0,1) label:

```
pd.read_parquet(training_set_path).head()
```

```
   feature_0  feature_1  feature_2  feature_3  feature_4  feature_5  \
0   0.572754   0.171079   0.403080   0.955429   0.272039   0.360277
1   0.623733  -0.149823  -1.410537  -0.729388  -1.996337  -1.213348
2   0.814168  -0.221412   0.020822   1.066718  -0.573164   0.067838
3   1.062279  -0.966309   0.341471  -0.737059   1.460671   0.367851
```

(continues on next page)

(continued from previous page)

```
4   0.195755   0.576332  -0.260496   0.841489   0.398269  -0.717972

   feature_6  feature_7  feature_8  feature_9  ...  feature_11  feature_12  \
0  -0.995429   0.437239   0.991556   0.010004  ...    0.112194   -0.319256
1   1.461307   1.187854  -1.790926  -0.981600  ...    0.428653   -0.503820
2   0.923045   0.338146   0.981413   1.481757  ...   -1.052559   -0.241873
3  -0.435336   0.445308  -0.655663  -0.196220  ...    0.641017    0.099059
4   0.810550  -1.058326   0.368610   0.606007  ...    0.195267    0.876144

   feature_13  feature_14  feature_15  feature_16  feature_17  feature_18  \
0   -0.392631   -0.290766    1.265054    1.037082   -1.200076    0.820992
1   -0.798035    2.038105   -3.080463    0.408561    1.647116   -0.838553
2   -1.232272   -0.010758    0.806800    0.661162    0.589018    0.522137
3    1.902592   -1.024929    0.030703   -0.198751   -0.342009   -1.286865
4    0.151615    0.094867    0.627353   -0.389023    0.662846   -0.857000

   feature_19  label
0    0.834868      0
1    0.680983      1
2   -0.924624      0
3   -1.118373      1
4    1.091218      1

[5 rows x 21 columns]
```

The prediciton data has 20 numerical features, but no label - this is what you will predict:

```
pd.read_parquet(prediction_set_path).head()
```

```
   feature_0  feature_1  feature_2  feature_3  feature_4  feature_5  \
0  -2.059506  -1.314291   2.721516  -2.132869  -0.693963   0.376643
1  -1.190382   0.891571   3.726070   0.673870  -0.252565  -0.729156
2  -0.996384  -0.099537   3.421476   0.162771  -1.143458  -1.026791
3  -0.289976  -1.680019   3.126478  -0.704451  -1.149112   1.174962
4  -0.294866   1.044919   2.924139   0.814049  -1.455054  -0.270432

   feature_6  feature_7  feature_8  feature_9  feature_10  feature_11  \
0   3.017790   3.876329  -1.294736   0.030773    0.401491    2.775699
1   2.646563   4.782729   0.318952  -0.781567    1.473632    1.101721
2   2.114702   2.517553  -0.154620  -0.465423   -1.723025    1.729386
3   2.860341   3.753661  -0.326119   2.128411   -0.508000    2.328688
4   3.380195   2.339669   1.029101  -1.171018   -1.459395    1.283565

   feature_12  feature_13  feature_14  feature_15  feature_16  feature_17  \
0    2.361580    0.173441    0.879510    1.141007    4.608280   -0.518388
1    3.723400   -0.466867   -0.056224    3.344701    0.194332    0.463992
2    2.820340   -1.041428   -0.331871    2.909172    2.138613   -0.046252
3    3.397321   -0.932060   -1.442370    2.058517    3.881936    2.090635
4    0.677006   -2.147444   -0.494150    3.222041    6.219348   -1.914110

   feature_18  feature_19
0    0.129690    2.794967
```

(continues on next page)

(continued from previous page)

```
1    0.292268    4.665876
2   -0.732631    4.716266
3   -0.045832    4.197315
4    0.317786    4.143443
```

3.7.4 Log the model with training data

Next, log the model using MLRun experiment tracking. This is usually done in a training pipeline, but you can also bring in your pre-trained models from other sources. See Working with data and model artifacts and Automated experiment tracking for more information.

In this example, you are logging a training set with the model for future comparison, however you can also directly pass in your training set to the batch prediction function.

```
model_artifact = project.log_model(
    key="model",
    model_file=model_path,
    framework="sklearn",
    training_set=pd.read_parquet(training_set_path),
    label_column="label"
)
```

```
# the model artifact unique URI
model_artifact.uri
```

```
'store://models/batch-predict/model#0:latest'
```

3.7.5 Import and run the batch inference function

Next, import the batch inference function from the MLRun Function Hub:

```
fn = mlrun.import_function("hub://batch_inference")
```

Run batch inference

Finally, perform the batch prediction by passing in your model and datasets. See the corresponding batch inference example notebook for an exhaustive list of other parameters that are supported:

```
run = project.run_function(
    fn,
    inputs={
        "dataset": prediction_set_path,
        # If you do not log a dataset with your model, you can pass it in here:
#       "sample_set" : training_set_path
    },
```

(continues on next page)

(continued from previous page)

```
    params={
        "model": model_artifact.uri,
        "perform_drift_analysis" : True,
    },
)
```

3.7.6 View predictions and drift status

These are the batch predicictions on the prediction set from the model:

```
run.artifact("prediction").as_df().head()
```

```
   feature_0  feature_1  feature_2  feature_3  feature_4  feature_5  \
0  -2.059506  -1.314291   2.721516  -2.132869  -0.693963   0.376643
1  -1.190382   0.891571   3.726070   0.673870  -0.252565  -0.729156
2  -0.996384  -0.099537   3.421476   0.162771  -1.143458  -1.026791
3  -0.289976  -1.680019   3.126478  -0.704451  -1.149112   1.174962
4  -0.294866   1.044919   2.924139   0.814049  -1.455054  -0.270432

   feature_6  feature_7  feature_8  feature_9  ...  feature_11  feature_12  \
0   3.017790   3.876329  -1.294736   0.030773  ...    2.775699    2.361580
1   2.646563   4.782729   0.318952  -0.781567  ...    1.101721    3.723400
2   2.114702   2.517553  -0.154620  -0.465423  ...    1.729386    2.820340
3   2.860341   3.753661  -0.326119   2.128411  ...    2.328688    3.397321
4   3.380195   2.339669   1.029101  -1.171018  ...    1.283565    0.677006

   feature_13  feature_14  feature_15  feature_16  feature_17  feature_18  \
0    0.173441    0.879510    1.141007    4.608280   -0.518388    0.129690
1   -0.466867   -0.056224    3.344701    0.194332    0.463992    0.292268
2   -1.041428   -0.331871    2.909172    2.138613   -0.046252   -0.732631
3   -0.932060   -1.442370    2.058517    3.881936    2.090635   -0.045832
4   -2.147444   -0.494150    3.222041    6.219348   -1.914110    0.317786

   feature_19  label
0    2.794967      0
1    4.665876      0
2    4.716266      0
3    4.197315      1
4    4.143443      1

[5 rows x 21 columns]
```

There is also a drift table plot that compares the drift between the training data and prediction data per feature:

```
run.artifact("drift_table_plot").show()
```

	Count		Mean		Std		Min		Max		Tvd	Hellinger	Kld	Histograms
	Sample	Input	Sample	Input	Sample	Input	Sample	Input	Sample	Input				sample input
feature_10	2.5k	2.5k	0.0099619	-0.028823	1.0024	1.015	-3.468	-3.252	3.6221	3.4528	0.0404	0.044734	0.016108	
feature_0	2.5k	2.5k	-0.0040279	0.021953	0.99581	0.98954	-3.549	-3.267	3.4002	3.4347	0.0226	0.033574	0.0074546	
feature_17	2.5k	2.5k	-0.0034057	-0.018602	1.0027	0.98876	-3.481	-3.457	3.2288	3.3332	0.0328	0.038956	0.0090139	
feature_3	2.5k	2.5k	0.015564	-0.0099402	0.98579	0.99494	-3.986	-3.464	3.5643	3.217	0.0384	0.04914	0.017332	
feature_15	2.5k	2.5k	0.019173	2.9823	1.2	1.3556	-3.654	-1.256	3.2802	6.4363	0.519	0.74682	3.7154	
feature_12	2.5k	2.5k	0.020975	3.0041	0.99692	1.1486	-3.07	-0.86306	3.0677	6.7735	0.599	0.80796	4.5742	
feature_4	2.5k	2.5k	0.0043483	0.0038642	1.2859	1.2864	-3.757	-3.742	4.375	4.7429	0.0386	0.046911	0.01621	
feature_7	2.5k	2.5k	-0.022209	3.0088	1.0213	1.1624	-3.3	-0.62968	3.4935	6.7219	0.6646	0.79498	4.9492	
feature_6	2.5k	2.5k	-0.011094	3.0164	1.0216	1.1284	-3.258	-1.085	3.4455	7.1375	0.6598	0.79261	4.6801	
feature_16	2.5k	2.5k	0.017349	3.0134	0.97533	1.1673	-3.592	-0.82201	3.2731	7.0476	0.636	0.80032	4.6827	
feature_19	2.5k	2.5k	0.009379	2.9819	0.99404	1.1578	-3.745	-0.42378	4.5189	7.1048	0.7812	0.79934	6.9668	
feature_18	2.5k	2.5k	0.00756	0.01097	0.98851	1.0066	-3.753	-3.349	3.8614	3.4863	0.0424	0.046475	0.015894	
feature_8	2.5k	2.5k	0.014922	-0.023379	0.82918	0.84283	-2.13	-2.085	1.4455	1.6216	0.0384	0.03972	0.0094354	
label	2.5k	2.5k	0.4952	0.5156	0.50008	0.49986	0	0	1	1	0.0204	0.014427	0.0016651	
feature_14	2.5k	2.5k	0.013093	-0.0087792	1.0037	1.0024	-3.563	-3.216	3.3913	3.1733	0.038	0.054729	0.022306	
feature_1	2.5k	2.5k	0.018511	0.00019738	1.0159	0.99085	-3.159	-3.086	3.8723	4.213	0.0434	0.046301	0.014582	
feature_5	2.5k	2.5k	-0.029516	-0.0017621	0.98998	0.99237	-3.288	-3.235	3.0054	3.3326	0.0496	0.054084	0.017044	
feature_13	2.5k	2.5k	-0.0056543	0.019551	1.3625	1.3556	-4.484	-4.512	4.283	4.629	0.0396	0.045194	0.013241	
feature_11	2.5k	2.5k	0.0060043	3.0076	0.97702	1.1349	-3.991	-0.61732	3.2947	6.5215	0.6384	0.80589	4.8005	
feature_2	2.5k	2.5k	-0.009449	2.992	0.98855	1.1657	-3.242	-0.58678	3.697	7.0218	0.6882	0.79006	5.2144	
feature_9	2.5k	2.5k	0.023047	0.045638	0.98049	0.98343	-3.287	-4.242	3.6956	3.0063	0.0428	0.046567	0.012618	

Finally, you also get a numerical drift metric and boolean flag denoting whether or not data drift is detected:

```
run.status.results
```

```
{'drift_status': False, 'drift_metric': 0.29934242566253266}
```

```
# Data/concept drift per feature
import json
json.loads(run.artifact("features_drift_results").get())
```

```
{'feature_6': 0.7262042202197605,
 'feature_2': 0.7391279921664593,
 'feature_16': 0.7181622588902428,
 'feature_0': 0.028086840976606773,
 'feature_3': 0.043769819014849734,
 'feature_18': 0.04443732609382538,
 'feature_15': 0.6329075683793959,
```

(continues on next page)

(continued from previous page)

```
 'feature_1': 0.04485072701663093,
 'feature_12': 0.7034787615778625,
 'feature_14': 0.046364723781764774,
 'feature_8': 0.039060131873550404,
 'feature_10': 0.042567035578799796,
 'feature_11': 0.7221431701127441,
 'feature_5': 0.05184219833790496,
 'feature_9': 0.04468363504674985,
 'feature_19': 0.7902698698155215,
 'feature_4': 0.042755641152500176,
 'feature_13': 0.04239724655474124,
 'feature_7': 0.7297906294873706,
 'feature_17': 0.03587785749574268,
 'label': 0.017413285340161608}
```

3.7.7 Next steps

In a production setting, you probably want to incorporate this as part of a larger pipeline or application.

For example, if you use this function for the prediction capabilities, you can pass the `prediction` output as the input to another pipeline step, store it in an external location like S3, or send to an application or user.

If you use this function for the drift detection capabilities, you can use the `drift_status` and `drift_metrics` outputs to automate further pipeline steps, send a notification, or kick off a re-training pipeline.

3.8 Feature store example (stocks)

This notebook demonstrates the following:

- Generate features and feature-sets
- Build complex transformations and ingest to offline and real-time data stores
- Fetch feature vectors for training
- Save feature vectors for re-use in real-time pipelines
- Access features and their statistics in real-time

Note

By default, this demo works with the online feature store, which is currently not part of the Open Source MLRun default deployment.

In this section

3.8.1 Get started

Install the latest MLRun package and restart the notebook.

Setting up the environment and project:

```
import mlrun
mlrun.set_environment(project="stocks")
```

```
> 2021-05-23 09:04:04,507 [warning] Failed resolving version info. Ignoring and using␣
↪defaults
> 2021-05-23 09:04:07,033 [warning] Unable to parse server or client version. Assuming␣
↪compatible: {'server_version': '0.6.4-rc3', 'client_version': 'unstable'}
```

```
('stocks', 'v3io:///projects/{{run.project}}/artifacts')
```

3.8.2 Create sample data for demo

```
import pandas as pd
quotes = pd.DataFrame(
    {
        "time": [
            pd.Timestamp("2016-05-25 13:30:00.023"),
            pd.Timestamp("2016-05-25 13:30:00.023"),
            pd.Timestamp("2016-05-25 13:30:00.030"),
            pd.Timestamp("2016-05-25 13:30:00.041"),
            pd.Timestamp("2016-05-25 13:30:00.048"),
            pd.Timestamp("2016-05-25 13:30:00.049"),
            pd.Timestamp("2016-05-25 13:30:00.072"),
            pd.Timestamp("2016-05-25 13:30:00.075")
        ],
        "ticker": [
                "GOOG",
                "MSFT",
                "MSFT",
                "MSFT",
                "GOOG",
                "AAPL",
                "GOOG",
                "MSFT"
            ],
            "bid": [720.50, 51.95, 51.97, 51.99, 720.50, 97.99, 720.50, 52.01],
            "ask": [720.93, 51.96, 51.98, 52.00, 720.93, 98.01, 720.88, 52.03]
    }
)

trades = pd.DataFrame(
       {
            "time": [
                pd.Timestamp("2016-05-25 13:30:00.023"),
                pd.Timestamp("2016-05-25 13:30:00.038"),
                pd.Timestamp("2016-05-25 13:30:00.048"),
```

(continues on next page)

(continued from previous page)

```
                pd.Timestamp("2016-05-25 13:30:00.048"),
                pd.Timestamp("2016-05-25 13:30:00.048")
            ],
            "ticker": ["MSFT", "MSFT", "GOOG", "GOOG", "AAPL"],
            "price": [51.95, 51.95, 720.77, 720.92, 98.0],
            "quantity": [75, 155, 100, 100, 100]
        }
)


stocks = pd.DataFrame(
        {
            "ticker": ["MSFT", "GOOG", "AAPL"],
            "name": ["Microsoft Corporation", "Alphabet Inc", "Apple Inc"],
            "exchange": ["NASDAQ", "NASDAQ", "NASDAQ"]
        }
)

import datetime
def move_date(df, col):
    max_date = df[col].max()
    now_date = datetime.datetime.now()
    delta = now_date - max_date
    df[col] = df[col] + delta
    return df

quotes = move_date(quotes, "time")
trades = move_date(trades, "time")
```

View the demo data

```
quotes
```

```
                        time ticker     bid     ask
0 2021-05-23 09:04:07.013574   GOOG  720.50  720.93
1 2021-05-23 09:04:07.013574   MSFT   51.95   51.96
2 2021-05-23 09:04:07.020574   MSFT   51.97   51.98
3 2021-05-23 09:04:07.031574   MSFT   51.99   52.00
4 2021-05-23 09:04:07.038574   GOOG  720.50  720.93
5 2021-05-23 09:04:07.039574   AAPL   97.99   98.01
6 2021-05-23 09:04:07.062574   GOOG  720.50  720.88
7 2021-05-23 09:04:07.065574   MSFT   52.01   52.03
```

```
trades
```

```
                        time ticker   price  quantity
0 2021-05-23 09:04:07.041766   MSFT   51.95        75
1 2021-05-23 09:04:07.056766   MSFT   51.95       155
2 2021-05-23 09:04:07.066766   GOOG  720.77       100
```

(continues on next page)

(continued from previous page)

```
3 2021-05-23 09:04:07.066766   GOOG  720.92       100
4 2021-05-23 09:04:07.066766   AAPL   98.00       100
```

```
stocks
```

```
  ticker                  name exchange
0   MSFT  Microsoft Corporation   NASDAQ
1   GOOG            Alphabet Inc   NASDAQ
2   AAPL               Apple Inc   NASDAQ
```

3.8.3 Define, infer and ingest feature sets

```
import mlrun.feature_store as fstore
from mlrun.feature_store.steps import *
from mlrun.features import MinMaxValidator
```

Build and ingest simple feature set (stocks)

```
# add feature set without time column (stock ticker metadata)
stocks_set = fstore.FeatureSet("stocks", entities=[fstore.Entity("ticker")])
fstore.ingest(stocks_set, stocks, infer_options=fstore.InferOptions.default())
```

```
                        name exchange
ticker
MSFT    Microsoft Corporation   NASDAQ
GOOG              Alphabet Inc   NASDAQ
AAPL                 Apple Inc   NASDAQ
```

Build an advanced feature set - with feature engineering pipeline

Define a feature set with custom data processing and time aggregation functions:

```
# create a new feature set
quotes_set = fstore.FeatureSet("stock-quotes", entities=[fstore.Entity("ticker")])
```

Define a custom pipeline step (python class)

```
class MyMap(MapClass):
    def __init__(self, multiplier=1, **kwargs):
        super().__init__(**kwargs)
        self._multiplier = multiplier

    def do(self, event):
        event["multi"] = event["bid"] * self._multiplier
        return event
```

Build and show the transformatiom pipeline

Use `storey` stream processing classes along with library and custom classes:

```
quotes_set.graph.to("MyMap", multiplier=3)\
                .to("storey.Extend", _fn="({'extra': event['bid'] * 77})")\
                .to("storey.Filter", "filter", _fn="(event['bid'] > 51.92)")\
                .to(FeaturesetValidator())

quotes_set.add_aggregation("ask", ["sum", "max"], "1h", "10m", name="asks1")
quotes_set.add_aggregation("ask", ["sum", "max"], "5h", "10m", name="asks5")
quotes_set.add_aggregation("bid", ["min", "max"], "1h", "10m", name="bids")

# add feature validation policy
quotes_set["bid"] = fstore.Feature(validator=MinMaxValidator(min=52, severity="info"))

# add default target definitions and plot
quotes_set.set_targets()
quotes_set.plot(rankdir="LR", with_targets=True)
```

```
<graphviz.dot.Digraph at 0x7fa9a4154250>
```

Test and show the pipeline results locally (allow to quickly develop and debug)

```
fstore.preview(
    quotes_set,
    quotes,
    entity_columns=["ticker"],
    timestamp_key="time",
    options=fstore.InferOptions.default(),
)
```

```
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.013574 args={
↪'min': 52, 'value': 51.95}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.020574 args={
↪'min': 52, 'value': 51.97}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.031574 args={
↪'min': 52, 'value': 51.99}
```

```
        asks1_sum_1h  asks1_max_1h  asks5_sum_5h  asks5_max_5h  bids_min_1h  \
ticker
GOOG          720.93        720.93        720.93        720.93       720.50
MSFT           51.96         51.96         51.96         51.96        51.95
MSFT          103.94         51.98        103.94         51.98        51.95
MSFT          155.94         52.00        155.94         52.00        51.95
GOOG         1441.86        720.93       1441.86        720.93       720.50
AAPL           98.01         98.01         98.01         98.01        97.99
GOOG         2162.74        720.93       2162.74        720.93       720.50
MSFT          207.97         52.03        207.97         52.03        51.95

        bids_max_1h                       time     bid     ask    multi  \
ticker
GOOG         720.50 2021-05-23 09:04:07.013574  720.50  720.93  2161.50
MSFT          51.95 2021-05-23 09:04:07.013574   51.95   51.96   155.85
MSFT          51.97 2021-05-23 09:04:07.020574   51.97   51.98   155.91
MSFT          51.99 2021-05-23 09:04:07.031574   51.99   52.00   155.97
```

(continues on next page)

(continued from previous page)

```
GOOG         720.50 2021-05-23 09:04:07.038574  720.50  720.93  2161.50
AAPL          97.99 2021-05-23 09:04:07.039574   97.99   98.01   293.97
GOOG         720.50 2021-05-23 09:04:07.062574  720.50  720.88  2161.50
MSFT          52.01 2021-05-23 09:04:07.065574   52.01   52.03   156.03

           extra
ticker
GOOG    55478.50
MSFT     4000.15
MSFT     4001.69
MSFT     4003.23
GOOG    55478.50
AAPL     7545.23
GOOG    55478.50
MSFT     4004.77
```

```
# print the feature set object
print(quotes_set.to_yaml())
```

```
kind: FeatureSet
metadata:
  name: stock-quotes
spec:
  entities:
  - name: ticker
    value_type: str
  features:
  - name: asks1_sum_1h
    value_type: float
    aggregate: true
  - name: asks1_max_1h
    value_type: float
    aggregate: true
  - name: asks5_sum_5h
    value_type: float
    aggregate: true
  - name: asks5_max_5h
    value_type: float
    aggregate: true
  - name: bids_min_1h
    value_type: float
    aggregate: true
  - name: bids_max_1h
    value_type: float
    aggregate: true
  - name: bid
    value_type: float
    validator:
      kind: minmax
      severity: info
      min: 52
```

(continues on next page)

(continued from previous page)

```
  - name: ask
    value_type: float
  - name: multi
    value_type: float
  - name: extra
    value_type: float
partition_keys: []
timestamp_key: time
source:
  path: None
targets:
- name: parquet
  kind: parquet
- name: nosql
  kind: nosql
graph:
  states:
    MyMap:
      kind: task
      class_name: MyMap
      class_args:
        multiplier: 3
    storey.Extend:
      kind: task
      class_name: storey.Extend
      class_args:
        _fn: '({''extra'': event[''bid''] * 77})'
      after:
      - MyMap
    filter:
      kind: task
      class_name: storey.Filter
      class_args:
        _fn: (event['bid'] > 51.92)
      after:
      - storey.Extend
    FeaturesetValidator:
      kind: task
      class_name: mlrun.feature_store.steps.FeaturesetValidator
      class_args:
        featureset: .
        columns: null
      after:
      - filter
    Aggregates:
      kind: task
      class_name: storey.AggregateByKey
      class_args:
        aggregates:
        - name: asks1
          column: ask
          operations:
```

(continues on next page)

(continued from previous page)

```
            - sum
            - max
            windows:
            - 1h
            period: 10m
          - name: asks5
            column: ask
            operations:
            - sum
            - max
            windows:
            - 5h
            period: 10m
          - name: bids
            column: bid
            operations:
            - min
            - max
            windows:
            - 1h
            period: 10m
          table: .
        after:
        - FeaturesetValidator
  output_path: v3io:///projects/{{run.project}}/artifacts
status:
  state: created
  stats:
    ticker:
      count: 8
      unique: 3
      top: MSFT
      freq: 4
    asks1_sum_1h:
      count: 8.0
      mean: 617.9187499999999
      min: 51.96
      max: 2162.74
      std: 784.8779804245735
      hist:
      - - 4
        - 1
        - 0
        - 0
        - 0
        - 0
        - 1
        - 0
        - 0
        - 0
        - 0
        - 0
```

(continues on next page)

(continued from previous page)


```
      - 0
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 1
    - - 51.96
      - 157.499
      - 263.03799999999995
      - 368.57699999999994
      - 474.11599999999993
      - 579.655
      - 685.194
      - 790.733
      - 896.2719999999999
      - 1001.8109999999999
      - 1107.35
      - 1212.889
      - 1318.4279999999999
      - 1423.9669999999999
      - 1529.5059999999999
      - 1635.0449999999998
      - 1740.5839999999998
      - 1846.1229999999998
      - 1951.6619999999998
      - 2057.2009999999996
      - 2162.74
  asks1_max_1h:
    count: 8.0
    mean: 308.59625
    min: 51.96
    max: 720.93
    std: 341.7989955655851
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
```


(continues on next page)

(continued from previous page)

```
      - 0
      - 0
      - 0
      - 3
    - - 51.96
      - 85.4085
      - 118.857
      - 152.3055
      - 185.754
      - 219.2025
      - 252.65099999999998
      - 286.0995
      - 319.54799999999994
      - 352.9964999999999
      - 386.44499999999994
      - 419.89349999999996
      - 453.3419999999999
      - 486.7904999999999
      - 520.2389999999999
      - 553.6875
      - 587.136
      - 620.5844999999999
      - 654.0329999999999
      - 687.4815
      - 720.93
  asks5_sum_5h:
    count: 8.0
    mean: 617.9187499999999
    min: 51.96
    max: 2162.74
    std: 784.8779804245735
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 1
```

(continues on next page)

(continued from previous page)

```
    - - 51.96
      - 157.499
      - 263.03799999999995
      - 368.57699999999994
      - 474.11599999999993
      - 579.655
      - 685.194
      - 790.733
      - 896.2719999999999
      - 1001.8109999999999
      - 1107.35
      - 1212.889
      - 1318.4279999999999
      - 1423.9669999999999
      - 1529.5059999999999
      - 1635.0449999999998
      - 1740.5839999999998
      - 1846.1229999999998
      - 1951.6619999999998
      - 2057.2009999999996
      - 2162.74
  asks5_max_5h:
    count: 8.0
    mean: 308.59625
    min: 51.96
    max: 720.93
    std: 341.7989955655851
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 51.96
      - 85.4085
      - 118.857
      - 152.3055
```

(continues on next page)

(continued from previous page)

```
      - 185.754
      - 219.2025
      - 252.65099999999998
      - 286.0995
      - 319.54799999999994
      - 352.9964999999999
      - 386.44499999999994
      - 419.89349999999996
      - 453.3419999999999
      - 486.7904999999999
      - 520.2389999999999
      - 553.6875
      - 587.136
      - 620.5844999999999
      - 654.0329999999999
      - 687.4815
      - 720.93
  bids_min_1h:
    count: 8.0
    mean: 308.41125
    min: 51.95
    max: 720.5
    std: 341.59667259325835
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 51.95
      - 85.3775
      - 118.80499999999999
      - 152.2325
      - 185.65999999999997
      - 219.08749999999998
      - 252.515
      - 285.94249999999994
```

(continues on next page)

(continued from previous page)

```
      - 319.36999999999995
      - 352.79749999999996
      - 386.22499999999997
      - 419.6524999999999
      - 453.0799999999999
      - 486.50749999999994
      - 519.935
      - 553.3625
      - 586.79
      - 620.2175
      - 653.645
      - 687.0725
      - 720.5
  bids_max_1h:
    count: 8.0
    mean: 308.42625
    min: 51.95
    max: 720.5
    std: 341.58380276661245
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 51.95
      - 85.3775
      - 118.80499999999999
      - 152.2325
      - 185.65999999999997
      - 219.08749999999998
      - 252.515
      - 285.94249999999994
      - 319.36999999999995
      - 352.79749999999996
      - 386.22499999999997
      - 419.6524999999999
```

(continues on next page)

(continued from previous page)

```
      - 453.0799999999999
      - 486.50749999999994
      - 519.935
      - 553.3625
      - 586.79
      - 620.2175
      - 653.645
      - 687.0725
      - 720.5
  time:
    count: 8
    mean: '2021-05-23 09:04:07.035699200'
    min: '2021-05-23 09:04:07.013574'
    max: '2021-05-23 09:04:07.065574'
  bid:
    count: 8.0
    mean: 308.42625
    min: 51.95
    max: 720.5
    std: 341.58380276661245
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 51.95
      - 85.3775
      - 118.80499999999999
      - 152.2325
      - 185.65999999999997
      - 219.08749999999998
      - 252.515
      - 285.94249999999994
      - 319.36999999999995
      - 352.79749999999996
      - 386.22499999999997
```

(continues on next page)

(continued from previous page)


```
      - 419.6524999999999
      - 453.0799999999999
      - 486.50749999999994
      - 519.935
      - 553.3625
      - 586.79
      - 620.2175
      - 653.645
      - 687.0725
      - 720.5
  ask:
    count: 8.0
    mean: 308.59
    min: 51.96
    max: 720.93
    std: 341.79037903369954
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 51.96
      - 85.4085
      - 118.857
      - 152.3055
      - 185.754
      - 219.2025
      - 252.65099999999998
      - 286.0995
      - 319.54799999999994
      - 352.9964999999999
      - 386.44499999999994
      - 419.89349999999996
      - 453.3419999999999
      - 486.7904999999999
      - 520.2389999999999
```


(continues on next page)

(continued from previous page)

```
      - 553.6875
      - 587.136
      - 620.5844999999999
      - 654.0329999999999
      - 687.4815
      - 720.93
  multi:
    count: 8.0
    mean: 925.27875
    min: 155.85000000000002
    max: 2161.5
    std: 1024.7514082998375
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 155.85000000000002
      - 256.13250000000005
      - 356.415
      - 456.6975
      - 556.98
      - 657.2625
      - 757.545
      - 857.8275
      - 958.11
      - 1058.3925
      - 1158.6750000000002
      - 1258.9575
      - 1359.2399999999998
      - 1459.5225
      - 1559.8049999999998
      - 1660.0875
      - 1760.37
      - 1860.6525000000001
      - 1960.935
```

(continues on next page)

(continued from previous page)

```
      - 2061.2175
      - 2161.5
  extra:
    count: 8.0
    mean: 23748.82125
    min: 4000.15
    max: 55478.5
    std: 26301.95281302916
    hist:
    - - 4
      - 1
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 0
      - 3
    - - 4000.15
      - 6574.0675
      - 9147.985
      - 11721.9025
      - 14295.82
      - 16869.7375
      - 19443.655000000002
      - 22017.572500000002
      - 24591.49
      - 27165.4075
      - 29739.325
      - 32313.2425
      - 34887.16
      - 37461.0775
      - 40034.995
      - 42608.9125
      - 45182.83
      - 47756.747500000005
      - 50330.665
      - 52904.582500000004
      - 55478.5
preview:
- - asks1_sum_1h
```

(continues on next page)

(continued from previous page)

```
    - asks1_max_1h
    - asks5_sum_5h
    - asks5_max_5h
    - bids_min_1h
    - bids_max_1h
    - time
    - bid
    - ask
    - multi
    - extra
  - - 720.93
    - 720.93
    - 720.93
    - 720.93
    - 720.5
    - 720.5
    - 2021-05-23T09:04:07.013574
    - 720.5
    - 720.93
    - 2161.5
    - 55478.5
  - - 51.96
    - 51.96
    - 51.96
    - 51.96
    - 51.95
    - 51.95
    - 2021-05-23T09:04:07.013574
    - 51.95
    - 51.96
    - 155.85000000000002
    - 4000.15
  - - 103.94
    - 51.98
    - 103.94
    - 51.98
    - 51.95
    - 51.97
    - 2021-05-23T09:04:07.020574
    - 51.97
    - 51.98
    - 155.91
    - 4001.69
  - - 155.94
    - 52.0
    - 155.94
    - 52.0
    - 51.95
    - 51.99
    - 2021-05-23T09:04:07.031574
    - 51.99
    - 52.0
```

(continues on next page)

(continued from previous page)


```
  - 155.97
  - 4003.23
- - 1441.86
  - 720.93
  - 1441.86
  - 720.93
  - 720.5
  - 720.5
  - 2021-05-23T09:04:07.038574
  - 720.5
  - 720.93
  - 2161.5
  - 55478.5
- - 98.01
  - 98.01
  - 98.01
  - 98.01
  - 97.99
  - 97.99
  - 2021-05-23T09:04:07.039574
  - 97.99
  - 98.01
  - 293.96999999999997
  - 7545.23
- - 2162.74
  - 720.93
  - 2162.74
  - 720.93
  - 720.5
  - 720.5
  - 2021-05-23T09:04:07.062574
  - 720.5
  - 720.88
  - 2161.5
  - 55478.5
- - 207.97
  - 52.03
  - 207.97
  - 52.03
  - 51.95
  - 52.01
  - 2021-05-23T09:04:07.065574
  - 52.01
  - 52.03
  - 156.03
  - 4004.77
```

Ingest data into offline and online stores

This writes to both targets (Parquet and NoSQL).

```
# save ingest data and print the FeatureSet spec
df = fstore.ingest(quotes_set, quotes)
```

```
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.013574 args={
↪'min': 52, 'value': 51.95}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.020574 args={
↪'min': 52, 'value': 51.97}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.031574 args={
↪'min': 52, 'value': 51.99}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.013574 args={
↪'min': 52, 'value': 51.95}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.020574 args={
↪'min': 52, 'value': 51.97}
info! bid value is smaller than min, key=['MSFT'] time=2021-05-23 09:04:07.031574 args={
↪'min': 52, 'value': 51.99}
```

3.8.4 Get an offline feature vector for training

Example of combining features from 3 sources with time travel join of 3 tables with **time travel**.

Specify a set of features and request the feature vector offline result as a dataframe:

```
features = [
    "stock-quotes.multi",
    "stock-quotes.asks5_sum_5h as total_ask",
    "stock-quotes.bids_min_1h",
    "stock-quotes.bids_max_1h",
    "stocks.*",
]

vector = fstore.FeatureVector("stocks-vec", features, description="stocks demo feature␣
↪vector")
vector.save()
```

```
resp = fstore.get_offline_features(vector, entity_rows=trades, entity_timestamp_column=
↪"time")
resp.to_dataframe()
```

```
    price  quantity    multi  total_ask  bids_min_1h  bids_max_1h  \
0   51.95        75   155.97     155.94        51.95        51.99
1   51.95       155   155.97     155.94        51.95        51.99
2  720.77       100  2161.50    2162.74       720.50       720.50
3  720.92       100  2161.50    2162.74       720.50       720.50
4   98.00       100   293.97      98.01        97.99        97.99

                   name exchange
0  Microsoft Corporation   NASDAQ
1  Microsoft Corporation   NASDAQ
```

(continues on next page)

(continued from previous page)

```
2           Alphabet Inc   NASDAQ
3           Alphabet Inc   NASDAQ
4              Apple Inc   NASDAQ
```

3.8.5 Initialize an online feature service and use it for real-time inference

```
service = fstore.get_online_feature_service("stocks-vec")
```

Request feature vector statistics, can be used for imputing or validation

```
service.vector.get_stats_table()
```

```
             count       mean     min      max          std  \
multi          8.0  925.27875  155.85  2161.50  1024.751408
total_ask      8.0  617.91875   51.96  2162.74   784.877980
bids_min_1h    8.0  308.41125   51.95   720.50   341.596673
bids_max_1h    8.0  308.42625   51.95   720.50   341.583803
name           3.0        NaN     NaN      NaN          NaN
exchange       3.0        NaN     NaN      NaN          NaN

                                                      hist  unique  \
multi        [[4, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...     NaN
total_ask    [[4, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0,...     NaN
bids_min_1h  [[4, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...     NaN
bids_max_1h  [[4, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...     NaN
name                                                       NaN     3.0
exchange                                                   NaN     1.0

                      top  freq
multi                 NaN   NaN
total_ask             NaN   NaN
bids_min_1h           NaN   NaN
bids_max_1h           NaN   NaN
name          Alphabet Inc   1.0
exchange            NASDAQ   3.0
```

Real-time feature vector request

```
service.get([{"ticker": "GOOG"}, {"ticker": "MSFT"}])
```

```
[{'asks5_sum_5h': 2162.74,
  'bids_min_1h': 720.5,
  'bids_max_1h': 720.5,
  'multi': 2161.5,
  'name': 'Alphabet Inc',
  'exchange': 'NASDAQ',
  'total_ask': None},
 {'asks5_sum_5h': 207.97,
  'bids_min_1h': 51.95,
  'bids_max_1h': 52.01,
```

(continues on next page)

(continued from previous page)

```
  'multi': 156.03,
  'name': 'Microsoft Corporation',
  'exchange': 'NASDAQ',
  'total_ask': None}]
```

```
service.get([{"ticker": "AAPL"}])
```

```
[{'asks5_sum_5h': 98.01,
  'bids_min_1h': 97.99,
  'bids_max_1h': 97.99,
  'multi': 293.97,
  'name': 'Apple Inc',
  'exchange': 'NASDAQ',
  'total_ask': None}]
```

```
service.close()
```

3.9 Targeted Tutorials

Each of the following tutorials is a dedicated Jupyter notebook. You can download them by clicking the `download` icon at the top of each page.

Train, compare, and register Models Demo of training ML models, hyper-parameters, track and compare experiments, register and use the models.

Serving pre-trained ML/DL models How to deploy real-time serving pipelines with MLRun Serving and different types of pre-trained ML/DL models.

Projects & automated ML pipeline How to work with projects, source control (git), CI/CD, to easily build and deploy multi-stage ML pipelines.

Real-time monitoring & drift detection Demonstrate MLRun Serving pipelines, MLRun model monitoring, and automated drift detection.

Add MLOps to existing code Turn a Kaggle research notebook to a production ML micro-service with minimal code changes using MLRun.

Basic feature store example (stocks) Understand MLRun feature store with a simple example: build, transform, and serve features in batch and in real-time.

Batch inference and drift detection Use MLRun batch inference function (from MLRun Function Hub), run it as a batch job, and generate drift reports.

Advanced real-time pipeline Demonstrates a multi-step online pipeline with data prep, ensemble, model serving, and post processing.

Feature store end-to-end demo Use the feature store with data ingestion, model training, model serving, and automated pipeline.

3.10 End to End Demos

You can find the different end-to-end demos in the MLRun demos repository: **github.com/mlrun/demos**.

3.11 Running the demos in Open Source MLRun

By default, these demos work with the online feature store, which is currently not part of the Open Source MLRun default deployment:

- fraud-prevention-feature-store
- network-operations
- azureml_demo

CHAPTER

FOUR

INSTALLATION AND SETUP GUIDE

This guide outlines the steps for installing and running MLRun.

MLRun has two main components, the service and the client (SDK):

- MLRun service runs over Kubernetes (can also be deployed using local Docker for demo and test purposes). It can orchestrate and integrate with other open source open source frameworks, as shown in the following diagram.
- MLRun client SDK is installed in your development environment and interacts with the service using REST API calls.

In this section

4.1 Deployment options

There are several deployment options:

- *Local deployment*: Deploy a Docker on your laptop or on a single server. This option is good for testing the waters or when working in a small scale environment. It's limited in terms of computing resources and scale, but simpler for deployment.
- *Kubernetes cluster*: Deploy an MLRun server on Kubernetes. This option deploys MLRun on a Kubernetes cluster, which supports elastic scaling. Yet, it is more complex to install as it requires you to install Kubernetes on your own.
- *Amazon Web Services (AWS)*: Deploy an MLRun server on AWS. This option is the easiest way to install MLRun cluster and use cloud-based services. The MLRun software is free of charge, however, there is a cost for the AWS infrastructure services.
- Iguazio's Managed Service: A commercial offering by Iguazio. This is the fastest way to explore the full set of MLRun functionalities. Note that Iguazio provides a 14 day free trial.

4.2 Set up your client

You can work with your favorite IDE (e.g. Pycharm, VScode, Jupyter, Colab etc...). Read how to configure your client against the deployed MLRun server in *How to configure your client*.

Once you have installed and configured MLRun, follow the Quick Start tutorial and additional *Tutorials and Examples* to learn how to use MLRun to develop and deploy machine learning applications to production.

4.2.1 MLRun client backward compatibility

Starting from MLRun 0.10.0, the MLRun client and images are compatible with minor MLRun releases that are released during the following 6 months. When you upgrade to 0.11.0, for example, you can continue to use your 0.10-based images.

Important

- Images from 0.9.0 are not compatible with 0.10.0. Backward compatibility starts from 0.10.0.
- When you upgrade the MLRun major version, for example 0.10.x to 1.0.x, there is no backward compatibility.
- The feature store is not backward compatible.
- When you upgrade the platform, for example from 3.2 to 3.3, the clients should be upgraded. There is no guaranteed compatibility with an older MLRun client after a platform upgrade.

See also *Images and their usage in MLRun*.

4.3 Security

4.3.1 Non-root user support

By default, MLRun assigns the root user to MLRun runtimes and pods. You can improve the security context by changing the security mode, which is implemented by Iguazio during installation, and applied system-wide:

- Override: Use the user id of the user that triggered the current run or use the nogroupid for group id. Requires Iguazio v3.5.1.
- Disabled: Security context is not auto applied (the system applies the root user). (default)

4.3.2 Security context

If your system is configured in disabled mode, you can apply the security context to individual runtimes/pods by using `function.with_security_context`, and the job is assigned to the user or to the user's group that ran the job. (You cannot override the user of individual jobs if the system is configured in override mode.) The options are:

```
from kubernetes import client as k8s_client

security_context = k8s_client.V1SecurityContext(
            run_as_user=1000,
            run_as_group=3000,
```

(continues on next page)

(continued from previous page)

```
        )
function.with_security_context(security_context)
```

See the full definition of the V1SecurityContext object.

Some services do not support security context yet:

- Infrastructure services
  - Kubeflow pipelines core services
- Services created by MLRun
  - Kaniko, used for building images. (To avoid using Kaniko, use prebuilt images that contain all the requirements.)
  - Spark services

Install MLRun locally using Docker

You can install and use MLRun and Nuclio locally on your computer. This does not include all the services and elastic scaling capabilities, which you can get with the Kubernetes based deployment, but it is much simpler to start with.

Note

Using Docker is limited to local, Nuclio, serving runtimes, and local pipelines.

Prerequisites

- Memory: 8GB
- Storage: 7GB

Overview

Use `docker compose` to install MLRun. It deploys the MLRun service, MLRun UI, Nuclio serverless engine, and optionally the Jupyter server. The MLRun service, MLRun UI, Nuclio, and Jupyter, do not have default resources. This means that they are set with the default cluster/namespace resources limits. These can be modified.

There are two installation options:

- ***Use MLRun with your own client (PyCharm, VSCode, Jupyter)***
- ***Use MLRun with MLRun Jupyter image (pre loaded with examples/demos)***

In both cases you need to set the `SHARED_DIR` environment variable to point to a host path for storing MLRun artifacts and DB, for example `export SHARED_DIR=~/mlrun-data` (or use `set SHARED_DIR=c:\mlrun-data` in windows). Make sure the directory exists.

You also need to set the `HOST_IP` variable with your computer IP address (required for Nuclio dashboard). You can select a specific MLRun version with the `TAG` variable and Nuclio version with the `NUCLIO_TAG` variable.

Add the `-d` flag to `docker-compose` for running in detached mode (in the background).

Note

Support for running as a non-root user was added in 1.0.5, hence the underlying exposed port was changed. If you want to use previous mlrun versions, modify the mlrun-ui port from 8090 back to 80.

If you are running more than one instance of MLRun, change the exposed port.

Watch the installation:

Use MLRun with your own client

The following commands install MLRun and Nuclio for work with your own IDE or notebook.

[Download here] the `compose.yaml` file, save it to the working dir and type:

show the compose.yaml file

```
services:
  init_nuclio:
    image: alpine:3.16
    command:
      - "/bin/sh"
      - "-c"
      - |
        mkdir -p /etc/nuclio/config/platform; \
        cat << EOF | tee /etc/nuclio/config/platform/platform.yaml
        runtime:
          common:
            env:
              MLRUN_DBPATH: http://${HOST_IP:?err}:8080
        local:
          defaultFunctionContainerNetworkName: mlrun
          defaultFunctionRestartPolicy:
            name: always
            maxRetryCount: 0
          defaultFunctionVolumes:
            - volume:
                name: mlrun-stuff
                hostPath:
                  path: ${SHARED_DIR:?err}
              volumeMount:
                name: mlrun-stuff
                mountPath: /home/jovyan/data/
        logger:
          sinks:
            myStdoutLoggerSink:
              kind: stdout
          system:
            - level: debug
              sink: myStdoutLoggerSink
          functions:
            - level: debug
              sink: myStdoutLoggerSink
        EOF
```

(continues on next page)

(continued from previous page)

```
    volumes:
      - nuclio-platform-config:/etc/nuclio/config

  mlrun-api:
    image: "mlrun/mlrun-api:${TAG:-1.2.0}"
    ports:
      - "8080:8080"
    environment:
      MLRUN_ARTIFACT_PATH: "${SHARED_DIR}/{{project}}"
      # using local storage, meaning files / artifacts are stored locally, so we want to␣
↪allow access to them
      MLRUN_HTTPDB__REAL_PATH: /data
      MLRUN_HTTPDB__DATA_VOLUME: "${SHARED_DIR}"
      MLRUN_LOG_LEVEL: DEBUG
      MLRUN_NUCLIO_DASHBOARD_URL: http://nuclio:8070
      MLRUN_HTTPDB__DSN: "sqlite:////data/mlrun.db?check_same_thread=false"
      MLRUN_UI__URL: http://localhost:8060
      # not running on k8s meaning no need to store secrets
      MLRUN_SECRET_STORES__KUBERNETES__AUTO_ADD_PROJECT_SECRETS: "false"
      # let mlrun control nuclio resources
      MLRUN_HTTPDB__PROJECTS__FOLLOWERS: "nuclio"
    volumes:
      - "${SHARED_DIR:?err}:/data"
    networks:
      - mlrun

  mlrun-ui:
    image: "mlrun/mlrun-ui:${TAG:-1.2.0}"
    ports:
      - "8060:8090"
    environment:
      MLRUN_API_PROXY_URL: http://mlrun-api:8080
      MLRUN_NUCLIO_MODE: enable
      MLRUN_NUCLIO_API_URL: http://nuclio:8070
      MLRUN_NUCLIO_UI_URL: http://localhost:8070
    networks:
      - mlrun

  nuclio:
    image: "quay.io/nuclio/dashboard:${NUCLIO_TAG:-stable-amd64}"
    ports:
      - "8070:8070"
    environment:
      NUCLIO_DASHBOARD_EXTERNAL_IP_ADDRESSES: "${HOST_IP:?err}"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - nuclio-platform-config:/etc/nuclio/config
    depends_on:
      - init_nuclio
    networks:
      - mlrun
```

(continues on next page)

(continued from previous page)

```
volumes:
  nuclio-platform-config: {}

networks:
  mlrun:
    name: mlrun
```

Linux/Mac

```
export HOST_IP=<your host IP address>
export SHARED_DIR=~/mlrun-data
mkdir $SHARED_DIR -p
docker-compose -f compose.yaml up
```

Your `HOST_IP` address can be found using the `ip addr` or `ifconfig` commands. It is recommended to select an address that does not change dynamically (for example the IP of the bridge interface).

Windows (cmd)

```
set HOST_IP=<your host IP address>
set SHARED_DIR=c:\mlrun-data
mkdir %SHARED_DIR%
docker-compose -f compose.yaml up
```

Your `HOST_IP` address can be found using the `ipconfig` shell command, it is recommended to select an address that does not change dynamically (for example the IP of the `vEthernet` interface).

Powershell

```
$Env:HOST_IP=<your host IP address>
$Env:SHARED_DIR="~/mlrun-data"
mkdir $Env:SHARED_DIR
docker-compose -f compose.yaml up
```

Your `HOST_IP` address can be found using the `Get-NetIPConfiguration` cmdlet, it is recommended to select an address that does not change dynamically (for example the IP of the `vEthernet` interface).

This creates 3 services:

- MLRun API (in http://localhost:8080)
- MLRun UI (in http://localhost:8060)
- Nuclio Dashboard/controller (in http://localhost:8070)

After installing MLRun service, set your client environment to work with the service, by setting the MLRun path env variable to `MLRUN_DBPATH=http://localhost:8080` or using `.env` files (see setting client environment).

Use MLRun with MLRun Jupyter image

For the quickest experience with MLRun you can deploy MLRun with a pre-integrated Jupyter server loaded with various ready-to-use MLRun examples.

[Download here] the `compose.with-jupyter.yaml` file, save it to the working dir and type:

```
services:
  init_nuclio:
    image: alpine:3.16
    command:
      - "/bin/sh"
      - "-c"
      - |
        mkdir -p /etc/nuclio/config/platform; \
        cat << EOF | tee /etc/nuclio/config/platform/platform.yaml
        runtime:
          common:
            env:
              MLRUN_DBPATH: http://${HOST_IP:?err}:8080
        local:
          defaultFunctionContainerNetworkName: mlrun
          defaultFunctionRestartPolicy:
            name: always
            maxRetryCount: 0
          defaultFunctionVolumes:
            - volume:
                name: mlrun-stuff
                hostPath:
                  path: ${SHARED_DIR:?err}
              volumeMount:
                name: mlrun-stuff
                mountPath: /home/jovyan/data/
        logger:
          sinks:
            myStdoutLoggerSink:
              kind: stdout
          system:
            - level: debug
              sink: myStdoutLoggerSink
          functions:
            - level: debug
              sink: myStdoutLoggerSink
        EOF
    volumes:
      - nuclio-platform-config:/etc/nuclio/config

  jupyter:
    image: "mlrun/jupyter:${TAG:-1.2.0}"
    ports:
      - "8080:8080"
      - "8888:8888"
    environment:
```

(continues on next page)

(continued from previous page)

```
      MLRUN_ARTIFACT_PATH: "/home/jovyan/data/{{project}}"
      MLRUN_LOG_LEVEL: DEBUG
      MLRUN_NUCLIO_DASHBOARD_URL: http://nuclio:8070
      MLRUN_HTTPDB__DSN: "sqlite:////home/jovyan/data/mlrun.db?check_same_thread=false"
      MLRUN_UI__URL: http://localhost:8060
      # using local storage, meaning files / artifacts are stored locally, so we want to␣
↪allow access to them
      MLRUN_HTTPDB__REAL_PATH: "/home/jovyan/data"
      # not running on k8s meaning no need to store secrets
      MLRUN_SECRET_STORES__KUBERNETES__AUTO_ADD_PROJECT_SECRETS: "false"
      # let mlrun control nuclio resources
      MLRUN_HTTPDB__PROJECTS__FOLLOWERS: "nuclio"
    volumes:
      - "${SHARED_DIR:?err}:/home/jovyan/data"
    networks:
      - mlrun

  mlrun-ui:
    image: "mlrun/mlrun-ui:${TAG:-1.2.0}"
    ports:
      - "8060:8090"
    environment:
      MLRUN_API_PROXY_URL: http://jupyter:8080
      MLRUN_NUCLIO_MODE: enable
      MLRUN_NUCLIO_API_URL: http://nuclio:8070
      MLRUN_NUCLIO_UI_URL: http://localhost:8070
    networks:
      - mlrun

  nuclio:
    image: "quay.io/nuclio/dashboard:${NUCLIO_TAG:-stable-amd64}"
    ports:
      - "8070:8070"
    environment:
      NUCLIO_DASHBOARD_EXTERNAL_IP_ADDRESSES: "${HOST_IP:?err}"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - nuclio-platform-config:/etc/nuclio/config
    depends_on:
      - init_nuclio
    networks:
      - mlrun

volumes:
  nuclio-platform-config: {}

networks:
  mlrun:
    name: mlrun
```

Linux/Mac

```
export HOST_IP=<your host IP address>
export SHARED_DIR=~/mlrun-data
mkdir -p $SHARED_DIR
docker-compose -f compose.with-jupyter.yaml up
```

Your `HOST_IP` address can be found using the `ip addr` or `ifconfig` commands. It is recommended to select an address that does not change dynamically (for example the IP of the bridge interface).

Windows (cmd)

```
set HOST_IP=<your host IP address>
set SHARED_DIR=c:\mlrun-data
mkdir %SHARED_DIR%
docker-compose -f compose.with-jupyter.yaml up
```

Your `HOST_IP` address can be found using the `ipconfig` shell command, it is recommended to select an address that does not change dynamically (for example the IP of the `vEthernet` interface).

Powershell

```
$Env:HOST_IP=<your host IP address>
$Env:SHARED_DIR="~/mlrun-data"
mkdir $Env:SHARED_DIR
docker-compose -f compose.with-jupyter.yaml up
```

Your `HOST_IP` address can be found using the `Get-NetIPConfiguration` cmdlet, it is recommended to select an address that does not change dynamically (for example the IP of the `vEthernet` interface).

This creates 4 services:

- Jupyter lab (in http://localhost:8888)
- MLRun API (in http://localhost:8080), running on the Jupyter container
- MLRun UI (in http://localhost:8060)
- Nuclio Dashboard/controller (in http://localhost:8070)

After the installation, access the Jupyter server (in http://localhost:8888) and run through the quick-start tutorial and `demos`. You can see the projects, tasks, and artifacts in MLRun UI (in http://localhost:8060).

The Jupyter environment is pre-configured to work with the local MLRun and Nuclio services. You can switch to a remote or managed MLRun cluster by editing the `mlrun.env` file in the Jupyter files tree.

The artifacts and DB are stored under **/home/jovyan/data** (`/data` in Jupyter tree).

Install MLRun on Kubernetes

In this section

Prerequisites

- Access to a Kubernetes cluster. You must have administrator permissions in order to install MLRun on your cluster. For local installation on Windows or Mac, Docker Desktop is recommended. MLRun fully supports k8s releases 1.22 and 1.23.
- The Kubernetes command-line tool (kubectl) compatible with your Kubernetes cluster is installed. Refer to the kubectl installation instructions for more information.
- Helm 3.6 CLI is installed. Refer to the Helm installation instructions for more information.
- An accessible docker-registry (such as Docker Hub). The registry's URL and credentials are consumed by the applications via a pre-created secret.
- Storage: 7Gi
- RAM: A minimum of 8Gi is required for running all the initial MLRun components. The amount of RAM required for running MLRun jobs depends on the job's requirements.

Note

The MLRun Community Edition resources are configured initially with the default cluster/namespace resources limits. You can modify the resources from outside if needed.

Community Edition flavors

The MLRun CE (Community Edition) chart arrives in 2 flavors - lite and full. The lite version is the default installation and includes the following components:

- MLRun - https://github.com/mlrun/mlrun
  - MLRun API
  - MLRun UI
  - MLRun DB (MySQL)
- Nuclio - https://github.com/nuclio/nuclio
- Jupyter - https://github.com/jupyter/notebook (+MLRun integrated)
- MPI Operator - https://github.com/kubeflow/mpi-operator
- Minio - https://github.com/minio/minio/tree/master/helm/minio

The Full Version also includes:

- Spark Operator - https://github.com/GoogleCloudPlatform/spark-on-k8s-operator
- Pipelines - https://github.com/kubeflow/pipelines
- Prometheus stack - https://github.com/prometheus-community/helm-charts
  - Prometheus
  - Grafana

Installing on Docker Desktop

Docker Desktop is available for Mac and Windows. For download information, system requirements, and installation instructions, see:

- Install Docker Desktop on Mac
- Install Docker Desktop on Windows. Note that WSL 2 backend was tested, Hyper-V was not tested.

Configuring Docker Desktop

Docker Desktop includes a standalone Kubernetes server and client, as well as Docker CLI integration that runs on your machine. The Kubernetes server runs locally within your Docker instance. To enable Kubernetes support and install a standalone instance of Kubernetes running as a Docker container, go to **Preferences > Kubernetes** and then click **Enable Kubernetes**. Click **Apply & Restart** to save the settings and then click **Install** to confirm. This instantiates the images that are required to run the Kubernetes server as containers, and installs the `/usr/local/bin/kubectl` command on your machine. For more information, see the Kubernetes documentation.

It's recommended to limit the amount of memory allocated to Kubernetes. If you're using Windows and WSL 2, you can configure global WSL options by placing a `.wslconfig` file into the root directory of your users folder: `C:\Users\<yourUserName>\.wslconfig`. Keep in mind that you might need to run `wsl --shutdown` to shut down the WSL 2 VM and then restart your WSL instance for these changes to take effect.

```
[wsl2]
memory=8GB # Limits VM memory in WSL 2 to 8 GB
```

To learn about the various UI options and their usage, see:

- Docker Desktop for Mac user manual
- Docker Desktop for Windows user manual

Installing the Lite Version

Note

These instructions use `mlrun` as the namespace (`-n` parameter). You can choose a different namespace in your kubernetes cluster.

Create a namespace for the deployed components:

```
kubectl create namespace mlrun
```

Add the Community Edition helm chart repo:

```
helm repo add mlrun-ce https://mlrun.github.io/ce
```

Update the repo to make sure you're getting the latest chart:

```
helm repo update
```

Create a secret with your docker-registry named `registry-credentials`:

```
kubectl --namespace mlrun create secret docker-registry registry-credentials \
    --docker-server <your-registry-server> \
    --docker-username <your-username> \
    --docker-password <your-password> \
    --docker-email <your-email>
```

Where:

- <your-registry-server> is your Private Docker Registry FQDN. (https://index.docker.io/v1/ for Docker Hub).
- <your-username> is your Docker username.
- <your-password> is your Docker password.
- <your-email> is your Docker email.

Note

First-time MLRun users will experience a relatively longer installation time because all required images are being pulled locally for the first time (it will take an average of 10-15 minutes mostly depends on your internet speed).

To install the chart with the release name `mlrun-ce` use the following command. Note the reference to the pre-created `registry-credentials` secret in `global.registry.secretName`:

```
helm --namespace mlrun \
    install mlrun-ce \
    --wait \
```

(continues on next page)

(continued from previous page)

```
    --timeout 960s \
    --set global.registry.url=<registry-url> \
    --set global.registry.secretName=registry-credentials \
    --set global.externalHostAddress=<host-machine-address> \
    mlrun-ce/mlrun-ce
```

Where:

- <registry-url> is the registry URL that can be authenticated by the registry-credentials secret (e.g., index.docker.io/<your-username> for Docker Hub).
- <host-machine-address> is the IP address of the host machine (or $(minikube ip) if using minikube).

When the installation is complete, the helm command prints the URLs and Ports of all the MLRun CE services.

Installing the Full Version

To install the full version of the chart, use the following command:

```
helm --namespace mlrun \
    install mlrun-ce \
    --wait \
    --timeout 960s \
    -f override-full.yaml \
    --set global.registry.url=<registry-url> \
    --set global.registry.secretName=registry-credentials \
    --set global.externalHostAddress=<host-machine-address> \
    mlrun-ce/mlrun-ce
```

Configuring Online Feature Store

The MLRun Community Edition now supports the online feature store. To enable it, you need to first deploy a REDIS service that is accessible to your MLRun CE cluster. To deploy a REDIS service, refer to the following link.

When you have a REDIS service deployed, you can configure MLRun CE to use it by adding the following helm value configuration to your helm install command:

```
--set mlrun.api.extraEnvKeyValue.MLRUN_REDIS__URL=<redis-address>
```

Usage

Your applications are now available in your local browser:

- jupyter-notebook - http://:30040
- nuclio - http://:30050
- mlrun UI - http://:30060
- mlrun API (external) - http://:30070
- minio API - http://:30080
- minio UI - http://:30090

- pipeline UI - http://:30100
- grafana UI - http://:30110

Check state

You can check current state of installation via command `kubectl -n mlrun get pods`, where the main information is in columns `Ready` and `State`. If all images have already been pulled locally, typically it will take a minute for all services to start.

Note:

- You can change the ports by providing values to the helm install command.
- You can add and configure a k8s ingress-controller for better security and control over external access.

Start Working

Open the Jupyter notebook on **jupyter-notebook UI** and run the code in the **examples/mlrun_basics.ipynb** notebook.

Important

Make sure to save your changes in the `data` folder within the Jupyter Lab. The root folder and any other folder do not retain the changes when you restart the Jupyter Lab.

Configuring the remote environment

You can use your code on a local machine while running your functions on a remote cluster. Refer to Set up your client environment for more information.

Advanced chart configuration

Configurable values are documented in the `values.yaml`, and the `values.yaml` of all sub charts. Override those in the normal methods.

Uninstalling the Chart

```
helm --namespace mlrun uninstall mlrun-ce
```

Note on terminating pods and hanging resources

This chart generates several persistent volume claims that provide persistency (via PVC) out of the box. Upon uninstallation, any hanging / terminating pods will hold the PVCs and PVs respectively, as those prevent their safe removal. Since pods that are stuck in terminating state seem to be a never-ending plague in k8s, note this, and remember to clean the remaining PVs and PVCs.

Handing stuck-at-terminating pods:

```
kubectl --namespace mlrun delete pod --force --grace-period=0 <pod-name>
```

Reclaim dangling persistency resources:

WARNING

This will result in data loss!

```
# To list PVCs
$ kubectl --namespace mlrun get pvc
...

# To remove a PVC
$ kubectl --namespace mlrun delete pvc <pvc-name>
...

# To list PVs
$ kubectl --namespace mlrun get pv
...

# To remove a PVC
$ kubectl --namespace mlrun delete pvc <pv-name>
...
```

Upgrading the chart

In order to upgrade to the latest version of the chart, first make sure you have the latest helm repo

```
helm repo update
```

Then upgrade the chart:

```
helm upgrade --install --reuse-values mlrun-ce mlrun-ce/mlrun-ce
```

Install MLRun on AWS

For AWS users, the easiest way to install MLRun is to use a native AWS deployment. This option deploys MLRun on an AWS EKS service using a CloudFormation stack.

Prerequisites

AWS account with permissions that include the ability to:

- Run a CloudFormation stack
- Create an EKS cluster
- Create EC2 instances
- Create VPC
- Create S3 buckets
- Deploy and pull images from ECR.

For the full set of required permissions, **download the IAM policy** or expand & copy the IAM policy below:

show the IAM policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "BasicServices",
            "Effect": "Allow",
            "Action": [
                "autoscaling:*",
                "cloudwatch:*",
                "elasticloadbalancing:*",
                "sns:*",
                "ec2:*",
                "s3:*",
                "s3-object-lambda:*",
                "eks:*",
                "elasticfilesystem:*",
                "cloudformation:*",
                "acm:*",
                "route53:*"
            ],
            "Resource": "*"
        },
        {
            "Sid": "ServiceLinkedRoles",
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": [
```

(continues on next page)

(continued from previous page)

```
                    "autoscaling.amazonaws.com",
                    "ec2scheduled.amazonaws.com",
                    "elasticloadbalancing.amazonaws.com",
                    "spot.amazonaws.com",
                    "spotfleet.amazonaws.com",
                    "transitgateway.amazonaws.com"
                ]
            }
        }
    },
    {
        "Sid": "IAMPermissions",
        "Effect": "Allow",
        "Action": [
            "iam:AddRoleToInstanceProfile",
            "iam:AttachRolePolicy",
            "iam:TagOpenIDConnectProvider",
            "iam:CreateInstanceProfile",
            "iam:CreateOpenIDConnectProvider",
            "iam:CreateRole",
            "iam:CreateServiceLinkedRole",
            "iam:DeleteInstanceProfile",
            "iam:DeleteOpenIDConnectProvider",
            "iam:DeleteRole",
            "iam:DeleteRolePolicy",
            "iam:DetachRolePolicy",
            "iam:GenerateServiceLastAccessedDetails",
            "iam:GetAccessKeyLastUsed",
            "iam:GetAccountPasswordPolicy",
            "iam:GetAccountSummary",
            "iam:GetGroup",
            "iam:GetInstanceProfile",
            "iam:GetLoginProfile",
            "iam:GetOpenIDConnectProvider",
            "iam:GetPolicy",
            "iam:GetPolicyVersion",
            "iam:GetRole",
            "iam:GetRolePolicy",
            "iam:GetServiceLastAccessedDetails",
            "iam:GetUser",
            "iam:ListAccessKeys",
            "iam:ListAccountAliases",
            "iam:ListAttachedGroupPolicies",
            "iam:ListAttachedRolePolicies",
            "iam:ListAttachedUserPolicies",
            "iam:ListGroupPolicies",
            "iam:ListGroups",
            "iam:ListGroupsForUser",
            "iam:ListInstanceProfilesForRole",
            "iam:ListMFADevices",
            "iam:ListOpenIDConnectProviders",
            "iam:ListPolicies",
```

(continues on next page)

(continued from previous page)

```
                "iam:ListPoliciesGrantingServiceAccess",
                "iam:ListRolePolicies",
                "iam:ListRoles",
                "iam:ListRoleTags",
                "iam:ListSAMLProviders",
                "iam:ListSigningCertificates",
                "iam:ListUserPolicies",
                "iam:ListUsers",
                "iam:ListUserTags",
                "iam:PassRole",
                "iam:PutRolePolicy",
                "iam:RemoveRoleFromInstanceProfile",
                "kms:CreateGrant",
                "kms:CreateKey",
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:Encrypt",
                "kms:GenerateDataKeyWithoutPlaintext",
                "kms:GetKeyPolicy",
                "kms:GetKeyRotationStatus",
                "kms:ListResourceTags",
                "kms:PutKeyPolicy",
                "kms:ScheduleKeyDeletion",
                "kms:TagResource"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowLanbda",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateAlias",
                "lambda:CreateCodeSigningConfig",
                "lambda:CreateEventSourceMapping",
                "lambda:CreateFunction",
                "lambda:CreateFunctionUrlConfig",
                "lambda:Delete*",
                "lambda:Get*",
                "lambda:InvokeAsync",
                "lambda:InvokeFunction",
                "lambda:InvokeFunctionUrl",
                "lambda:List*",
                "lambda:PublishLayerVersion",
                "lambda:PublishVersion",
                "lambda:PutFunctionCodeSigningConfig",
                "lambda:PutFunctionConcurrency",
                "lambda:PutFunctionEventInvokeConfig",
                "lambda:PutProvisionedConcurrencyConfig",
                "lambda:TagResource",
                "lambda:UntagResource",
                "lambda:UpdateAlias",
                "lambda:UpdateCodeSigningConfig",
```

(continues on next page)

(continued from previous page)

```
                "lambda:UpdateEventSourceMapping",
                "lambda:UpdateFunctionCode",
                "lambda:UpdateFunctionCodeSigningConfig",
                "lambda:UpdateFunctionConfiguration",
                "lambda:UpdateFunctionEventInvokeConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "*"
        },
        {
            "Sid": "CertificateService",
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-service-role/acm.amazonaws.com/
↪AWSServiceRoleForCertificateManager*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "acm.amazonaws.com"
                }
            }
        },
        {
            "Sid": "DeleteRole",
            "Effect": "Allow",
            "Action": [
                "iam:DeleteServiceLinkedRole",
                "iam:GetServiceLinkedRoleDeletionStatus",
                "iam:GetRole"
            ],
            "Resource": "arn:aws:iam::*:role/aws-service-role/acm.amazonaws.com/
↪AWSServiceRoleForCertificateManager*"
        },
        {
            "Sid": "SSM",
            "Effect": "Allow",
            "Action": [
                "logs:*",
                "ssm:AddTagsToResource",
                "ssm:GetParameter",
                "ssm:DeleteParameter",
                "ssm:PutParameter",
                "cloudtrail:GetTrail",
                "cloudtrail:ListTrails"
            ],
            "Resource": "*"
        }
    ]
}
```

For more information, see how to create a new AWS account and policies and permissions in IAM.

Notes:

- To access the instances, you will need to have at least one key pair for SSH keys. For more information

see Amazon EC2 key pairs and Linux instances.

- You need to have a Route53 domain configured. External domain registration is currently not supported. For more information see What is Amazon Route 53.
- The MLRun software is free of charge, however, there is a cost for the AWS infrastructure services such as EKS, EC2, S3 and ECR. The actual pricing depends on a large set of factors including, for example, the region, the number of EC2 instances, the amount of storage consumed, and the data transfer costs. Other factors include, for example, reserved instance configuration, saving plan, and AWS credits you have associated with your account. It is recommended to use the AWS pricing calculator to calculate the expected cost, as well as the AWS Cost Explorer to manage the cost, monitor and set-up alerts.

Post deployment expectations

The key components deployed on your EKS cluster are:

- MLRun server (including the feature store and the MLRun graph)
- MLRun UI
- Kubeflow pipeline
- Real time serverless framework (Nuclio)
- Spark operator
- Jupyter lab
- Grafana

Configuration settings

Make sure you are logged in to the correct AWS account.

Click the AWS icon to deploy MLRun.

After clicking the icon, the browser directs you to the CloudFormation stack page in your AWS account, or redirects you to the AWS login page if you are not currently logged in.

> **Note**: You must fill in fields marked as mandatory (m) for the configuration to complete. Fields marked as optional (o) can be left blank.

1. **Stack name** (m)—the name of the stack. You cannot continue if left blank. This field becomes the logical id of the stack. Stack name can include letters (A-Z and a-z), numbers (0-9), and dashes (-).

Parameters

2. **EKS cluster name** (m)—the name of EKS cluster created. The EKS cluster is used to run the MLRun services. For example, `https://jupyter.<eks_cluster_name>.<route53_domain_name>`

VPC network Configuration

3. **Number of Availability Zones** (m)—number of availability zones. The default is set to 3. Choose from the dropdown to change the number. The minimum is 2.
4. **Availability zones** (m)—select a zone from the dropdown. The list is based on the region of the instance. The number of zones must match the number of zones Number of Availability Zones.

5. **Allowed external access CIDR** (m)—range of IP address allowed to access the cluster. Addresses that are not in this range are will not be able to access the cluster.

Amazon EKS configuration

6. **Additional EKS admin ARN (IAM user)** (o)—add an additional admin user to the instance. Users can be added after the stack has been created. For more information see Create a kubeconfig for Amazon EKS.
7. **Instance type** (m)—select from the dropdown list. The default is m5.2xlarge. For size considerations see Amazon EC2 Instance Types.
8. **Number of Nodes** (m)—number of nodes in the cluster. The minimum must match the number of **Availability Zones**. The number of nodes combined with the **Instance type** determines the AWS infrastructure cost.

Amazon EC2 configuration

9. **SSH key name** (m)—select from the stored keys in the dropdown. The list is based on the SSH keys that are in your account. For more information about SSH Keys see Amazon EC2 key pairs and Linux instances.
10. **Provision bastion host** (m)—create a bastion host for SSH access to the Kubernetes nodes. The default is enabled.

Iguazio MLRun configuration

11. **Route 53 hosted DNS domain** (m)—enter the name of your registered Route53 domain. **Note:** Only route53 domains are acceptable.
12. **The URL of your REDIS database** (o)—the URL of your Redis database. This is only required if you're using Redis with the online feature store. See *how to configure the online feature store* for more details.

Other parameters

13. **MLrunCeVersion** (m)—the MLRun Community Edition version to install. Leave the default value for the latest CE release.

Capabilities

14. Check all the capabilities boxes (m).

Press **Create Stack** to continue the deployment. The stack creates a VPC with an EKS cluster and deploys all the services on top of it.

> **Note**: It could take up to 2 hours for your stack to be created.

Getting started

When the stack is complete, go to the **output** tab for the stack you created. There are links for the MLRun UI, Jupyter and the Kubeconfig command.

It's recommended to go through the quick-start and the other tutorials as shown in the documentation. These tutorials and demos come built-in with Jupyter under the root folder of Jupyter.

Storage resources

When installing the MLRun Community Edition via Cloud Formation, several storage resources are created:

- **PVs via AWS storage provider**: Used to hold the file system of the stacks pods, including the MySQL database of MLRun. These are deleted when the stack is uninstalled.
- **S3 Bucket**: A bucket named <EKS cluster name>-<Random string> is created in the AWS account that installs the stack (where <EKS cluster name> is the name of the EKS cluster you chose and <Random string> is part of the CloudFormation stack ID). You can see the bucket name in the output tab of the stack. The bucket is used for MLRun's artifact storage, and is not deleted when uninstalling the stack. The user must empty the bucket and delete it.
- **Container Images in ECR**: When building and deploying MLRun and Nuclio functions via the MLRun Community Edition, the function images are stored in an ECR belonging to the AWS account that installs the stack. These images persist in the account's ECR and are not deleted either.

How to configure the online feature store

The feature store can store data on a fast key value database table for quick serving. This online feature store capability requires an external key value database.

Currently the MLRun feature store supports the following options:

- Redis
- Iguazio key value database

To use Redis, you must install Redis separately and provide the Redis URL when configuring the AWS CloudFormation stack. Refer to the Redis getting-started page for information about Redis installation.

Streaming support

For online serving, it is often convenient to use MLRun graph with a streaming engine. This allows managing queues between steps and functions. MLRun supports Kafka streams as well as Iguazio V3IO streams. See the examples on how to configure the MLRun serving graph with *kafka* and *V3IO*.

Cleanup

To free up the resources used by MLRun:

- Delete the stack. See instructions for deleting a stack on the AWS CloudFormation console for more details.
- Delete the S3 bucket that begins with the same name as your EKS cluster. The S3 bucket name is available in the CloudFormation stack output tab.
- Delete any remaining images in ECR.

You may also need to check any external storage that you used.

Set up your client environment

You can write your code on a local machine while running your functions on a remote cluster. This tutorial explains how to set this up.

In this section

Prerequisites

Before you begin, ensure that the following prerequisites are met:

1. Applications:
   - Python 3.7
   - Recommended pip 22.x+
2. Install MLRun locally.

   You need to install MLRun locally. Make sure the that the MLRun version you install is the same as the MLRun service version. Install a specific version using the following command; replace the `<version>` placeholder with the MLRun version number (e.g., `1.0.0`):

   ```
   pip install mlrun==<version>
   ```

   There are a two `pip install` options:

   - To install the requirements in the requirements.txt, run: `pip install mlrun`
   - If you expect to connect to, or work with, cloud providers (Azure/Google Cloud/S3), you can install additional packages. This is not part of the regular requirements since not all users work with those platforms. Using this option reduces the dependencies and the size of the installation. The additional packages include:
     - pip install mlrun[s3] # Install requirements for S3
     - pip install mlrun[azure-blob-storage] # install requirements for Azure blob storage
     - pip install mlrun[google-cloud-storage] # install requirements for Google cloud storage

     See the full list here. To install all extras, run: `pip install mlrun[complete]`
3. Alternatively, if you already installed a previous version of MLRun, upgrade it by running:

```
pip install -U mlrun==<version>
```

4. Ensure that you have remote access to your MLRun service (i.e., to the service URL on the remote Kubernetes cluster).

Configure remote environment

Set environment variables

Set environment variables to define your MLRun configuration. As a minimum requirement:

1. Set `MLRUN_DBPATH` to the URL of the remote MLRun database/API service:

```
MLRUN_DBPATH=<URL endpoint of the MLRun APIs service endpoint; e.g., "https://mlrun-
↪api.default-tenant.app.mycluster.iguazio.com">
```

2. If the remote service is on an instance of the Iguazio MLOps Platform (**"the platform"**), set the following environment variables as well:

```
V3IO_USERNAME=<username of a platform user with access to the MLRun service>
V3IO_ACCESS_KEY=<platform access key>
```

   You can get the platform access key from the platform dashboard: select the user-profile picture or icon from the top right corner of any page, and select **Access Keys** from the menu. In the **Access Keys** window, either copy an existing access key or create a new key and copy it. Alternatively, you can get the access key by checking the value of the `V3IO_ACCESS_KEY` environment variable in a web-shell or Jupyter Notebook service.

You can also set the environment using MLRun SDK, for example:

```
# Use local service
mlrun.set_environment("http://localhost:8080", artifact_path="./")
# Use remote service
mlrun.set_environment("<remote-service-url>", access_key="xyz", username="joe")
```

Load the configuration and credential environmental variables from file

You can load the env via config file when working from remote (e.g. via PyCharm).

Example env file:

```
# this is an env file
V3IO_USERNAME=admin
V3IO_ACCESS_KEY=MYKEY123
MLRUN_DBPATH=https://mlrun-api.default-tenant.app.xxx.iguazio-cd1.com
AWS_ACCESS_KEY_ID=XXXX
AWS_SECRET_ACCESS_KEY=YYYY
```

Usage:

- `set_env_from_file()` for reading `.env` files, setting the OS environment and reloading MLRun config
- `project.set_secrets()` reads dict or secrets env file and stores it in the project secrets (note that MLRUN_DBPATH and V3IO_xxx vars are not written to the project secrets)

- `function.set_envs()` set the pod environment variables from key/value dict or .env file

---

Note

The V3IO API is determined automatically. If you want to connect to a different V3IO service, set the service in the variable,br. `V3IO_API=<API endpoint of the webapi service endpoint; e.g., "https://default-tenant.app.mycluster.iguazio.com:8444">`

```
# set the env vars from a file and also return the results as a dict (e.g. for using in a function)
env_dict = mlrun.set_env_from_file(env_path, return_dict=True)

# read env vars from dict or file and set as project secrets (plus set the local env)
project.set_secrets({"SECRET1": "value"})
project.set_secrets(file_path=env_file)

# copy env from file into a function spec
function.set_envs(file_path=env_file)
```

Load the configuration and credential environmental variables from the command line

1. Create an env file similar to the example, with lines in the form KEY=VALUE, and comment lines starting with "#".
2. Use `--env-file <env file path>` in mlrun run/build/deploy/project CLI commands to load the config and credential env vars from file.
3. Set the `MLRUN_ENV_FILE=<env file path>` env var to point to a default env file (which will be loaded on import). If the `MLRUN_DBPATH` points to a remote iguazio cluster and the `V3IO_API` and/or `V3IO_FRAMESD` vars are not set, they will be inferred from the DBPATH.
4. Add the default `env` file template in the Jupyter container `~/env` (to allow quick setup of remote demos).

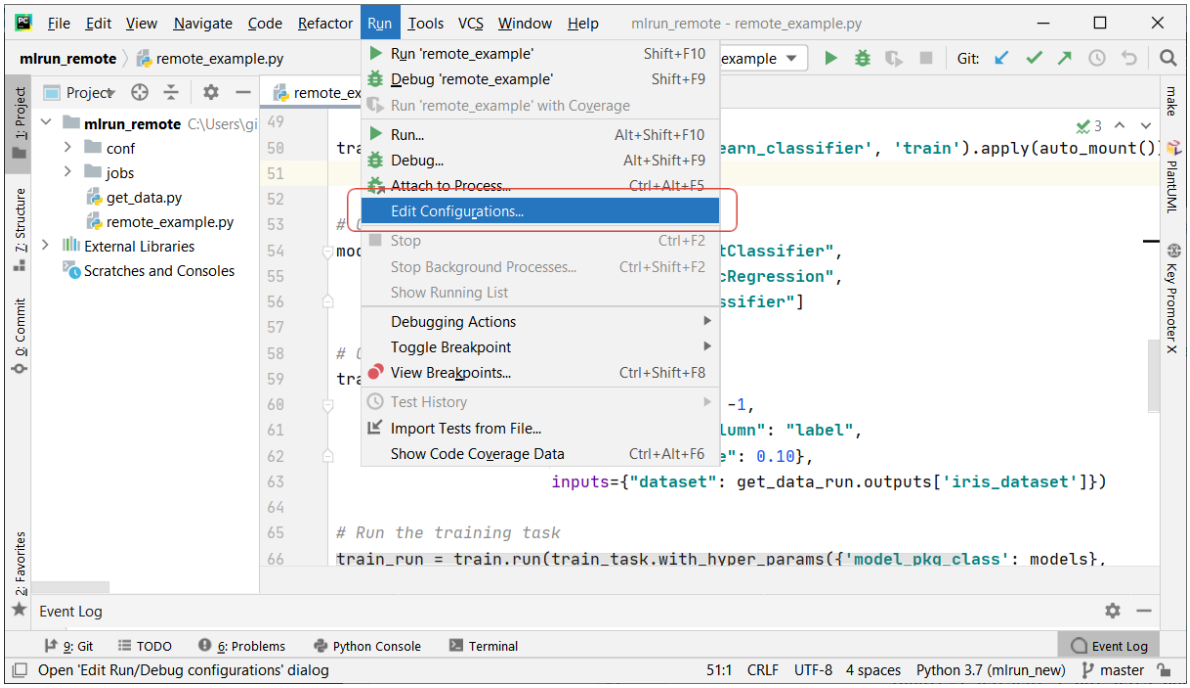
IDE configuration

Use these procedures to access MLRun remotely from your IDE (PyCharm or VSCode).

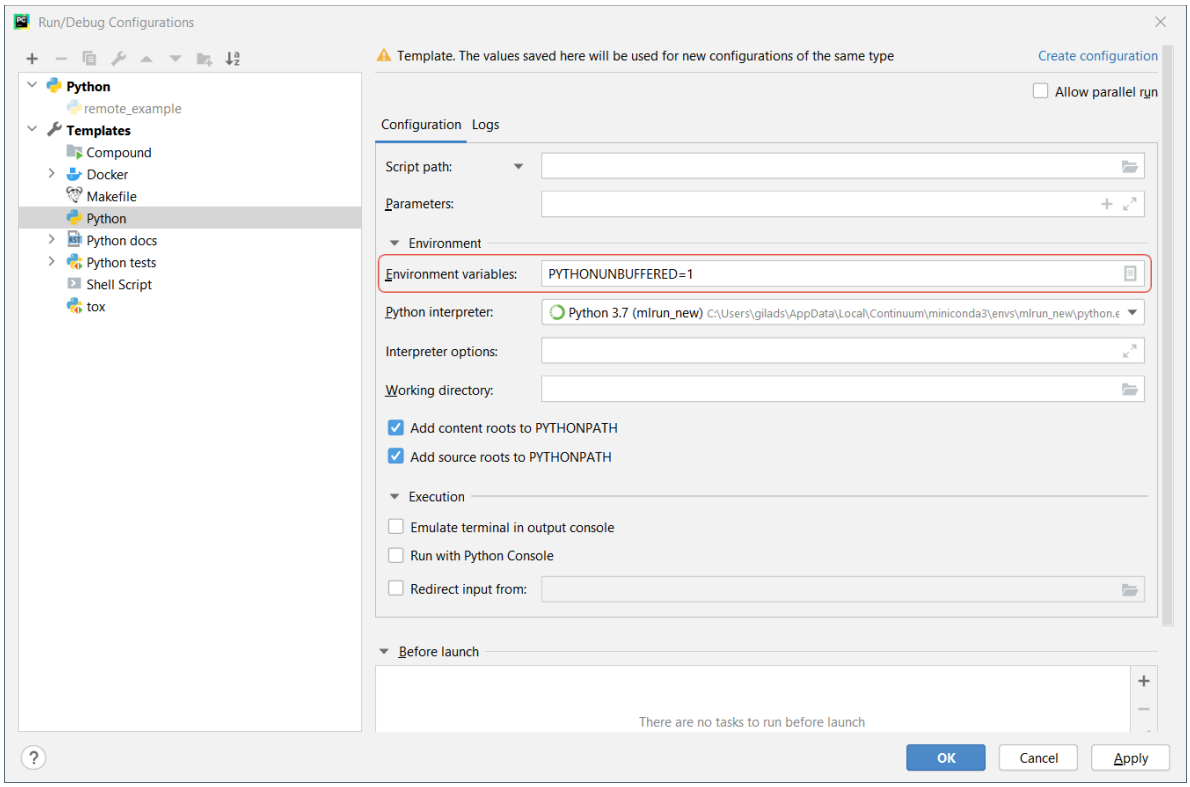
Remote environment from PyCharm

You can use PyCharm with MLRun remote by changing the environment variables configuration.

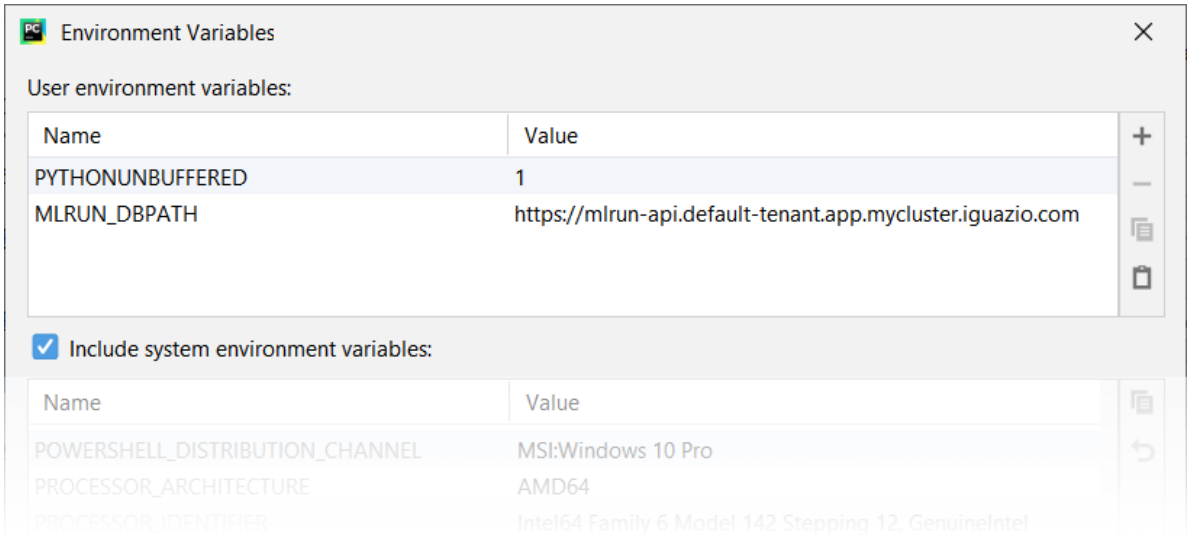
1. From the main menu, choose **Run | Edit Configurations**.



2. To set-up default values for all Python configurations, on the left-hand pane of the run/debug configuration dialog, expand the **Templates** node and select the **Python** node. The corresponding configuration template appears in the right-hand pane. Alternatively, you can edit a specific file configuration by choosing the corresponding file on the left-hand pane. Choose the **Environment Variables** edit box and expand it to edit the environment variables.



3. Add the environment variable and value of `MLRUN_DBPATH`.



If the remote service is on an instance of the Iguazio MLOps Platform, also set the environment variables and values of `V3IO_USERNAME`, and `V3IO_ACCESS_KEY`.

Remote environment from VSCode

Create environment file

Create an environment file called `mlrun.env` in your workspace folder. Copy-paste the configuration below:

```
# Remote URL to mlrun service
MLRUN_DBPATH=<API endpoint of the MLRun APIs service endpoint; e.g., "https://mlrun-api.
↪default-tenant.app.mycluster.iguazio.com">
# Iguazio platform username
V3IO_USERNAME=<username of a platform user with access to the MLRun service>
# Iguazio V3IO data layer credentials (copy from your user settings)
V3IO_ACCESS_KEY=<platform access key>
```

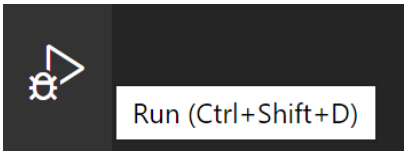
Note

Make sure that you add `.env` to your `.gitignore` file. The environment file contains sensitive information that you should not store in your source control.

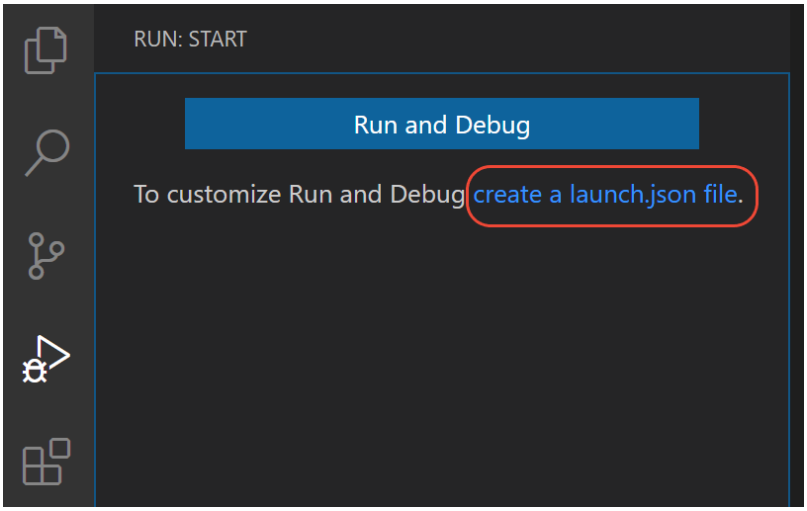
Create Python debug configuration

Create a debug configuration in VSCode. Configurations are defined in a `launch.json` file that's stored in a `.vscode` folder in your workspace.

To initialize debug configurations, first select the Run view in the sidebar:

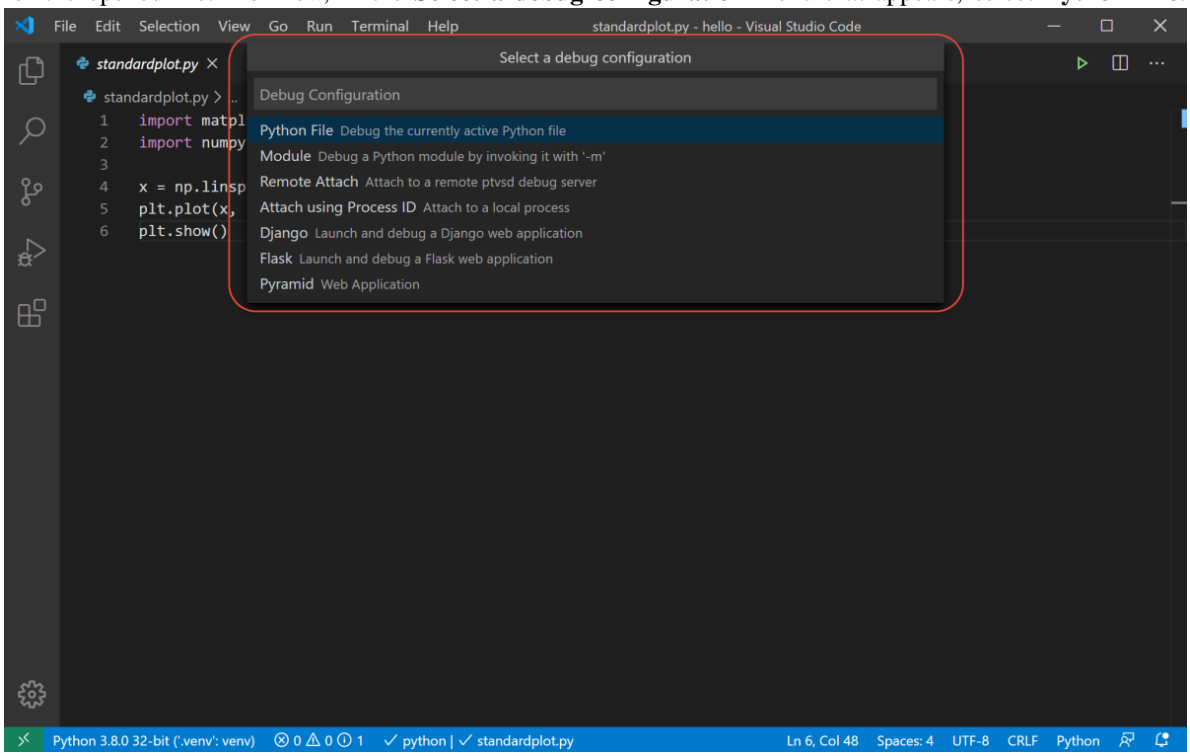


If you don't yet have any configurations defined, you'll see a button to Run and Debug, as well as a link to create a configuration (launch.json) file:



To generate a `launch.json` file with Python configurations:

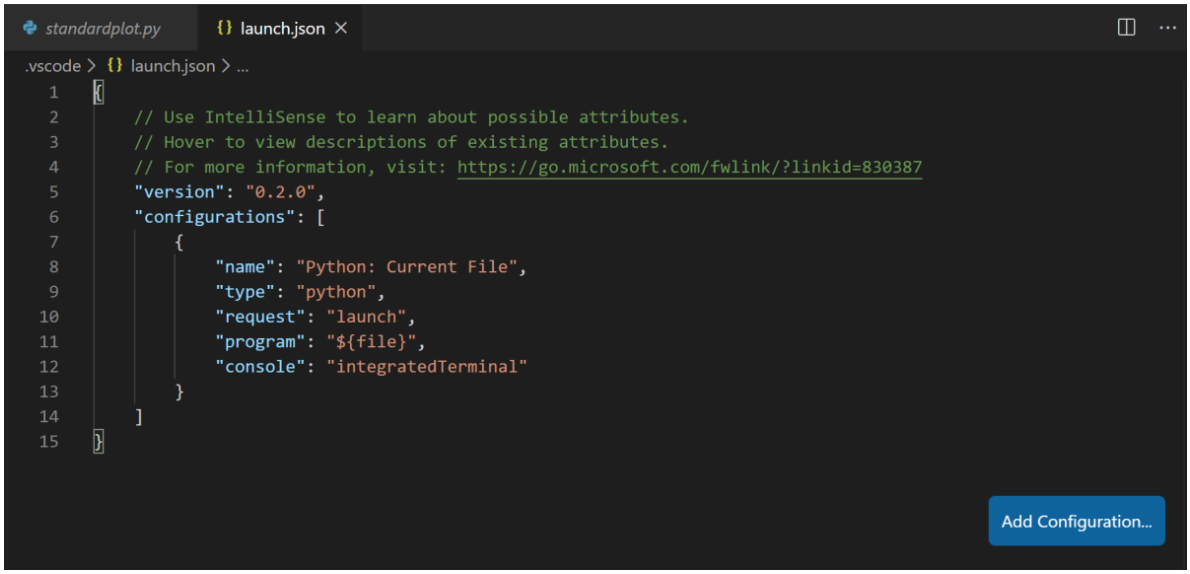
1. Click the **create a launch.json file** link (circled in the image above) or use the **Run > Open configurations** menu command.
2. A configuration menu opens from the Command Palette. Select the type of debug configuration you want for the opened file. For now, in the **Select a debug configuration** menu that appears, select **Python File**.



Note

Starting a debugging session through the Debug Panel, **F5** or **Run > Start Debugging**, when no configuration exists also brings up the debug configuration menu, but does not create a launch.json file.

3. The Python extension then creates and opens a `launch.json` file that contains a pre-defined configuration based on what you previously selected, in this case **Python File**. You can modify configurations (to add arguments, for example), and also add custom configurations.



Set environment file in debug configuration

Add an `envFile` setting to your configuration with the value of `${workspaceFolder}/mlrun.env`

If you created a new configuration in the previous step, your `launch.json` would look as follows:

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python: Current File",
            "type": "python",
            "request": "launch",
            "program": "${file}",
            "console": "integratedTerminal",
            "envFile": "${workspaceFolder}/mlrun.env"
        }
    ]
}
```

CHAPTER

FIVE

PROJECTS AND AUTOMATION

MLRun **Project** is a container for all your work on a particular ML application. Projects host functions, workflows, artifacts (datasets, models, etc.), features (sets, vectors), and configuration (parameters, secrets , source, etc.). Projects have owners and members with role-based access control.

Projects are stored in a GIT or archive and map to IDE projects (in PyCharm, VSCode, etc.), which enables versioning, collaboration, and CI/CD. Projects simplify how you process data, submit jobs, run multi-stage workflows, and deploy real-time pipelines in continious development or production environments.

In this section

5.1 Create, save, and use projects

A project is a container for all the assets, configuration, and code of a particular application. It is the starting point for your work. Projects are stored in a versioned source repository (GIT) or archive and can map to IDE projects (in PyCharm, VSCode, etc.).

In this section

- *Creating a new project*
- *Adding functions, artifacts, workflow, and config*
- *Pushing the project content into git or an archive*
- *Get a project from DB or create it*

5.1.1 Creating a new project

Project files (code, configuration, etc.) are stored in a directory (the project `context` path) and can be pushed to, or loaded from, the source repository. See the following project directory example:

```
my-project           # Parent directory of the project (context)
├── data             # Project data for local tests or outputs (not tracked by version␣
↪control)
├── docs             # Project documentation
├── src              # Project source code (functions, libs, workflows)
├── tests            # Unit tests (pytest) for the different functions
├── project.yaml     # MLRun project spec file
├── README.md        # Project README
└── requirements.txt # Default Python requirements file (can have function specific␣
↪requirements as well)
```

(continues on next page)

(continued from previous page)

To define a new project from scratch, use *new_project()*. You must specify a `name` and a location for the `context` directory (e.g. `"./"`). There are additional, optional parameters. The `context` dir holds the configuration, code, and workflow files. File paths in the project are relative to the context root. The `user_project` flag indicates that the project name is unique per user, and the `init_git` flag is used to initialize git in the context dir.

```
import mlrun
project = mlrun.new_project("myproj", "./", user_project=True,
                            init_git=True, description="my new project")
```

Projects can also be created from a template (yaml file, zip file, or git repo), allowing users to create reusable skeletons. The content of the zip/tar/git archive is copied into the context dir. The `remote` attribute can be used to register a remote git repository URL.

Example of creating a new project from a zip template:

```
# create a project from zip, initialize a local git, and register the git remote path
project = mlrun.new_project("myproj", "./", init_git=True, user_project=True,
                            remote="git://github.com/myorg/some-project.git",
                            from_template="http://mysite/proj.zip")
```

5.1.2 Adding functions, artifacts, workflow, and config

Projects host functions, workflows, artifacts (files, datasets, models, etc.), features, and configuration (parameters, secrets, source, etc.). This section explains how to add or register different project elements. For details on the feature store and its elements (sets, vectors) see the **feature store documentation**.

Adding and registering functions:

Functions with basic attributes such as code, requirements, image, etc. can be registered using the *set_function()* method. Functions can be created from a single code/notebook file or have access to the entire project context directory. (By adding the `with_repo=True` flag, the project context is cloned into the function runtime environment.) See the examples:

```
# register a (single) python file as a function
project.set_function('src/data_prep.py', 'data-prep', image='mlrun/mlrun', handler='prep
↪', kind="job")

# register a notebook file as a function, specify custom image and extra requirements
project.set_function('src/mynb.ipynb', name='test-function', image="my-org/my-image",
                     handler="run_test", requirements="requirements.txt", kind="job")

# register a module.handler as a function (requires defining the default sources/work␣
↪dir, if it's not root)
project.spec.workdir = "src"
project.set_function(name="train", handler="training.train",  image="mlrun/mlrun", kind=
↪"job", with_repo=True)
```

See details and examples on how to **create and register functions**, how to **annotate notebooks** (to be used as functions), how to **run, build, or deploy** functions, and how to **use them in workflows**.

Register artifacts:

Artifacts are used by functions and workflows and are referenced by a key (name) and optional tag (version). Users can define artifact files or objects in the project spec, which are registered during project load or when calling `project.register_artifacts()`. To register artifacts use the *set_artifact()*) method. See the examples:

```
# register a simple file artifact in the project (point to remote object)
data_url = 'https://s3.wasabisys.com/iguazio/data/iris/iris.data.raw.csv'
project.set_artifact('data', target_path=data_url)

# register a model artifact
project.set_artifact('model', ModelArtifact(model_file="model.pkl"), target_path=model_
↪dir_url)

# register local or remote artifact object (yaml or zip), will be imported on project␣
↪load
# to generate such a package use `artifact.export(zip_path)`
project.set_artifact('model', 'https://mystuff.com/models/mymodel.zip')
```

Note

Local file paths are relative to the context dir.

Registering workflows:

Projects contain one or more workflows (pipelines). The workflows can be registered using the `set_workflow()`) method. Project workflows are executed using the *run()*) method. See **building and running workflows** for details.

```
# Add a multi-stage workflow (./myflow.py) to the project with the name 'main' and save␣
↪the project
project.set_workflow('main', "./src/workflow.py")
```

Set project wide parameters and secrets:

You can define global project parameters and secrets and use them in your functions enabling simple configuration and templates. See the examples:

```
# Read env vars from dict or file and set as project secrets
project.set_secrets({"SECRET1": "value"})
project.set_secrets(file_path="secrets.env")

project.spec.params = {"x": 5}
```

Note

Secrets are not loaded automatically (not part of the `project.yaml`); you need to apply `set_secrets()` methods manually or use the UI.

Project parameters, secrets and configuration can also be set in the UI, in the relevant project, click the 

button at the left bottom corner.

Example, secrets configuration screen:

Save the project:

Use the *save()* method to store all the definitions (functions, artifacts, workflows, parameters, etc.) in the MLRun DB and in the `project.yaml` file (for automated loading and CI/CD).

```
project.save()
```

show an example project.yaml file

The generated `project.yaml` for the above project looks like:

```
ind: project
metadata:
  name: myproj
spec:
  description: my new project
  params:
    x: 5
  functions:
  - url: src/data_prep.py
    name: data-prep
    image: mlrun/mlrun
    handler: prep
  - url: src/mynb.ipynb
    name: test-function
    kind: job
    image: my-org/my-image
    handler: run_test
    requirements: requirements.txt
  - name: train
    kind: job
    image: mlrun/mlrun
    handler: training.train
    with_repo: true
  workflows:
  - path: ./src/workflow.py
    name: main
  artifacts:
  - kind: artifact
    metadata:
      project: myproj
      key: data
    spec:
      target_path: https://s3.wasabisys.com/iguazio/data/iris/iris.data.raw.csv
  source: ''
  workdir: src```
```

5.1.3 Pushing the project content into git or an archive

Project code, metadata, and configuration are stored and versioned in source control systems like GIT or archives (zip, tar). This allows **loading an entire project** (with a specific version) into a development or production environment, or seamlessly **integrating with CI/CD frameworks**.

Note

You must push the updates before you build functions or run workflows that use code from git, since the builder or containers pull the code from the git repo.

Use standard Git commands to push the current project tree into a git archive. Make sure you `.save()` the project before pushing it

```
git remote add origin <server>
git commit -m "Commit message"
git push origin master
```

Alternatively you can use MLRun SDK calls:

- *create_remote()* - to register the remote Git path
- *push()* - save project spec (`project.yaml`) and commit/push updates to remote repo

```
{admonition} Note
If you are using containerized Jupyter you might need to first set your Git parameters,␣
↪e.g. using the following commands:
```

```
git config --global user.email "<my@email.com>"
git config --global user.name "<name>"
git config --global credential.helper store
```

You can also save the project content and metadata into a local or remote `.zip` archive, for example:

```
project.export("../archive1.zip")
project.export("s3://my-bucket/archive1.zip")
project.export(f"v3io://projects/{project.name}/archive1.zip")
```

5.1.4 Get a project from DB or create it

If you already have a project saved in the DB and you need to access/use it (for example, from a different notebook or file), use the *get_or_create_project()* method. It first tries to read the project from the DB, and only if it doesn't exist in the DB it loads/creates it.

Note

If you update the project object from different files/notebooks/users, make sure you `.save()` your project after a change, and run `get_or_create_project` to load changes made by others.

Example:

```
    # load project from the DB (if exists) or the source repo
    project = mlrun.get_or_create_project("myproj", "./", "git://github.com/mlrun/demo-
↪xgb-project.git")
    project.pull("development")  # pull the latest code from git
    project.run("main", arguments={'data': data_url})  # run the workflow "main"
```

5.2 Load and run projects

Project code, metadata, and configuration are stored and versioned in source control systems like Git or archives (zip, tar) and can be loaded into your work environment or CI system with a single SDK or CLI command.

The project root (context) directory contains the `project.yaml` file with the required metadata and links to various project files/objects, and is read during the `load` process.

In this section

- *Load projects using the SDK*
- *Load projects using the CLI*

See also details on loading and using projects **with CI/CD frameworks**.

5.2.1 Load projects using the SDK

When a project is already created and stored in a local dir, git, or archive, you can quickly load and use it with the *load_project()* method. `load_project` uses a local context directory (with initialized `git`) or clones a remote repo into the local dir and returns a project object.

You need to provide the path to the `context` dir and the git/zip/tar archive `url`. The `name` can be specified or taken from the project object. The project can also specify `secrets` (dict with repo credentials), `init_git` flag (initializes Git in the context dir), `clone` flag (project is cloned into the context dir, and the local copy is ignored/deleted), and `user_project` flag (indicates the project name is unique to the user).

Example of loading a project from git and running the `main` workflow:

```
# load the project and run the 'main' workflow
project = load_project(context="./", name="myproj", url="git://github.com/mlrun/project-
↪archive.git")
project.run("main", arguments={'data': data_url})
```

Note

If the `url` parameter is not specified it searches for Git repo inside the context dir and uses its metadata, or if the flag init_git=True, it initializes a Git repo in the target context directory.

After the project object is loaded use the *run()* method to execute workflows. See details on **building and running workflows**), and how to **run, build, or deploy** individual functions.

You can edit or add project elements like functions, workflows, artifacts, etc. (See **create and use projects**.) Once you make changes use GIT or MLRun commands to push those changes to the archive (See **save into git or an archive**.)

5.2.2 Load projects using the CLI

Loading a project from `git` into `./` :

```
mlrun project -n myproj --url "git://github.com/mlrun/project-demo.git" .
```

Running a specific workflow (`main`) from the project stored in `.` (current dir):

```
mlrun project --run main --watch .
```

CLI usage details

```
Usage: mlrun project [OPTIONS] [CONTEXT]

Options:
  -n, --name TEXT           project name
  -u, --url TEXT            remote git or archive url
  -r, --run TEXT            run workflow name of .py file
  -a, --arguments TEXT      Kubeflow pipeline arguments name and value tuples
                            (with -r flag), e.g. -a x=6
  -p, --artifact-path TEXT  output artifacts path
  -x, --param TEXT          mlrun project parameter name and value tuples,
                            e.g. -p x=37 -p y='text'
  -s, --secrets TEXT        secrets file=<filename> or env=ENV_KEY1,..
  --db TEXT                 api and db service path/url
  --init-git                for new projects init git context
  -c, --clone               force override/clone into the context dir
  --sync                    sync functions into db
  -w, --watch               wait for pipeline completion (with -r flag)
  -d, --dirty               allow run with uncommitted git changes
  --handler TEXT            workflow function handler name
  --engine TEXT             workflow engine (kfp/local/remote)
  --local                   try to run workflow functions locally
  --timeout INTEGER         timeout in seconds to wait for pipeline completion
                            (used when watch=True)
  --env-file TEXT           path to .env file to load config/variables from
  --ensure-project          ensure the project exists, if not, create project
  --schedule TEXT           To create a schedule define a standard crontab
                            expression string. For using the
                            pre-defined workflow's schedule, set --schedule 'true'
```

5.3 Run, build, and deploy functions

In this section

5.3.1 Overview

There is a set of methods used to deploy and run project functions. They can be used interactively or inside a pipeline (e.g. Kubeflow). When used inside a pipeline, each method is automatically mapped to the relevant pipeline engine command.

- *run_function()* — Run a local or remote task as part of local or remote batch/scheduled task
- *build_function()* — deploy an ML function, build a container with its dependencies for use in runs
- *deploy_function()* — deploy real-time/online (nuclio or serving based) functions

You can use those methods as `project` methods, or as global (`mlrun.`) methods. For example:

```
# run the "train" function in myproject
run = myproject.run_function("train", inputs={"data": data_url})

# run the "train" function in the current/active project (or in a pipeline)
run = mlrun.run_function("train", inputs={"data": data_url})
```

The first parameter in all three methods is either the function name (in the project), or a function object, used if you want to specify functions that you imported/created ad hoc, or to modify a function spec. For example:

```
# import a serving function from the Function Hub and deploy a trained model over it
serving = import_function("hub://v2_model_server", new_name="serving")
serving.spec.replicas = 2
deploy = deploy_function(
    serving,
    models=[{"key": "mymodel", "model_path": train.outputs["model"]}],
)
```

You can use the *get_function()* method to get the function object and manipulate it, for example:

```
trainer = project.get_function("train")
trainer.with_limits(mem="2G", cpu=2, gpus=1)
run = project.run_function("train", inputs={"data": data_url})
```

5.3.2 run_function

Use the *run_function()* method to run a local or remote batch/scheduled task. The `run_function` method accepts various parameters such as `name`, `handler`, `params`, `inputs`, `schedule`, etc. Alternatively, you can pass a **`Task`** object (see: *new_task()*) that holds all of the parameters and the advanced options.

Functions can host multiple methods (handlers). You can set the default handler per function. You need to specify which handler you intend to call in the run command. You can pass `parameters` (arguments) or data `inputs` (such as datasets, feature-vectors, models, or files) to the functions through the `run_function` method.

The *run_function()* command returns an MLRun *RunObject* object that you can use to track the job and its results. If you pass the parameter `watch=True` (default), the command blocks until the job completes.

MLRun also supports iterative jobs that can run and track multiple child jobs (for hyperparameter tasks, AutoML, etc.). See *Hyperparameter tuning optimization* for details and examples.

Read further details on **running tasks and getting their results**.

Usage examples:

```
# create a project with two functions (local and from Function Hub)
project = mlrun.new_project(project_name, "./proj")
project.set_function("mycode.py", "prep", image="mlrun/mlrun")
project.set_function("hub://auto_trainer", "train")

# run functions (refer to them by name)
run1 = project.run_function("prep", params={"x": 7}, inputs={'data': data_url})
run2 = project.run_function("train", inputs={"dataset": run1.outputs["data"]})
run2.artifact('confusion-matrix').show()
```

Run/simulate functions locally:

Functions can also run and be debugged locally by using the `local` runtime or by setting the `local=True` parameter in the *run()* method (for batch functions).

5.3.3 build_function

The *build_function()* method is used to deploy an ML function and build a container with its dependencies for use in runs.

Example:

```
# build the "trainer" function image (based on the specified requirements and code repo)
project.build_function("trainer")
```

The *build_function()* method accepts different parameters that can add to, or override, the function build spec. You can specify the target or base `image` extra docker `commands`, builder environment, and source credentials (`builder_env`), etc.

See further details and examples in **Build function image**.

5.3.4 deploy_function

The *deploy_function()* method is used to deploy real-time/online (nuclio or serving) functions and pipelines. Read more about **Real-time serving pipelines**.

Basic example:

```
# Deploy a real-time nuclio function ("myapi")
deployment = project.deploy_function("myapi")

# invoke the deployed function (using HTTP request)
resp = deployment.function.invoke("/do")
```

You can provide the `env` dict with: extra environment variables; `models` list to specify specific models and their attributes (in the case of serving functions); builder environment; and source credentials (`builder_env`).

Example of using `deploy_function` inside a pipeline, after the `train` step, to generate a model:

```
# Deploy the trained model (from the "train" step) as a serverless serving function
serving_fn = mlrun.new_function("serving", image="mlrun/mlrun", kind="serving")
mlrun.deploy_function(
    serving_fn,
    models=[
        {
            "key": model_name,
            "model_path": train.outputs["model"],
            "class_name": 'mlrun.frameworks.sklearn.SklearnModelServer',
        }
    ],
)
```

Note

If you want to create a simulated (mock) function instead of a real Kubernetes service, set the `mock` flag is set to `True`. See deploy_function api.

5.4 Build and run workflows/pipelines

This section shows how to write a batch pipeline so that it can be executed via an MLRun Project. With a batch pipelineyou can use the MLRun Project to execute several Functions in a DAG using the Python SDK or CLI.

This example creates a project with three MLRun functions and a single pipeline that orchestrates them. The pipeline steps are:

- `get-data` — Get iris data from sklearn
- `train-model` — Train model via sklearn
- `deploy-model` — Deploy model to HTTP endpoint

```
import mlrun
project = mlrun.get_or_create_project(\"iguazio-academy\", context=\"./\")
```

5.4.1 Add functions to a project

Add the functions to a project:

```
project.set_function(name='get-data', func='functions/get_data.py', kind='job', image=
↪'mlrun/mlrun')
project.set_function(name='train-model', func='functions/train.py', kind='job', image=
↪'mlrun/mlrun'),
project.set_function(name='deploy-model', func='hub://v2_model_server')
```

5.4.2 Write a pipeline

Next, define the pipeline that orchestrates the three comoponents. This pipeline is simple, however you can create very complex pipelines with branches, conditions, and more.

```
%%writefile pipelines/training_pipeline.py
from kfp import dsl
import mlrun

@dsl.pipeline(
    name=\"batch-pipeline-academy\",
    description=\"Example of batch pipeline for Iguazio Academy\"
)
def pipeline(label_column: str, test_size=0.2):

    # Ingest the data set
    ingest = mlrun.run_function(
        'get-data',
        handler='prep_data',
        params={'label_column': label_column},
        outputs=[\"iris_dataset\"]
    )

    # Train a model
    train = mlrun.run_function(
        \"train-model\",
        handler=\"train_model\",
        inputs={\"dataset\": ingest.outputs[\"iris_dataset\"]},
        params={
            \"label_column\": label_column,
            \"test_size\" : test_size
        },
        outputs=['model']
    )

    # Deploy the model as a serverless function
    deploy = mlrun.deploy_function(
        \"deploy-model\",
        models=[{\"key\": \"model\", \"model_path\": train.outputs[\"model\"]}]
    )
```

5.4.3 Add a pipeline to a project

Add the pipeline to your project:

```
project.set_workflow(name='train', workflow_path=\"pipelines/training_pipeline.py")
project.save()
```

5.5 CI/CD integration

+You can run your ML Pipelines using CI frameworks like Github Actions, GitLab CI/CD, etc. MLRun supports a simple and native integration with the CI systems.

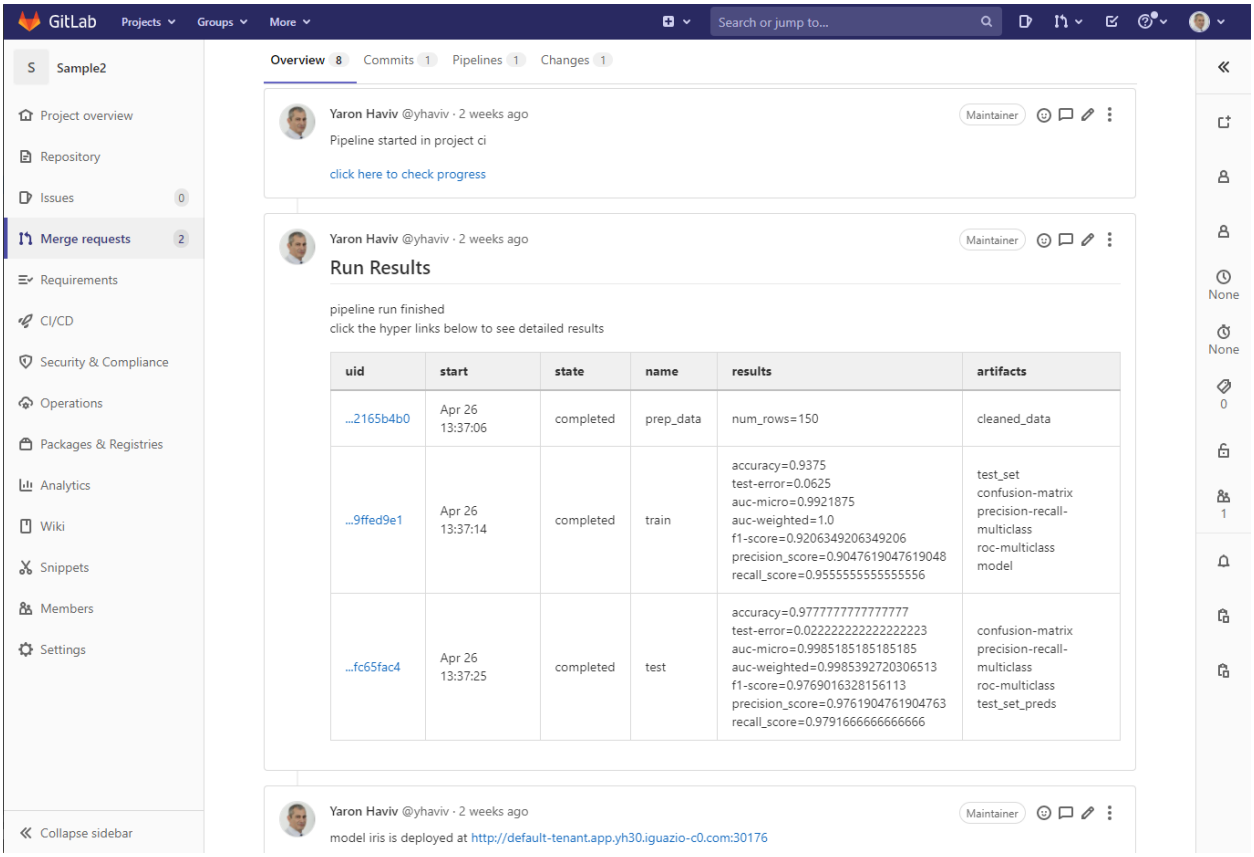
- Build/run complex workflows composed of local/library functions or external cloud services (e.g. AutoML)
- Support various Pipeline/CI engines (Kubeflow, GitHub, Gitlab, Jenkins)
- Track & version code, data, params, results with minimal effort
- Elastic scaling of each step
- Extensive Function Hub

MLRun workflows can run inside the CI system. The most common method is to use the CLI command `mlrun project` to load the project and run a workflow as part of a code update (e.g. pull request, etc.). The pipeline tasks are executed on the Kubernetes cluster, which is orchestrated by MLRun.

When MLRun is executed inside a GitHub Action or GitLab CI/CD pipeline it detects the environment attributes automatically (e.g. repo, commit id, etc.). In addition, a few environment variables and credentials must be set:

- **MLRUN_DBPATH** — url of the MLRun cluster.
- **V3IO_USERNAME** — username in the remote Iguazio cluster.
- **V3IO_ACCESS_KEY** — access key to the remote Iguazio cluster.
- **GIT_TOKEN** or **GITHUB_TOKEN** — Github/Gitlab API token (set automatically in Github Actions).
- **SLACK_WEBHOOK** — optional. Slack API key when using slack notifications.

When the workflow runs inside the Git CI system it reports the pipeline progress and results back into the Git tracking system, similar to:



Contents

5.5.1 Using GitHub Actions

When running using GitHub Actions you need to set the credentials/secrets and add a script under the `.github/workflows/` directory, which is executed when the code is commited/pushed.

Example script that is invoked when you add the comment "/run" to your pull request:

```
name: mlrun-project-workflow
on: [issue_comment]

jobs:
  submit-project:
    if: github.event.issue.pull_request != null && startsWith(github.event.comment.body,
↪'/run')
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
    - name: Set up Python 3.7
```

(continues on next page)

(continued from previous page)

```
      uses: actions/setup-python@v4
      with:
        python-version: '3.7'
        architecture: 'x64'

    - name: Install mlrun
      run: python -m pip install pip install mlrun
    - name: Submit project
      run: python -m mlrun project ./ -w -r main ${CMD:5}
      env:
        V3IO_USERNAME: ${{ secrets.V3IO_USERNAME }}
        V3IO_API: ${{ secrets.V3IO_API }}
        V3IO_ACCESS_KEY: ${{ secrets.V3IO_ACCESS_KEY }}
        MLRUN_DBPATH: ${{ secrets.MLRUN_DBPATH }}
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        SLACK_WEBHOOK: ${{ secrets.SLACK_WEBHOOK }}
        CMD: ${{ github.event.comment.body}}
```

See the full example in **https://github.com/mlrun/project-demo**

5.5.2 Using GitLab CI/CD

When running using GitLab CI/CD you need to set the credentials/secrets and update the script `.gitlab-ci.yml` directory, which is executed when code is commited/pushed.

Example script that is invoked when you create a pull request (merge requests):

```
image: mlrun/mlrun

run:
  script:
    - python -m mlrun project ./ -w -r ci
  only:
    - merge_requests
```

See the full example in **https://gitlab.com/yhaviv/test2**

5.5.3 Using Jenkins Pipeline

When using Jenkins Pipeline you need to set up the credentials/secrets in Jenkins and and update the script `Jenkinsfile` in your codebase. You can trigger the Jenkins pipeline either through Jenkins triggers or through the GitHub webhooks.

Example `Jenkinesfile` that is invoked when you start a Jenkins pipeline (via triggers or GitHub webhooks):

```
pipeline {
   agent any
    environment {
      RELEASE='1.0.0'
      PROJECT_NAME='project-demo'
    }
   stages {
```

(continues on next page)

(continued from previous page)

```
      stage('Audit tools') {
         steps{
            auditTools()
         }
      }
      stage('Build') {
            environment {
               MLRUN_DBPATH='https://mlrun-api.default-tenant.app.us-sales-341.iguazio-
↪cd1.com'
               V3IO_ACCESS_KEY=credentials('V3IO_ACCESS_KEY')
               V3IO_USERNAME='xingsheng'
            }
            agent {
                docker {
                    image 'mlrun/mlrun:1.2.0'
                }
            }
            steps {
               echo "Building release ${RELEASE} for project ${PROJECT_NAME}..."
               sh 'chmod +x build.sh'
               withCredentials([string(credentialsId: 'an-api-key', variable: 'API_KEY
↪')]) {
                  sh '''
                     ./build.sh
                  '''
               }
            }
        }
        stage('Test') {
            steps {
               echo "Testing release ${RELEASE}"
            }
        }
   }
   post {
      success {
         slackSend channel: '#builds',
                   color: 'good',
                   message: "Project ${env.PROJECT_NAME}, success: ${currentBuild.
↪fullDisplayName}."
      }
      failure {
         slackSend channel: '#builds',
                   color: 'danger',
                   message: "Project ${env.PROJECT_NAME}, FAILED: ${currentBuild.
↪fullDisplayName}."
      }
   }
}

void auditTools() {
   sh '''
```

(continues on next page)

(continued from previous page)

```
      git version
      docker version
   '''
}
```

After the Jenkins pipeline is complete, you can see the MLRun job in the MLRun UI.

See the full example in **https://github.com/mlrun/project-demo**.

5.6 Working with secrets

When executing jobs through MLRun, the code might need access to specific secrets, for example to access data residing on a data-store that requires credentials (such as a private S3 bucket), or many other similar needs.

MLRun provides some facilities that allow handling secrets and passing those secrets to execution jobs. It's important to understand how these facilities work, as this has implications on the level of security they provide and how much exposure they create for your secrets.

In this section

5.6.1 Overview

There are two main use-cases for providing secrets to an MLRun job. These are:

- *Use MLRun-managed secrets*. This is a flow that enables the MLRun user (for example a data scientist or engineer) to create and use secrets through interfaces that MLRun implements and manages.
- *Create secrets externally* to MLRun using a Kubernetes secret or some other secret management framework (such as Azure vault), and utilize these secrets from within MLRun to enrich execution jobs. For example, the secrets are created and managed by an IT admin, and the data-scientist only accesses them.

The following sections cover the details of those two use-cases.

5.6.2 MLRun-managed secrets

The easiest way to pass secrets to MLRun jobs is through the MLRun project secrets mechanism. MLRun jobs automatically gain access to all project secrets defined for the same project. More details are available *later in this page*.

The following is an example of using project secrets:

```
# Create project secrets for the myproj project
project = mlrun.get_or_create_project("myproj", "./")
secrets = {'AWS_KEY': '111222333'}
project.set_secrets(secrets=secrets, provider="kubernetes")

# Create and run the MLRun job
function = mlrun.code_to_function(
    name="secret_func",
    filename="my_code.py",
    handler="test_function",
    kind="job",
    image="mlrun/mlrun"
)
function.run()
```

The handler defined in `my_code.py` accesses the `AWS_KEY` secret by using the *`get_secret()`* API:

```
def test_function(context):
    context.logger.info("running function")
    aws_key = context.get_secret("AWS_KEY")
    # Use aws_key to perform processing.
    ...
```

Using tasks with secrets

MLRun uses the concept of Tasks to encapsulate runtime parameters. Tasks are used to specify execution context such as hyper-parameters. They can also be used to pass details about secrets that are going to be used in the runtime. This allows for control over specific secrets passed to runtimes, and support for the various MLRun secret providers.

To pass secret parameters, use the Task's *`with_secrets()`* function. For example, the following command passes specific project-secrets to the execution context:

```
function = mlrun.code_to_function(
    name="secret_func",
    filename="my_code.py",
    handler="test_function",
    kind="job",
    image="mlrun/mlrun"
)
task = mlrun.new_task().with_secrets("kubernetes", ["AWS_KEY", "DB_PASSWORD"])
run = function.run(task, ...)
```

The *`with_secrets()`* function tells MLRun what secrets the executed code needs to access. The MLRun framework prepares the needed infrastructure to make these secrets available to the runtime, and passes information about them to the execution framework by specifying those secrets in the spec of the runtime. For example, if running a kubernetes job, the secret keys are noted in the generated pod's spec.

The actual details of MLRun's handling of the secrets differ per the **secret provider** used. The following sections provide more details on these providers and how they handle secrets and their values.

Regardless of the type of secret provider used, the executed code uses the *get_secret()* API to gain access to the value of the secrets passed to it, as shown in the above example.

Secret providers

MLRun provides several secret providers. Each of these providers functions differently and have different traits with respect to what secrets can be passed and how they're handled. It's important to understand these parameters to make sure secrets are not compromised and that their secrecy is maintained.

Warning: The *Inline*, *environment* and *file* providers do not guarantee confidentiality of the secret values handled by them, and **should only be used for development and demo purposes**. The *Kubernetes* and *Azure Vault* providers are secure and should be used for any other use-case.

Kubernetes project secrets

MLRun can use Kubernetes (k8s) secrets to store and retrieve secret values on a per-project basis. This method is supported for all runtimes that generate k8s pods. MLRun creates a k8s secret per project, and stores multiple secret keys within this secret. Project secrets can be created through the MLRun SDK as well as through the MLRun UI.

By default, all jobs in a project automatically get access to all the associated project secrets. There is no need to use `with_secrets` to provide access to project secrets.

Creating project secrets

To populate the MLRun k8s project secret with secret values, use the project object's *set_secrets()* function, which accepts a dictionary of secret values or a file containing a list of secrets. For example:

```
# Create project secrets for the myproj project.
project = mlrun.get_or_create_project("myproj", "./")
secrets = {'password': 'myPassw0rd', 'AWS_KEY': '111222333'}
project.set_secrets(secrets=secrets, provider="kubernetes")
```

Warning: This action should not be part of the code committed to `git` or part of ongoing execution - it is only a setup action, which normally should only be executed once. After the secrets are populated, this code should be removed to protect the confidentiality of the secret values.

The MLRun API does not allow the user to see project secrets values, but it does allow seeing the keys that belong to a given project, assuming the user has permissions on that specific project. See the *HTTPRunDB* class documentation for additional details.

When MLRun is executed in the Iguazio platform, the secret management APIs are protected by the platform such that only users with permissions to access and modify a specific project can alter its secrets.

Creating secrets in the Projects UI page

The Settings dialog in the Projects page, accessed with the Settings icon, has a Secrets tab where you can add secrets as key-value pairs. The secrets are automatically available to all jobs belonging to this project. Users with the Editor or Admin role can add, modify, and delete secrets, and assign new secret values. Viewers can only view the secret keys. The values themselves are not visible to any users.

Accessing the secrets

By default, any runtime not executed locally (`local=False`) automatically gains access to all the secrets of the project it belongs to, so no configuration is required to enable that. **Jobs that are executed locally (`local=True`) do not have access to the project secrets.** It is possible to limit access of an executing job to a subset of these secrets by calling the following function with a list of the secrets to be accessed:

```
task.with_secrets('kubernetes', ['password', 'AWS_KEY'])
```

When the job is executed, the MLRun framework adds environment variables to the pod spec whose value is retrieved through the k8s `valueFrom` option, with `secretKeyRef` pointing at the secret maintained by MLRun. As a result, this method does not expose the secret values at all, except inside the pod executing the code where the secret value is exposed through an environment variable. This means that even a user with `kubectl` looking at the pod spec cannot see the secret values.

Users, however, can view the secrets using the following methods:

- Run `kubectl` to view the actual contents of the k8s secret.
- Perform `kubectl exec` into the running pod, and examine the environment variables.

To maintain the confidentiality of secret values, these operations must be strictly limited across the system by using k8s RBAC and ensuring that elevated permissions are granted to a very limited number of users (very few users have and use elevated permissions).

Accessing secrets in nuclio functions

Nuclio functions do not have the MLRun context available to retrieve secret values. Secret values need to be retrieved from the environment variable of the same name. For example, to access the `AWS_KEY` secret in a nuclio function use:

```
aws_key = os.environ.get("AWS_KEY")
```

Azure Vault

MLRun can serve secrets from an Azure key Vault.

Note: Azure key Vaults support 3 types of entities - `keys`, `secrets` and `certificates`. MLRun only supports accessing `secret` entities.

Setting up access to Azure key vault

To enable this functionality, a secret must first be created in the k8s cluster that contains the Azure key Vault credentials. This secret should include credentials providing access to your specific Azure key Vault. To configure this, the following steps are needed:

1. Set up a key vault in your Azure subscription.
2. Create a service principal in Azure that will be granted access to the key vault. For creating a service principal through the Azure portal follow the steps listed in this page.
3. Assign a key vault access policy to the service principal, as described in this page.
4. Create a secret access key for the service principal, following the steps listed in this page. Make sure you have access to the following three identifiers:
   - Directory (tenant) id
   - Application (client) id
   - Secret key
5. Generate a k8s secret with those details. Use the following command:

```
kubectl -n <namespace> create secret generic <azure_key_vault_k8s_secret> \
   --from-literal=secret=<secret key> \
   --from-literal=tenant_id=<tenant id> \
   --from-literal=client_id=<client id>
```

Note: The names of the secret keys *must* be as shown in the above example, as MLRun queries them by these exact names.

Accessing Azure key vault secrets

Once these steps are done, use `with_secrets` in the following manner:

```
task.with_secrets(
    "azure_vault",
    {
        "name": <azure_key_vault_name>,
        "k8s_secret": <azure_key_vault_k8s_secret>,
        "secrets": [],
    },
)
```

The `name` parameter should point at your Azure key Vault name. The `secrets` parameter is a list of the secret keys to be accessed from that specific vault. If it's empty (as in the example above) then all secrets in the vault can be accessed by their key name.

For example, if the Azure Vault has a secret whose name is `MY_AZURE_SECRET` and using the above example for *`with_secrets()`*, the executed code can use the following statement to access this secret:

```
azure_secret = context.get_secret("MY_AZURE_SECRET")
```

In terms of confidentiality, the executed pod has the Azure secret provided by the user mounted to it. This means that the access-keys to the vault are visible to a user that `exec`s into the pod in question. The same security rules should be followed as described in the *Kubernetes* section above.

Demo/Development secret providers

The rest of the MLRun secret providers are not secure by design, and should only be used for demonstration or development purposes.

Inline

The inline secrets provider is a very basic framework that should mostly be used for testing and demos. The secrets passed by this framework are exposed in the source code creating the MLRun function, as well as in the function spec, and in the generated pod specs. To add inline secrets to a job, perform the following:

```
task.with_secrets("inline", {"MY_SECRET": "12345"})
```

As can be seen, even the client code exposes the secret value. If this is used to pass secrets to a job running in a kubernetes pod, the secret is also visible in the pod spec. This means that any user that can run `kubectl` and is permitted to view pod specs can also see the secret keys and their values.

Environment

Environment variables are similar to the `inline` secrets, but their client-side value is not specified directly in code but rather is extracted from a client-side environment variable. For example, if running MLRun on a Jupyter notebook and there are environment variables named `MY_SECRET` and `ANOTHER_SECRET` on Jupyter, the following code passes those secrets to the executed runtime:

```
task.with_secrets("env", "MY_SECRET, ANOTHER_SECRET")
```

When generating the runtime execution environment (for example, pod for the `job` runtime), MLRun retrieves the value of the environment variable and places it in the pod spec. This means that a user with `kubectl` capabilities who can see pod specs can still see the secret values passed in this manner.

File

The file provider is used to pass secret values that are stored in a local file. The file needs to be made of lines, each containing a secret and its value separated by =. For example:

```
# secrets.txt
SECRET1=123456
SECRET2=abcdef
```

Use the following command to add these secrets:

```
task.with_secrets("file", "/path/to/file/secrets.txt")
```

5.6.3 Externally managed secrets

MLRun provides facilities to map k8s secrets that were created externally to jobs that are executed. To enable that, the spec of the runtime that is created should be modified by mounting secrets to it - either as files or as environment variables containing specific keys from the secret.

Mapping secrets to environment

Let's assume a k8s secret called `my-secret` was created in the same k8s namespace where MLRun is running, with two keys in it - `secret1` and `secret2`. The following example adds these two secret keys as environment variables to an MLRun job:

```
function = mlrun.code_to_function(
    name="secret_func",
    handler="test_function",
    ...
)

function.set_env_from_secret(
    "SECRET_ENV_VAR_1", secret="my-secret", secret_key="secret1"
)
function.set_env_from_secret(
    "SECRET_ENV_VAR_2", secret="my-secret", secret_key="secret2"
)
```

This only takes effect for functions executed remotely, as the secret value is injected to the function pod, which does not exist for functions executed locally. Within the function code, the secret values will be exposed as regular environment variables, for example:

```
# Function handler
def test_function(context):
    # Getting the value in the secret2 key.
    my_secret_value = os.environ.get("SECRET_ENV_VAR_2")
    ...
```

Mapping secrets as files

A k8s secret can be mapped as a filesystem folder to the function pod using the *mount_secret()* function:

```
# Mount all keys in the secret as files under /mnt/secrets
function.mount_secret("my-secret", "/mnt/secrets/")
```

This creates two files in the function pod, called `/mnt/secrets/secret1` and `/mnt/secrets/secret2`. Reading these files provide the values. It is possible to limit the keys mounted to the function - see the documentation of *mount_secret()* for more details.

CHAPTER

SIX

SERVERLESS FUNCTIONS

All the executions in MLRun are based on **Serverless functions**. The functions allow specifying code and all the operational aspects (image, required packages, cpu/mem/gpu resources, storage, environment, etc.). The different function runtimes take care of automatically transforming the code and spec to fully managed and elastic services over Kubernetes, which saves significant operational overhead, addresses scalability and reduces infrastructure costs.

MLRun supports:

- Real-time functions for: *serving*, APIs, and stream processing (based on the high-performance *Nuclio* engine).
- Batch functions (based on Kubernetes jobs, *Spark*, *Dask*, *Horovod*, etc.)

Function objects are all inclusive (code, spec, API, and metadata definitions), which allows placing them in a shared and versioned function market place. This means that different members of the team can produce or consume functions. Each function is versioned and stored in the MLRun database with a unique hash code, and gets a new hash code upon changes.

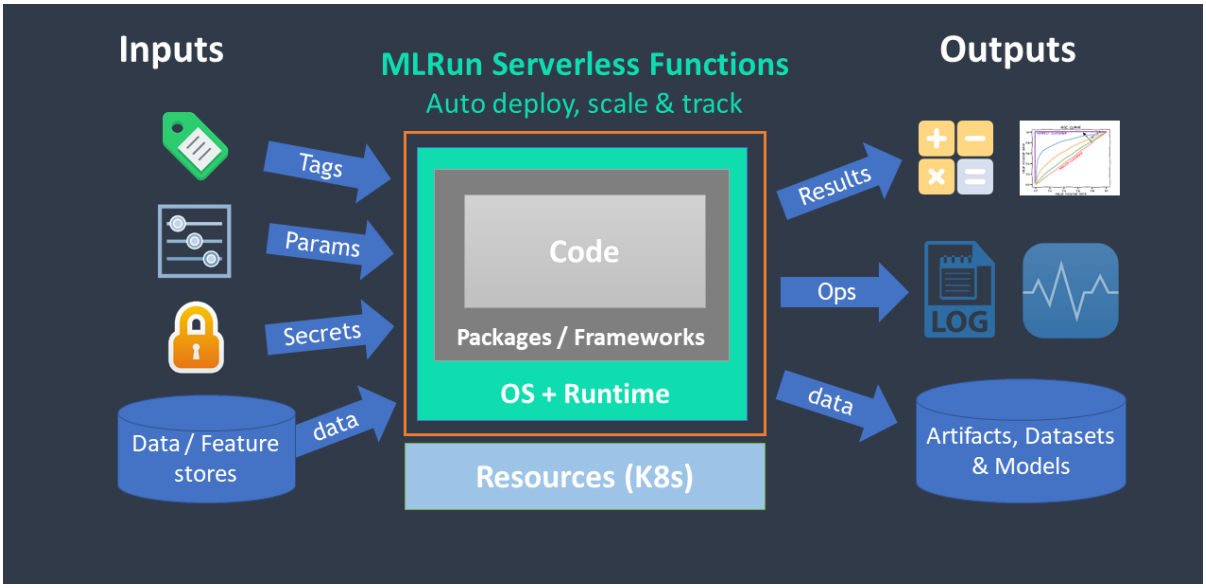
In this section

6.1 Functions architecture

MLRun supports:

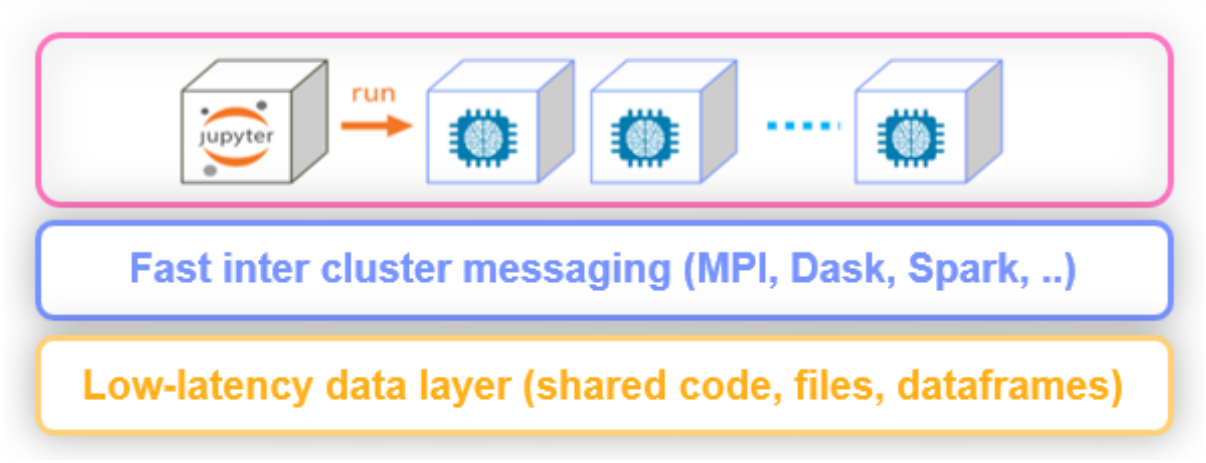
- Multiple types of runtimes.
- Configuring the function resources (replicas, CPU/GPU/memory limits, volumes, Spot vs. On-demand nodes, pod priority, node affinity). See details in Managing job resources.
- Iterative tasks for automatic and distributed execution of many tasks with variable parameters (hyperparams). See Hyperparam and iterative jobs.
- Horizontal scaling of functions across multiple containers. See Distributed and Parallel Jobs.

MLRun has an open public Function Hub that stores many pre-developed functions for use in your projects.



6.1.1 Distributed functions

Many of the runtimes support horizontal scaling. You can specify the number of `replicas` or the min—max value range (for auto scaling in *Dask* or *Nuclio*). When scaling functions, MLRun uses a high-speed messaging protocol and shared storage (volumes, objects, databases, or streams). MLRun runtimes handle the orchestration and monitoring of the distributed task.



6.2 Kinds of functions (runtimes)

When you create an MLRun function you need to specify a runtime kind (e.g. `kind='job'`). Each runtime supports its own specific attributes (e.g. Jars for Spark, Triggers for Nuclio, Auto-scaling for Dask, etc.).

MLRun supports real-time and batch runtimes.

Real-time runtimes:

- ***nuclio*** - real-time serverless functions over Nuclio
- ***serving*** - higher level real-time Graph (DAG) over one or more Nuclio functions

Batch runtimes:

- **handler** - execute python handler (used automatically in notebooks or for debug)
- **local** - execute a Python or shell program
- **job** - run the code in a Kubernetes Pod
- ***dask*** - run the code as a Dask Distributed job (over Kubernetes)
- ***mpijob*** - run distributed jobs and Horovod over the MPI job operator, used mainly for deep learning jobs
- ***spark*** - run the job as a Spark job (using Spark Kubernetes Operator)
- **remote-spark** - run the job on a remote Spark service/cluster (e.g. Iguazio Spark service)

Common attributes for Kubernetes based functions

All the Kubernetes based runtimes (Job, Dask, Spark, Nuclio, MPIJob, Serving) support a common set of spec attributes and methods for setting the Pods:

function.spec attributes (similar to k8s pod spec attributes):

- volumes
- volume_mounts
- env
- resources
- replicas
- image_pull_policy
- service_account
- image_pull_secret

common function methods:

- set_env(name, value)
- set_envs(env_vars)
- gpus(gpus, gpu_type)
- with_limits(mem, cpu, gpus, gpu_type)
- with_requests(mem, cpu)
- set_env_from_secret(name, secret, secret_key)

In this section

6.2.1 Dask distributed runtime

Quick Links

- Running Dask Over MLRun
- Pipelines Using Dask, Kubeflow and MLRun

Dask overview

Source: Dask docs Dask is a flexible library for parallel computing in Python.

Dask is composed of two parts:

1. **Dynamic task scheduling** optimized for computation. This is similar to Airflow, Luigi, Celery, or Make, but optimized for interactive computational workloads.
2. **"Big Data" collections** like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

Dask emphasizes the following virtues:

- **Familiar**: Provides parallelized NumPy array and Pandas DataFrame objects
- **Flexible**: Provides a task scheduling interface for more custom workloads and integration with other projects.
- **Native**: Enables distributed computing in pure Python with access to the PyData stack.
- **Fast**: Operates with low overhead, low latency, and minimal serialization necessary for fast numerical algorithms
- **Scales up**: Runs resiliently on clusters with 1000s of cores
- **Scales down**: Trivial to set up and run on a laptop in a single process
- **Responsive**: Designed with interactive computing in mind, it provides rapid feedback and diagnostics to aid humans Dask collections and schedulers

Dask DataFrame mimics Pandas

```
import pandas as pd                     import dask.dataframe as dd
df = pd.read_csv('2015-01-01.csv')      df = dd.read_csv('2015-*-*.csv')
df.groupby(df.user_id).value.mean()     df.groupby(df.user_id).value.mean().compute()
```

Dask Array mimics NumPy - documentation

```
import numpy as np                      import dask.array as da
f = h5py.File('myfile.hdf5')            f = h5py.File('myfile.hdf5')
x = np.array(f['/small-data'])          x = da.from_array(f['/big-data'],
                                                          chunks=(1000, 1000))
x - x.mean(axis=1)                      x - x.mean(axis=1).compute()
```

Dask Bag mimics iterators, Toolz, and PySpark - documentation

```
import dask.bag as db
b = db.read_text('2015-*-*.json.gz').map(json.loads)
b.pluck('name').frequencies().topk(10, lambda pair: pair[1]).compute()
```

Dask Delayed mimics for loops and wraps custom code - documentation

```
from dask import delayed
L = []
for fn in filenames:                    # Use for loops to build up computation
    data = delayed(load)(fn)            # Delay execution of function
```

(continues on next page)

(continued from previous page)

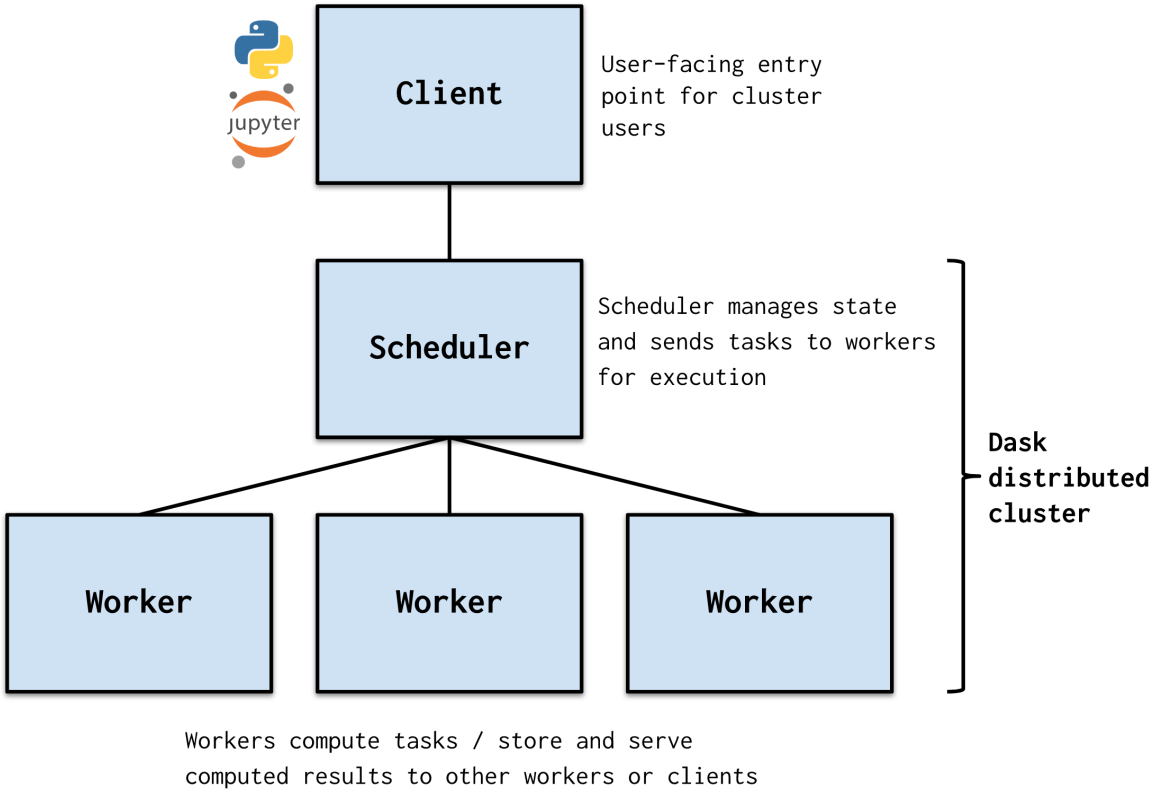
```
    L.append(delayed(process)(data))  # Build connections between variables

result = delayed(summarize)(L)
result.compute()
```

The concurrent.futures interface provides general submission of custom tasks: - documentation

```
from dask.distributed import Client
client = Client('scheduler:port')

futures = []
for fn in filenames:
    future = client.submit(load, fn)
    futures.append(future)

summary = client.submit(summarize, futures)
summary.result()
```

Dask.distributed

`Dask.distributed` is a lightweight library for distributed computing in Python. It extends both the concurrent.futures and dask APIs to moderate sized clusters.



Motivation

Distributed serves to complement the existing PyData analysis stack. In particular it meets the following needs:

- **Low latency**: Each task suffers about 1ms of overhead. A small computation and network roundtrip can complete in less than 10ms.
- **Peer-to-peer data sharing**: Workers communicate with each other to share data. This removes central bottlenecks for data transfer.
- **Complex Scheduling**: Supports complex workflows (not just map/filter/reduce) which are necessary for sophisticated algorithms used in nd-arrays, machine learning, image processing, and statistics.
- **Pure Python**: Built in Python using well-known technologies. This eases installation, improves efficiency (for Python users), and simplifies debugging.
- **Data Locality**: Scheduling algorithms cleverly execute computations where data lives. This minimizes network traffic and improves efficiency.
- **Familiar APIs**: Compatible with the concurrent.futures API in the Python standard library. Compatible with dask API for parallel algorithms
- **Easy Setup**: As a Pure Python package distributed is pip installable and easy to set up on your own cluster.

Architecture

`Dask.distributed` is a centrally managed, distributed, dynamic task scheduler. The central `dask-scheduler` process coordinates the actions of several `dask-worker` processes spread across multiple machines and the concurrent requests of several clients.

The scheduler is asynchronous and event driven, simultaneously responding to requests for computation from multiple clients and tracking the progress of multiple workers. The event-driven and asynchronous nature makes it flexible to concurrently handle a variety of workloads coming from multiple users at the same time while also handling a fluid worker population with failures and additions. Workers communicate amongst each other for bulk data transfer over TCP.

Internally the scheduler tracks all work as a constantly changing directed acyclic graph of tasks. A task is a Python function operating on Python objects, which can be the results of other tasks. This graph of tasks grows as users submit more computations, fills out as workers complete tasks, and shrinks as users leave or become disinterested in previous results.

Users interact by connecting a local Python session to the scheduler and submitting work, either by individual calls to the simple interface `client.submit(function, *args, **kwargs)` or by using the large data collections and parallel algorithms of the parent dask library. The collections in the dask library like `dask.array` and `dask.dataframe` provide easy access to sophisticated algorithms and familiar APIs like NumPy and Pandas, while the simple `client.submit` interface provides users with custom control when they want to break out of canned "big data" abstractions and submit fully custom workloads.

~5X Faster with Dask

Short example which demonstrates the power of Dask, in this notebook we will preform the following:

- Generate random text files
- Process the file by sorting and counting it's content
- Compare run times

Generate random text files

```
import random
import string
import os

from collections import Counter
from dask.distributed import Client

import warnings
warnings.filterwarnings('ignore')
```

```
def generate_big_random_letters(filename, size):
    """
    generate big random letters/alphabets to a file
    :param filename: the filename
    :param size: the size in bytes
    :return: void
    """
    chars = ''.join([random.choice(string.ascii_letters) for i in range(size)]) #1

    with open(filename, 'w') as f:
        f.write(chars)
    pass
```

```
PATH = '/User/howto/dask/random_files'
SIZE = 10000000

for i in range(100):
    generate_big_random_letters(filename = PATH + '/file_' + str(i) + '.txt',
                                size = SIZE)
```

Setfunction for benchmark

```
def count_letters(path):
    """
    count letters in text file
    :param path:  path to file
    """
    # open file in read mode
    file = open(path, "r")
```

(continues on next page)

(continued from previous page)

```
    # read the content of file
    data = file.read()

    # sort file
    sorted_file = sorted(data)

    # count file
    number_of_characters = len(sorted_file)

    return number_of_characters
```

```
def process_files(path):
    """
    list file and count letters
    :param path: path to folder with files
    """
    num_list = []
    files = os.listdir(path)

    for file in files:
        cnt = count_letters(os.path.join(path, file))
        num_list.append(cnt)

    l = num_list
    return print("done!")
```

Sort & count number of letters with Python

```
%%time
PATH = '/User/howto/dask/random_files/'
process_files(PATH)
```

```
done!
CPU times: user 2min 19s, sys: 9.31 s, total: 2min 29s
Wall time: 2min 32s
```

Sort & count number of letters with Dask

```
# get the dask client address
client = Client()
```

```
# list all files in folder
files = [PATH + x for x in os.listdir(PATH)]
```

```
%%time
# run the count_letter function on a list of files while using multiple workers
a = client.map(count_letters, files)
```

```
CPU times: user 13.2 ms, sys: 983 µs, total: 14.2 ms
Wall time: 12.2 ms
```

```
%%time
# gather results
l = client.gather(a)
```

```
CPU times: user 3.39 s, sys: 533 ms, total: 3.92 s
Wall time: 40 s
```

Additional topics

Running Dask on the cluster with mlrun

The dask frameworks enables users to parallelize their python code and run it as a distributed process on Iguazio cluster and dramatically accelerate their performance. In this notebook you'll learn how to create a dask cluster and then an mlrun function running as a dask client. It also demonstrates how to run parallelize custom algorithm using Dask Delayed option

For more information on dask over kubernetes: https://kubernetes.dask.org/en/latest/

Set up the environment

```
# set mlrun api path and artifact path for logging
import mlrun
project_name = "dask-demo"
mlrun.set_environment(project=project_name, artifact_path = './')
```

```
('dask-demo', '/User/dask')
```

Create and Start Dask Cluster

Dask functions can be local (local workers), or remote (use containers in the cluster), in the case of remote users can specify the number of replica (optional) or leave blank for auto-scale. We use `new_function()` to define our Dask cluster and set the desired configuration of that clustered function.

if the dask workers need to access the shared file system we apply a shared volume mount (e.g. via v3io mount).

Dask function spec have several unique attributes (in addition to the standard job attributes):

- **.remote** - bool, use local or clustered dask
- **.replicas** - number of desired replicas, keep 0 for auto-scale
- **.min_replicas**, **.max_replicas** - set replicas range for auto-scale
- **.scheduler_timeout** - cluster will be killed after timeout (inactivity), default is '60 minutes'
- **.nthreads** - number of worker threads

If you want to access the dask dashboard or scheduler from remote you need to use NodePort service type (set `.service_type` to 'NodePort'), and the external IP need to be specified in mlrun configuration (mlconf.remote_host), this will be set automatically if you are running on an Iguazio cluster.

We specify the kind (dask) and the container image

```
# create an mlrun function which will init the dask cluster
dask_cluster_name = "dask-cluster"
dask_cluster = mlrun.new_function(dask_cluster_name, kind='dask', image='mlrun/ml-models
↪')
dask_cluster.apply(mlrun.mount_v3io())
```

```
<mlrun.runtimes.daskjob.DaskCluster at 0x7f7dbe4166d0>
```

```
# set range for # of replicas with replicas and max_replicas
dask_cluster.spec.min_replicas = 1
dask_cluster.spec.max_replicas = 4

# set the use of dask remote cluster (distributed)
dask_cluster.spec.remote = True
dask_cluster.spec.service_type = "NodePort"

# set dask memory and cpu limits
dask_cluster.with_requests(mem='2G', cpu='2')
```

Initialize the Dask Cluster

When we request the dask cluster `client` attribute it will verify the cluster is up and running

```
# init dask client and use the scheduler address as param in the following cell
dask_cluster.client
```

```
> 2021-01-24 23:48:54,057 [info] trying dask client at: tcp://mlrun-dask-cluster-
↪b3c6e737-3.default-tenant:8786
> 2021-01-24 23:48:54,067 [info] using remote dask scheduler (mlrun-dask-cluster-
↪b3c6e737-3) at: tcp://mlrun-dask-cluster-b3c6e737-3.default-tenant:8786
```

```
/User/.pythonlibs/jupyter/lib/python3.7/site-packages/distributed/client.py:1129:␣
↪VersionMismatchWarning: Mismatched versions found

+---------+--------+-----------+---------+
| Package | client | scheduler | workers |
+---------+--------+-----------+---------+
| blosc   | 1.7.0  | 1.10.2    | 1.10.2  |
| lz4     | 3.1.0  | 3.1.3     | 3.1.3   |
| msgpack | 1.0.0  | 1.0.2     | 1.0.2   |
| numpy   | 1.19.2 | 1.18.1    | 1.18.1  |
| toolz   | 0.11.1 | 0.10.0    | 0.10.0  |
| tornado | 6.0.4  | 6.0.3     | 6.0.3   |
+---------+--------+-----------+---------+
Notes:
-  msgpack: Variation is ok, as long as everything is above 0.6
  warnings.warn(version_module.VersionMismatchWarning(msg[0]["warning"]))
```

```
<IPython.core.display.HTML object>
```

```
<Client: 'tcp://10.200.0.51:8786' processes=1 threads=1, memory=4.12 GB>
```

Creating A Function Which Run Over Dask

```
# mlrun: start-code
```

Import mlrun and dask. nuclio is used just to convert the code into an mlrun function

```
import mlrun
```

```
%nuclio config kind = "job"
%nuclio config spec.image = "mlrun/ml-models"
```

```
%nuclio: setting kind to 'job'
%nuclio: setting spec.image to 'mlrun/ml-models'
```

```
from dask.distributed import Client
from dask import delayed
from dask import dataframe as dd

import warnings
import numpy as np
import os
import mlrun

warnings.filterwarnings("ignore")
```

python function code

This simple function reads a csv file using dask dataframe and run group by and describe function on the dataset and store the results as a dataset artifact

```
def test_dask(context,
              dataset: mlrun.DataItem,
              client=None,
              dask_function: str=None) -> None:

    # setup dask client from the MLRun dask cluster function
    if dask_function:
        client = mlrun.import_function(dask_function).client
    elif not client:
        client = Client()

    # load the dataitem as dask dataframe (dd)
    df = dataset.as_df(df_module=dd)

    # run describe (get statistics for the dataframe) with dask
```

(continues on next page)

(continued from previous page)

```
    df_describe = df.describe().compute()

    # run groupby and count using dask
    df_grpby = df.groupby("VendorID").count().compute()

    context.log_dataset("describe",
                        df=df_grpby,
                        format='csv', index=True)
    return
```

```
# mlrun: end-code
```

Test Our Function Over Dask

Load sample data

```
DATA_URL="/User/examples/ytrip.csv"
```

```
!mkdir -p /User/examples/
!curl -L "https://s3.wasabisys.com/iguazio/data/Taxi/yellow_tripdata_2019-01_subset.csv"␣
↪> {DATA_URL}
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 84.9M  100 84.9M    0     0  17.3M      0  0:00:04  0:00:04 --:--:-- 19.1M
```

Convert the code to MLRun function

Use code_to_function to convert the code to MLRun and specify the configuration for the dask process (e.g. replicas, memory etc.) Note that the resource configurations are per worker

```
# mlrun will transform the code above (up to nuclio: end-code cell) into serverless␣
↪function
# which will run in k8s pods
fn = mlrun.code_to_function("test_dask",  kind='job', handler="test_dask").apply(mlrun.
↪mount_v3io())
```

Run the function

When running the function you would see a link as part of the result. click on this link takes you to the dask monitoring dashboard

```
# function URI is db://<project>/<name>
dask_uri = f'db://{project_name}/{dask_cluster_name}'
```

```
r = fn.run(handler = test_dask,
           inputs={"dataset": DATA_URL},
           params={"dask_function": dask_uri})
```

```
> 2021-01-24 23:49:37,858 [info] starting run test-dask-test_dask␣
↪uid=6410ec27b63e4a12b025696fcabc2dc9 DB=http://mlrun-api:8080
> 2021-01-24 23:49:38,069 [info] Job is running in the background, pod: test-dask-test-
↪dask-rmgkn
> 2021-01-24 23:49:41,647 [warning] Unable to parse server or client version. Assuming␣
↪compatible: {'server_version': 'unstable', 'client_version': 'unstable'}
> 2021-01-24 23:49:42,112 [info] using in-cluster config.
> 2021-01-24 23:49:42,113 [info] trying dask client at: tcp://mlrun-dask-cluster-
↪b3c6e737-3.default-tenant:8786
> 2021-01-24 23:49:42,134 [info] using remote dask scheduler (mlrun-dask-cluster-
↪b3c6e737-3) at: tcp://mlrun-dask-cluster-b3c6e737-3.default-tenant:8786
remote dashboard: default-tenant.app.yh57.iguazio-cd0.com:30433
> 2021-01-24 23:49:48,334 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```
to track results use .show() or .logs() or in CLI:
!mlrun get run 6410ec27b63e4a12b025696fcabc2dc9 --project dask-demo , !mlrun logs␣
↪6410ec27b63e4a12b025696fcabc2dc9 --project dask-demo
> 2021-01-24 23:49:50,284 [info] run executed, status=completed
```

Track the progress in the UI

Users can view the progress and detailed information in the mlrun UI by clicking on the uid above. Also, to track the dask progress in the dask UI click on the "dashboard link" above the "client" section

Pipelines using Dask, Kubeflow and MLRun

Create a project to host functions, jobs and artifacts

Projects are used to package multiple functions, workflows, and artifacts. Project code and definitions are usually stored in a Git archive.

The following code creates a new project in a local dir and initializes git tracking on it.

```
import os
import mlrun
import warnings
warnings.filterwarnings("ignore")

# set project name and dir
project_name = 'sk-project-dask'
project_dir = './'

# specify artifacts target location
```

(continues on next page)

(continued from previous page)

```
_, artifact_path = mlrun.set_environment(project=project_name)

# set project
sk_dask_proj = mlrun.get_or_create_project(project_name, project_dir, init_git=True)
```

```
> 2022-09-27 17:26:14,808 [info] loaded project sk-project-dask from MLRun DB
> 2022-09-27 17:26:14,839 [info] loaded project sk-project-dask from MLRun DB
```

Init Dask cluster

```
import mlrun
# set up function from local file
dsf = mlrun.new_function(name="mydask", kind="dask", image="mlrun/ml-models")

# set up function specs for dask
dsf.spec.remote = True
dsf.spec.replicas = 5
dsf.spec.service_type = 'NodePort'
dsf.with_limits(mem="6G")
dsf.spec.nthreads = 5
```

```
# apply mount_v3io over our function so that our k8s pod which run our function
# will be able to access our data (shared data access)
dsf.apply(mlrun.mount_v3io())
```

```
<mlrun.runtimes.daskjob.DaskCluster at 0x7f47fce9c850>
```

```
dsf.save()
```

```
'db://sk-project-dask/mydask'
```

```
# init dask cluster
dsf.client
```

```
> 2022-09-27 17:26:25,134 [info] trying dask client at: tcp://mlrun-mydask-d7df9301-d.
↪default-tenant:8786
> 2022-09-27 17:26:25,162 [info] using remote dask scheduler (mlrun-mydask-d7df9301-d)␣
↪at: tcp://mlrun-mydask-d7df9301-d.default-tenant:8786
```

```
<IPython.core.display.HTML object>
```

```
<Client: 'tcp://10.200.152.178:8786' processes=0 threads=0, memory=0 B>
```

Load and run a functions

Load the function object from .py .yaml file or Function Hub (marketplace).

```
# load function from the Function Hub
sk_dask_proj.set_function("hub://describe", name="describe")
sk_dask_proj.set_function("hub://sklearn_classifier_dask", name="dask_classifier")
```

```
<mlrun.runtimes.kubejob.KubejobRuntime at 0x7f48353d5130>
```

Create a fully automated ML pipeline

Add more functions to the project to be used in the pipeline (from the Function Hub)

Describe data, train and eval model with dask.

Define and save a pipeline

The following workflow definition will be written into a file. It describes a Kubeflow execution graph (DAG) and how functions and data are connected to form an end to end pipeline.

- Describe data.
- Train, test and evaluate with dask.

Check the code below to see how functions objects are initialized and used (by name) inside the workflow. The `workflow.py` file has two parts, initialize the function objects and define pipeline dsl (connect the function inputs and outputs).

> Note: The pipeline can include CI steps like building container images and deploying models as illustrated in the following example.

```
%%writefile workflow.py
import os
from kfp import dsl
import mlrun

# params
funcs = {}
LABELS = "label"
DROP = "congestion_surcharge"
DATA_URL = mlrun.get_sample_path("data/iris/iris_dataset.csv")
DASK_CLIENT = "db://sk-project-dask/mydask"


# init functions is used to configure function resources and local settings
def init_functions(functions: dict, project=None, secrets=None):
    for f in functions.values():
```

(continues on next page)

(continued from previous page)

```
        f.apply(mlrun.mount_v3io())
        pass


@dsl.pipeline(name="Demo training pipeline", description="Shows how to use mlrun")
def kfpipeline():
    # Describe the data
    describe = funcs["describe"].as_step(
        inputs={"table": DATA_URL},
        params={"dask_function": DASK_CLIENT},
    )

    # Train, test and evaluate:
    train = funcs["dask_classifier"].as_step(
        name="train",
        handler="train_model",
        inputs={"dataset": DATA_URL},
        params={
            "label_column": LABELS,
            "dask_function": DASK_CLIENT,
            "test_size": 0.10,
            "model_pkg_class": "sklearn.ensemble.RandomForestClassifier",
            "drop_cols": DROP,
        },
        outputs=["model", "test_set"],
    )
    train.after(describe)
```

```
Overwriting workflow.py
```

```
# register the workflow file as "main", embed the workflow code into the project YAML
sk_dask_proj.set_workflow('main', 'workflow.py', embed=False)
```

Save the project definitions to a file (project.yaml). It is recommended to commit all changes to a Git repo.

```
sk_dask_proj.save()
```

```
<mlrun.projects.project.MlrunProject at 0x7f48342e4880>
```

Run a pipeline workflow

Use the `run` method to execute a workflow. You can provide alternative arguments and specify the default target for workflow artifacts. The workflow ID is returned and can be used to track the progress or you can use the hyperlinks.

> Note: The same command can be issued through CLI commands: `mlrun project my-proj/ -r main -p "v3io:///users/admin/mlrun/kfp/{{workflow.uid}}/"`

The `dirty` flag lets you run a project with uncommitted changes (when the notebook is in the same git dir it is always dirty) The `watch` flag waits for the pipeline to complete and print results.

```
artifact_path = os.path.abspath('./pipe/{{workflow.uid}}')
run_id = sk_dask_proj.run(
    'main',
    arguments={},
    artifact_path=artifact_path,
    dirty=False,
    watch=True
)
```

```
<IPython.core.display.HTML object>
```

```
<graphviz.graphs.Digraph at 0x7f47fce02a90>
```

```
<IPython.core.display.HTML object>
```

back to top

6.2.2 MPIJob and Horovod runtime

Running distributed workloads

Training a Deep Neural Network is a hard task. With growing datasets, wider and deeper networks, training our Neural Network can require a lot of resources (CPUs / GPUs / Mem and Time).

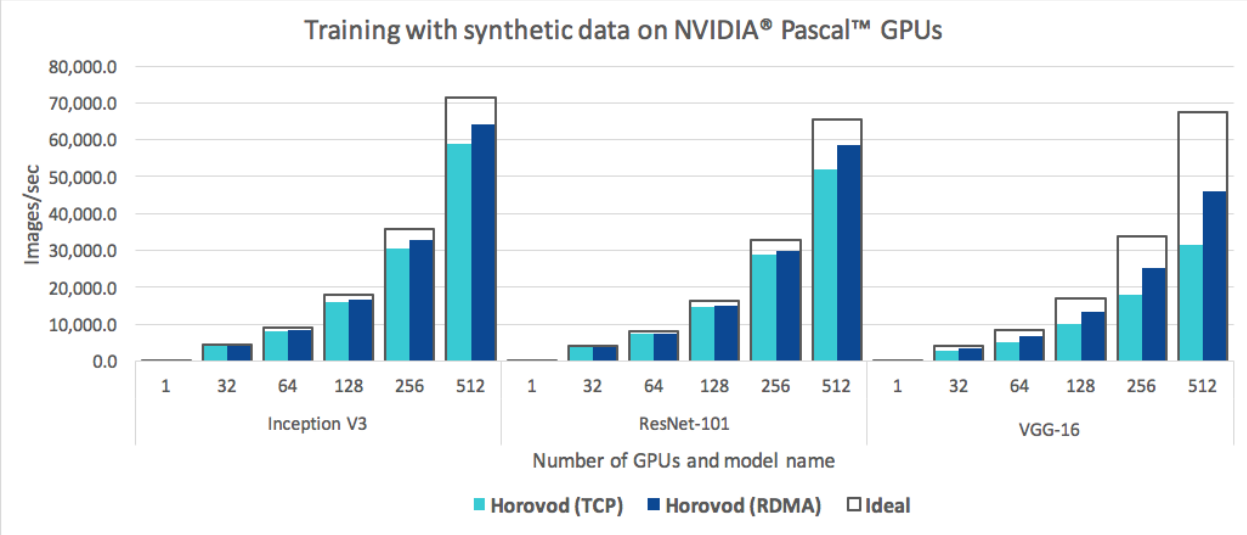
There are two main reasons why we would like to distribute our Deep Learning workloads:

1. **Model Parallelism** — The **Model** is too big to fit a single GPU.
   In this case the model contains too many parameters to hold within a single GPU.
   To negate this we can use strategies like **Parameter Server** or slicing the model into slices of consecutive layers which we can fit in a single GPU.
   Both strategies require **Synchronization** between the layers held on different GPUs / Parameter Server shards.
2. **Data Parallelism** — The **Dataset** is too big to fit a single GPU.
   Using methods like **Stochastic Gradient Descent** we can send batches of data to our models for gradient estimation. This comes at the cost of longer time to converge since the estimated gradient may not fully represent the actual gradient.
   To increase the likelihood of estimating the actual gradient we could use bigger batches, by sending small batches to different GPUs running the same Neural Network, calculating the batch gradient and then running a **Synchronization Step** to calculate the average gradient over the batches and update the Neural Networks running on the different GPUs.

   It is important to understand that the act of distribution adds extra **Synchronization Costs** which may vary according to your cluster's configuration.

   As the gradients and NN needs to be propagated to each GPU in the cluster every epoch (or a number of steps), Networking can become a bottleneck and sometimes different configurations need to be used for optimal performance.

   Scaling Efficiency is the metric used to show by how much each additional GPU should benefit the training process with Horovod showing up to 90% (When running with a well written code and good parameters).



How can we distribute our training?

There are two different cluster configurations (which can be combined) we need to take into account.

- **Multi Node** — GPUs are distributed over multiple nodes in the cluster.
- **Multi GPU** — GPUs are within a single Node.

In this demo we show a **Multi Node Multi GPU** — **Data Parallel** enabled training using Horovod.
However, you should always try and use the best distribution strategy for your use case (due to the added costs of the distribution itself, ability to run in an optimized way on specific hardware or other considerations that may arise).

How Horovod works?

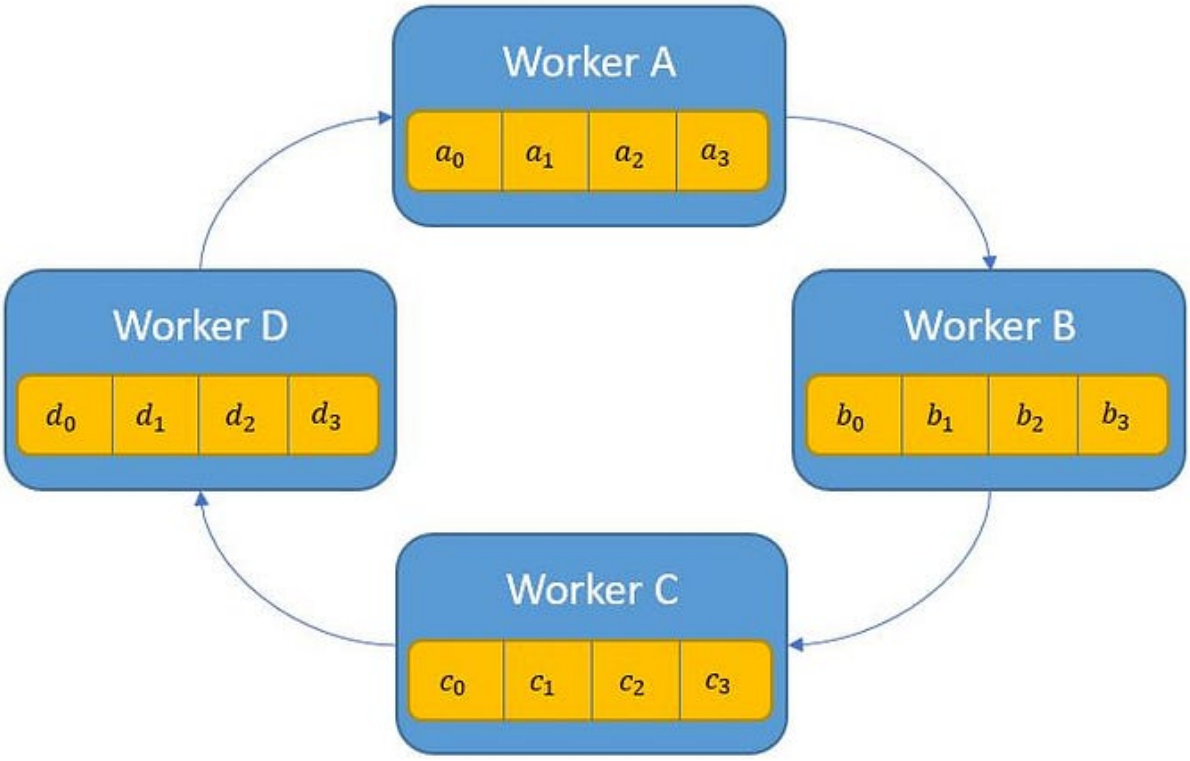
Horovod's primary motivation is to make it easy to take a single-GPU training script and successfully scale it to train across many GPUs in parallel. This has two aspects:

- How much modification does one have to make to a program to make it distributed, and how easy is it to run it?
- How much faster would it run in distributed mode?

Horovod Supports TensorFlow, Keras, PyTorch, and Apache MXNet.

in MLRun we use Horovod with MPI in order to create cluster resources and allow for optimized networking.
Note: Horovod and MPI may use NCCL when applicable which may require some specific configuration arguments to run optimally.

Horovod uses this MPI and NCCL concepts for distributed computation and messaging to quickly and easily synchronize between the different nodes or GPUs.



Horovod will run your code on all the given nodes (Specific node can be addressed via `hvd.rank()`) while using an `hvd.DistributedOptimizer` wrapper to run the **synchronization cycles** between the copies of your Neural Network running at each node.

Note: Since all the copies of your Neural Network must be the same, Your workers will adjust themselves to the rate of the slowest worker (simply by waiting for it to finish the epoch and receive its updates). Thus try not to make a specific worker do a lot of additional work on each epoch (Like a lot of saving, extra calculations, etc...) since this can affect the overall training time.

How do we integrate TF2 with Horovod?

As it's one of the main motivations, integration is fairly easy and requires only a few steps: (You can read the full instructions for all the different frameworks on Horovod's documentation website).

1. Run `hvd.init()`.
2. Pin each GPU to a single process. With the typical setup of one GPU per process, set this to local rank. The first process on the server will be allocated the first GPU, the second process will be allocated the second GPU, and so forth.

```
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')
```

3. Scale the learning rate by the number of workers.
   Effective batch size in synchronous distributed training is scaled by the number of workers. An increase in learning rate compensates for the increased batch size.

4. Wrap the optimizer in `hvd.DistributedOptimizer`.
   The distributed optimizer delegates gradient computation to the original optimizer, averages gradients using allreduce or allgather, and then applies those averaged gradients.
   For TensorFlow v2, when using a `tf.GradientTape`, wrap the tape in `hvd.DistributedGradientTape` instead of wrapping the optimizer.
5. Broadcast the initial variable states from rank 0 to all other processes.
   This is necessary to ensure consistent initialization of all workers when training is started with random weights or restored from a checkpoint.
   For TensorFlow v2, use `hvd.broadcast_variables` after models and optimizers have been initialized.
6. Modify your code to save checkpoints only on worker 0 to prevent other workers from corrupting them.
   For TensorFlow v2, construct a `tf.train.Checkpoint` and only call `checkpoint.save()` when `hvd.rank() == 0`.

You can go to Horovod's Documentation to read more about horovod.

Image classification use case

See the end to end **Image Classification with Distributed Training Demo**

6.2.3 Spark Operator runtime

Using Spark Operator for running Spark jobs over k8s.

The `spark-on-k8s-operator` allows Spark applications to be defined in a declarative manner and supports one-time Spark applications with `SparkApplication` and cron-scheduled applications with `ScheduledSparkApplication`.

When sending a request with MLRun to the Spark operator, the request contains your full application configuration including the code and dependencies to run (packaged as a docker image or specified via URIs), the infrastructure parameters, (e.g. the memory, CPU, and storage volume specs to allocate to each Spark executor), and the Spark configuration.

Kubernetes takes this request and starts the Spark driver in a Kubernetes pod (a k8s abstraction, just a docker container in this case). The Spark driver then communicates directly with the Kubernetes master to request executor pods, scaling them up and down at runtime according to the load if dynamic allocation is enabled. Kubernetes takes care of the bin-packing of the pods onto Kubernetes nodes (the physical VMs), and dynamically scales the various node pools to meet the requirements.

When using Spark operator the resources are allocated per task, meaning that it scales down to zero when the task is done.

```
import mlrun
import os

# set up new spark function with spark operator
# command will use our spark code which needs to be located on our file system
# the name param can have only non capital letters (k8s convention)
read_csv_filepath = os.path.join(os.path.abspath('.'), 'spark_read_csv.py')
sj = mlrun.new_function(kind='spark', command=read_csv_filepath, name='sparkreadcsv')

# set spark driver config (gpu_type & gpus=<number_of_gpus>  supported too)
sj.with_driver_limits(cpu="1300m")
sj.with_driver_requests(cpu=1, mem="512m")
```

(continues on next page)

(continued from previous page)

```
# set spark executor config (gpu_type & gpus=<number_of_gpus> are supported too)
sj.with_executor_limits(cpu="1400m")
sj.with_executor_requests(cpu=1, mem="512m")

# adds fuse, daemon & iguazio's jars support
sj.with_igz_spark()

# Alternately, move volume_mounts to driver and executor-specific fields and leave
# v3io mounts out of executor mounts if mount_v3io_to_executor=False
# sj.with_igz_spark(mount_v3io_to_executor=False)

# set spark driver volume mount
# sj.function.with_driver_host_path_volume("/host/path", "/mount/path")

# set spark executor volume mount
# sj.function.with_executor_host_path_volume("/host/path", "/mount/path")

# confs are also supported
sj.spec.spark_conf['spark.eventLog.enabled'] = True

# add python module
sj.spec.build.commands = ['pip install matplotlib']

# Number of executors
sj.spec.replicas = 2
```

```
# Rebuilds the image with MLRun - needed in order to support artifactlogging etc
sj.deploy()
```

```
# Run task while setting the artifact path on which our run artifact (in any) will be␣
↪saved
sj.run(artifact_path='/User')
```

Spark Code (spark_read_csv.py)

```
from pyspark.sql import SparkSession
from mlrun import get_or_create_ctx

context = get_or_create_ctx("spark-function")

# build spark session
spark = SparkSession.builder.appName("Spark job").getOrCreate()

# read csv
df = spark.read.load('iris.csv', format="csv",
                     sep=",", header="true")

# sample for logging
df_to_log = df.describe().toPandas()
```

(continues on next page)

(continued from previous page)

```
# log final report
context.log_dataset("df_sample",
                    df=df_to_log,
                    format="csv")
spark.stop()
```

6.2.4 Nuclio real-time functions

Nuclio is a high-performance "serverless" framework focused on data, I/O, and compute intensive workloads. It is well integrated with popular data science tools, such as Jupyter and Kubeflow; supports a variety of data and streaming sources; and supports execution over CPUs and GPUs.

You can use Nuclio through a fully managed application service (in the cloud or on-prem) in the Iguazio MLOps Platform. MLRun serving utilizes serverless Nuclio functions to create multi-stage real-time pipelines.

The underlying Nuclio serverless engine uses a high-performance parallel processing engine that maximizes the utilization of CPUs and GPUs, supports 13 protocols and invocation methods (for example, HTTP, Cron, Kafka, Kinesis), and includes dynamic auto-scaling for HTTP and streaming. Nuclio and MLRun support the full life cycle, including auto-generation of micro-services, APIs, load-balancing, logging, monitoring, and configuration management—such that developers can focus on code, and deploy to production faster with minimal work.

Nuclio is extremely fast: a single function instance can process hundreds of thousands of HTTP requests or data records per second. To learn more about how Nuclio works, see the Nuclio architecture documentation.

Nuclio is secure: Nuclio is integrated with Kaniko to allow a secure and production-ready way of building Docker images at run time.

Read more in the Nuclio documentation and the open-source MLRun library.

Why another "serverless" project?

None of the existing cloud and open-source serverless solutions addressed all the desired capabilities of a serverless framework:

- Real-time processing with minimal CPU/GPU and I/O overhead and maximum parallelism
- Native integration with a large variety of data sources, triggers, processing models, and ML frameworks
- Stateful functions with data-path acceleration
- Simple debugging, regression testing, and multi-versioned CI/CD pipelines
- Portability across low-power devices, laptops, edge and on-prem clusters, and public clouds
- Open-source but designed for the enterprise (including logging, monitoring, security, and usability)

Nuclio was created to fulfill these requirements. It was intentionally designed as an extendable open-source framework, using a modular and layered approach that supports constant addition of triggers and data sources, with the hope that many will join the effort of developing new modules, developer tools, and platforms for Nuclio.

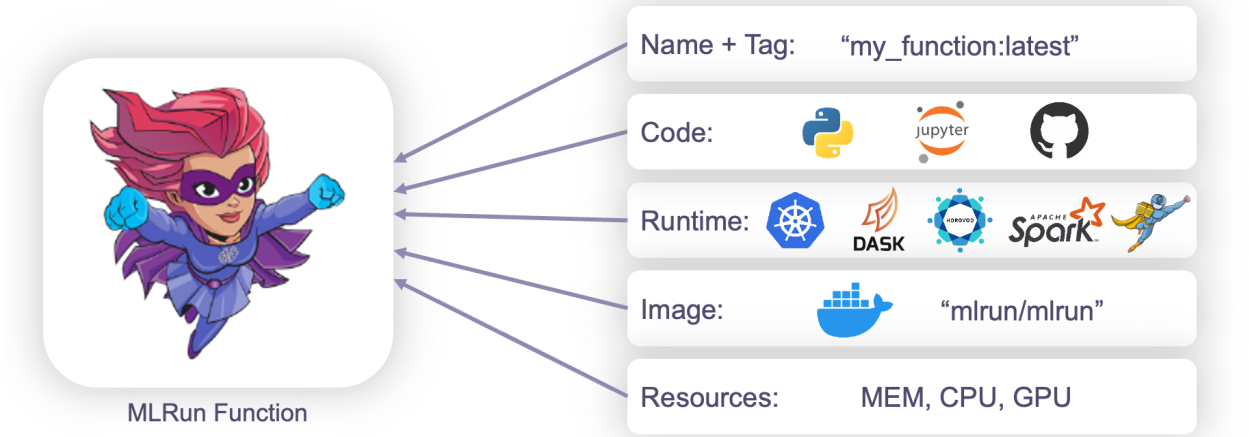
6.3 Create and use functions

Functions are the basic building blocks of MLRun. They are essentially Python objects that know how to run locally or on a Kubernetes cluster. This section covers how to create and customize an MLRun function, as well as common parameters across all functions.

In this section:

6.3.1 Functions overview

MLRun functions are used to run jobs, deploy models, create pipelines, and more. There are various kinds of MLRun functions with different capabilities, however, there are commonalities across all functions. In general, an MLRun function looks like the following:

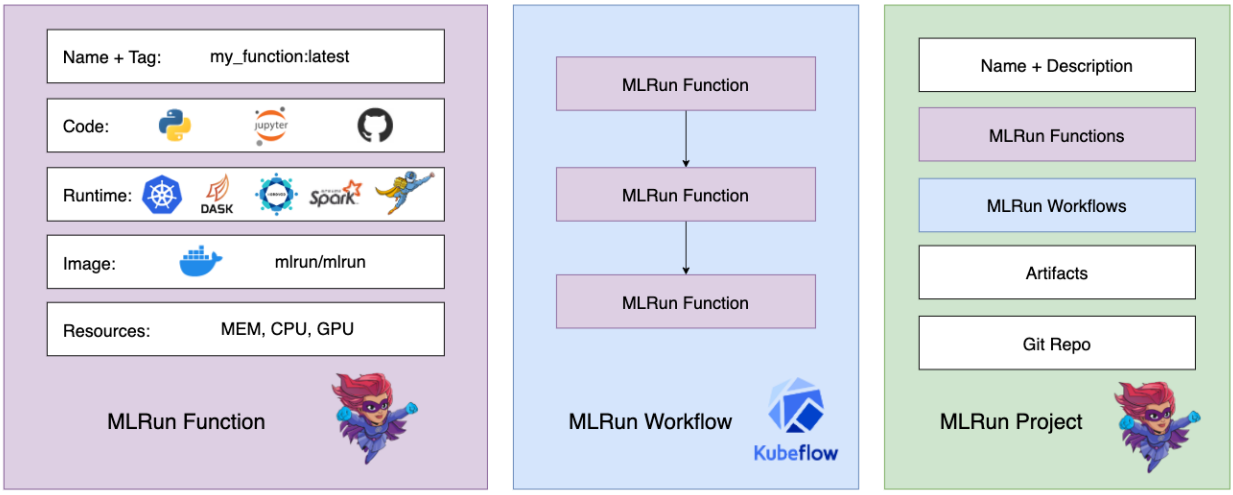


You can read more about MLRun Functions **here**. Each parameter and capability is explained in more detail in the following sections ***Creating functions*** and ***Customizing functions***.

6.3.2 Functions and projects

Function are members of an **MLRun project**. Once you register a function within a project, you can execute it in your local environment or at scale on a Kubernetes cluster.

The relationship between **functions**, **workflows**, and **projects** is as follows:



After the MLRun functions and workflows are created and **registered into the project**, they are invoked using the project object. This workflow pairs especially well with Git and **CI/CD** integration.

6.3.3 Creating functions

The recommended way to create an MLRun function is by using an MLRun project (see **create and use projects**). The general flow looks like the following:

```
project = mlrun.get_or_create_project(...)

fn = project.set_function(...)
```

When creating a function, there are 3 main scenarios:

1. ***Single source file*** — when your code can be contained in a single file
2. ***Multiple source files*** — when your code requires additional files or dependencies
3. ***Import existing function*** — when your function already exists elsewhere and you just want to import it

Note

Using the `set_function` method of an MLRun project allows for each of these scenarios in a transparent way. Depending on the source passed in, the project registers the function using some lower level functions. For specific use cases, you also have access to the lower level functions *`new_function()`*, *`code_to_function()`*, and *`import_function()`*.

Using set_function

The MLRun project object has a method called *set_function()*, which is a one-size-fits-all way of creating an MLRun function. This method accepts a variety of sources including Python files, Jupyter Notebooks, Git repos, and more.

Note

The return value of `set_function` is your MLRun function. You can immediately run it or apply additional configurations like resources, scaling, etc. See ***Customizing functions*** for more details.

When using `set_function` there are a number of common parameters across all function types and creation scenarios. Consider the following example:

```
fn = project.set_function(
    name="my-function", tag="latest", func="my_function.py",
    image="mlrun/mlrun", kind="job", handler="train_model",
    requirements=["pandas==1.3.5"], with_repo=True
)
```

- **name:** Name of your MLRun function within the given project. This is displayed in the MLRun UI, as well as the Kubernetes pod.
- **tag:** Tag for your function (much like a Docker image). Omitting this parameter defaults to `latest`.
- **func:** What to run with the MLRun function. This can be a number of things including files (`.py`, `.ipynb`, `.yaml`, etc.), URIs (`hub://` prefixed Function Hub URI, `db://` prefixed MLRun DB URI), existing MLRun function objects, or `None` (for current `.ipynb` file).
- **image:** Docker image to use when containerizing the piece of code. If you also specify the `requirements` parameter to build a new Docker image, the `image` parameter is used as the base image.
- **kind:** Runtime the MLRun function uses. See **Kinds of functions (runtimes)** forthe list of supported batch and real-time runtimes.
- **handler:** Default function handler to invoke (e.g. a Python function within your script). This handler can also be overriden when executing the function.
- **requirements:** Additional Python dependencies needed for the function to run. Using this parameter results in a new Docker image (using the `image` parameter as a base image). This can be a list of Python dependencies or a path to a `requirements.txt` file.
- **with_repo:** Whether a function requires additional files or dependencies within a Git repo or archive file. This Git repo or archive file is specified on a project level via `project.set_source(...)`, which the function consumes. If this parameter is omitted, the default is `False`.

Building images

If your MLRun function requires additional libraries or files, you might need to build a new Docker image. You can do this by specifying a base image to use as the `image`, your requirements via `requirements`, and (optionally) your source code via `with_repo=True` (where the source is specified by `project.set_source(...)`). See **Build function image** for more information on when a build is required.

Note

When using `with_repo`, the contents of the Git repo or archive are available in the current working directory of your MLRun function during runtime.

A good place to start is one of the default MLRun images:

- `mlrun/mlrun`: Suits most lightweight components (includes `sklearn`, `pandas`, `numpy` and more)
- `mlrun/ml-models`: Suits most CPU ML/DL workloads (includes `Tensorflow`, `Keras`, `PyTorch` and more)
- `mlrun/ml-models-gpu`: Suits most GPU ML/DL workloads (includes GPU `Tensorflow`, `Keras`, `PyTorch` and more)

Dockerfiles for the MLRun images can be found **here**.

Single source file

The simplest way to create a function is to use a single file as the source. The code itself is embedded into the MLRun function object. This makes the function quite portable since it does not depend on any external files. You can use any source file supported by MLRun such as Python or Jupyter notebook.

Note

MLRun is not limited to Python. Files of type Bash, Go, etc. are also supported.

Python

This is the simplest way to create a function out of a given piece of code. Simply pass in the path to the Python file *relative to your project context directory*.

```
fn = project.set_function(
    name="python", func="job.py",  kind="job",
    image="mlrun/mlrun", handler="handler"
)
```

Jupyter Notebook

This is a great way to create a function out of a Jupyter Notebook. Just pass in the path to the Jupyter Notebook *relative to your project context directory*. You can use **MLRun cell tags** to specify which parts of the notebook should be included in the function.

Note

To ensure that the latest changes are included, make sure you save your notebook before creating/updating the function.

```
fn = project.set_function(
    name="notebook", func="nb.ipynb", kind="serving",
    image="mlrun/ml-models", requirements=["pandas==1.3.5"]
)
```

You can also create an MLRun function out of the current Jupyter Notebook you are running in. To do this, simply ommit the `func` parameter in `set_function`.

Multiple source files

If your code requires additional files or external libraries, you need to use a source that supports multiple files such as Git, an archive (zip/tar/etc.), or V3IO file share. This approach (especially using a Git repo) pairs well with MLRun projects.

To do this, you must:

- Provide `with_repo=True` when creating your function via `project.set_function(...)`
- Set project source via `project.set_source(source=...)`

This instructs MLRun to load source code from the git repo/archive/file share associated with the project. There are two ways to load these additional files:

Load code from container

The function is built once. *This is the preferred approach for production workloads*. For example:

```
project.set_source(source="git://github.com/mlrun/project-archive.git")

fn = project.set_function(
    name="myjob", handler="job_func.job_handler",
    image="mlrun/mlrun", kind="job", with_repo=True,
)

project.build_function(fn)
```

Load code at runtime

The function pulls the source code at runtime. *This is a simpler approach during development that allows for making code changes without re-building the image each time*. For example:

```
archive_url = "https://s3.us-east-1.wasabisys.com/iguazio/project-archive/project-
↪archive.zip"
project.set_source(source=archive_url, pull_at_runtime=True)

fn = project.set_function(
    name="nuclio", handler="nuclio_func:nuclio_handler",
    image="mlrun/mlrun", kind="nuclio", with_repo=True,
)
```

Import or use an existing function

If you already have an MLRun function that you want to import, you can do so from multiple locations such as YAML, Function Hub, and MLRun DB.

YAML

MLRun functions can be exported to YAML files via `fn.export()`. These YAML files can then be imported via the following:

```
fn = project.set_function(name="import", func="function.yaml")
```

Function Hub

Functions can also be imported from the **MLRun Function Hub**: simply import using the name of the function and the `hub://` prefix:

Note

By default, the `hub://` prefix points to the official Function Hub. You can, however, also substitute your own repo to create your own hub.

```
fn = project.set_function(name="describe", func="hub://describe")
```

MLRun DB

You can also import functions directly from the MLRun DB. These could be functions that have not been pushed to a git repo, archive, or Function Hub. Import via the name of the function and the `db://` prefix:

```
fn = project.set_function(name="db", func="db://import")
```

MLRun function

You can also directly use an existing MLRun function object. This is usually used when more granular control over function parameters is required (e.g. advanced parameters that are not supported by *set_function()*).

This example uses a *real-time serving pipeline (graph)*.

```
fn = mlrun.new_function("serving", kind="serving", image="mlrun/mlrun")
graph = serving.set_topology("flow")
graph.to(name="double", handler="mylib.double") \
     .to(name="add3", handler="mylib.add3") \
     .to(name="echo", handler="mylib.echo").respond()

project.set_function(name="serving", func=fn, with_repo=True)
```

6.3.4 Customizing functions

Once you have created your MLRun function, there are many customizations you can add. Some potential customizations include:

Environment variables

Environment variables can be added individually, from a Python dictionary, or a file:

```
# Single variable
fn.set_env(name="MY_ENV", value="MY_VAL")

# Multiple variables
fn.set_envs(env_vars={"MY_ENV" : "MY_VAL", "SECOND_ENV" : "SECOND_VAL"})

# Multiple variables from file
fn.set_envs(file_path="env.txt")
```

Memory, CPU, GPU resources

Adding requests and limits to your function specify what compute resources are required. It is best practice to define this for each MLRun function. See **CPU, GPU, and memory limits for user jobs** for more information on configuring resources.

```
# Requests - lower bound
fn.with_requests(mem="1G", cpu=1)

# Limits - upper bound
fn.with_limits(mem="2G", cpu=2, gpus=1)
```

Scaling and auto-scaling

Scaling behavior can be added to real-time and distributed runtimes including `nuclio`, `serving`, `spark`, `dask`, and `mpijob`. See **Replicas** to see how to configure scaling behavior per runtime. This example demonstrates setting replicas for `nuclio`/`serving` runtimes:

```
# Nuclio/serving scaling
fn.spec.replicas = 2
fn.spec.min_replicas = 1
fn.spec.min_replicas = 4
```

Mount persistent storage

In some instances, you might need to mount a file-system to your container to persist data. This can be done with native K8s PVC's or the V3IO data layer for Iguazio clusters. See **Attach storage to functions** for more information on the storage options.

```
# Mount persistent storage - V3IO
fn.apply(mlrun.mount_v3io())

# Mount persistent storage - PVC
fn.apply(mlrun.platforms.mount_pvc(pvc_name="data-claim", volume_name="data", volume_
↪mount_path="/data"))
```

Node selection

Node selection can be used to specify where to run workloads (e.g. specific node groups, instance types, etc.). This is a more advanced parameter mainly used in production deployments to isolate platform services from workloads. See **Node affinity** for more information on how to configure node selection.

```
# Only run on non-spot instances
fn.with_node_selection(node_selector={"app.iguazio.com/lifecycle" : "non-preemptible"})
```

6.4 Converting notebooks to function

MLRun annotations are used to identify the code that needs to be converted into an MLRun function. They provide non-intrusive hints that indicate which parts of your notebook should be considered as the code of the function.

Annotations start a code block using `# mlrun: start-code` and end a code block(s), with `# mlrun: end-code`. Use the `#mlrun: ignore` to exclude items from the code qualified annotations. Make sure that the annotations include anything required for the function to run.

```
# mlrun: start-code

def sub_handler():
    return "hello world"
```

The `# mlrun: ignore` annotation enables you to exclude the cell from the function code.

```
# mlrun: ignore

# the handler in the code section below will not call this sub_handler
def sub_handler():
    return "I will be ignored!"
```

```
def handler(context, event):
    return sub_handler()

# mlrun: end-code
```

Convert the function with `mlrun.code_to_function` and run the handler. Notice the returned value under `results`.

Note

Make sure to save the notebook before running `mlrun.code_to_function` so that the lateset changes will be reflected in the function.

```
from mlrun import code_to_function

some_function = code_to_function('some-function-name', kind='job', code_output='.')
some_function.run(name='some-function-name', handler='handler', local=True)
```

```
> 2021-11-01 07:42:44,930 [info] starting run some-function-name␣
↪uid=742e7d6e930c48f3a2f1d6175e971455 DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-11-01 07:42:45,214 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f3fc9ed81d0>
```

In this section

6.4.1 Named annotations

The `# mlrun: start-code` and `# mlrun: end-code` annotations can be used to convert different code sections to different MLRun, functions in the same notebook. To do so add the name of the MLRun function to the end of the annotation as shown in the example below.

```
# mlrun: start-code my-function-name

def handler(context, event):
    return "hello from my-function"

# mlrun: end-code my-function-name
```

Convert the function and run the handler. Notice that the handler that is being used and that there is a change in the returned value under `results`.

```
my_function = code_to_function('my-function-name', kind='job')
my_function.run(name='my-function-name', handler='handler', local=True)
```

```
> 2021-11-01 07:42:53,892 [info] starting run my-function-name␣
↪uid=e4bbc3cae21042439cc1c3cb9631751c DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-11-01 07:42:54,137 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f3fc9ac71d0>
```

Note

Make sure to use the name given to the `code_to_function` parameter (`name='my-function-name'` in the example above) so that all relevant `start-code` and `end-code` annotations are included. If none of the annotations are marked with the function's name, all annotations without any name are used.

6.4.2 Multi section function

You can use the `# mlrun: start-code` and `# mlrun: end-code` annotations multiple times in a notebook since the whole notebook is scanned. The annotations can be named like the following example, and they can be nameless. If you choose nameless, remember all nameless annotations in the notebook are used.

```
# mlrun: start-code multi-section-function-name

function_name = "multi-section-function-name"

# mlrun: end-code multi-section-function-name
```

Any code between those sections are not included:

```
function_name = "I will be ignored!"
```

```
# mlrun: start-code multi-section-function-name
```

```
def handler(context, event):
    return f"hello from {function_name}"
```

```
# mlrun: end-code multi-section-function-name
```

```
my_multi_section_function = code_to_function('multi-section-function-name', kind='job')
my_multi_section_function.run(name='multi-section-function-name', handler='handler',␣
↪local=True)
```

```
> 2021-11-01 07:43:05,587 [info] starting run multi-section-function-name␣
↪uid=9ac6a0e977a54980b657bae067c2242a DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-11-01 07:43:05,834 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f3fc9a24e10>
```

6.4.3 Annotation's position in code cell

`# mlrun: start-code` and `# mlrun: end-code` annotations are relative to their positions inside the code block. Notice how the assignments to `function_name` below `# mlrun: end-code` don't override the assignment between the annotations in the function's context.

```
# mlrun: start-code part-cell-function

def handler(context, event):
```

(continues on next page)

(continued from previous page)

```
    return f"hello from {function_name}"

function_name = "part-cell-function"

# mlrun: end-code part-cell-function

function_name = "I will be ignored"
```

```
my_multi_section_function = code_to_function('part-cell-function', kind='job')
my_multi_section_function.run(name='part-cell-function', handler='handler', local=True)
```

```
> 2021-11-01 07:43:14,347 [info] starting run part-cell-function␣
↪uid=5426e665c7bc4ba492e0a704c5555fb6 DB=http://mlrun-api:8080
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-11-01 07:43:14,628 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f3fc9a2bf50>
```

6.4.4 Guidelines

- Make sure that every `# mlrun: start-code` has a corresponding `# mlrun: end-code` before the next `# mlrun: start-code` in the notebook.
- Only one MLRun function can have a nameless annotation per notebook.
- Do not use multiple `# mlrun: start-code` nor multiple `# mlrun: end-code` annotations in a single code cell. Only the first appearance of each is used.
- Using single annotations:
  - Use a `# mlrun: start-code` alone, and all code blocks from the annotation to the end of the notebook are included.
  - Use a `# mlrun: end-code` alone, and all code blocks from the beginning of the notebook to the annotation are included.

6.5 Attach storage to functions

In the vast majority of cases, an MLRun function requires access to storage. This storage might be used to provide inputs to the function including data-sets to process or data-streams that contain input events. Typically, storage is used to store function outputs and result artifacts. For example, trained models or processed data-sets.

Since MLRun functions can be distributed and executed in Kubernetes pods, the storage used would typically be shared, and execution pods would need some added configuration options applied to them so that the function code is able to access the designated storage. These configurations might be k8s volume mounts, specific environment variables that contain configuration and credentials, and other configuration of security settings. These storage configurations are not applicable to functions running locally in the development environment, since they are executed in the local context.

The common types of shared storage are:

1. `v3io` storage through API — When running as part of the Iguazio system, MLRun has access to the system's v3io storage through paths such as `v3io:///projects/my_projects/file.csv`. To enable this type of access, several environment variables need to be configured in the pod that provide the v3io API URL and access keys.
2. `v3io` storage through FUSE mount — Some tools cannot utilize the `v3io` API to access it and need basic filesystem semantics. For that purpose, `v3io` provides a FUSE (Filesystem in user-space) driver that can be used to mount `v3io` containers as specific paths in the pod itself. For example `/User`. To enable this, several specific volume mount configurations need to be applied to the pod spec.
3. NFS storage access — When MLRun is deployed as open-source, independent of Iguazio, the deployment automatically adds a pod running NFS storage. To access this NFS storage through pods, a kubernetes `pvc` mount is needed.
4. Others — As use-cases evolve, other cases of storage access may be needed. This will require various configurations to be applied to function execution pods.

MLRun attempts to offload this storage configuration task from the user by automatically applying the most common storage configuration to functions. As a result, most cases do not require any additional storage configurations before executing a function as a Kubernetes pod. The configurations applied by MLRun are:

- In an Iguazio system, apply configurations for `v3io` access through the API.
- In an open-source deployment where NFS is configured, apply configurations for `pvc` access to NFS storage.

This MLRun logic is referred to as **auto-mount**.

In this section

6.5.1 Disabling auto-mount

In cases where the default storage configuration does not fit the function needs, MLRun allows for function spec modifiers to be manually applied to functions. These modifiers can add various configurations to the function spec, adding environment variables, mounts and additional configurations. MLRun also provides a set of common modifiers that can be used to apply storage configurations. These modifiers can be applied by using the `.apply()` method on the function and adding the modifier to apply. You can see some examples of this later in this page.

When a different storage configuration is manually applied to a function, MLRun's auto-mount logic is disabled. This prevents conflicts between configurations. The auto-mount logic can also be disabled by setting `func.spec.disable_auto_mount = True` on any MLRun function.

6.5.2 Modifying the auto-mount default configuration

The default auto-mount behavior applied by MLRun is controlled by setting MLRun configuration parameters. For example, the logic can be set to automatically mount the `v3io` FUSE driver on all functions, or perform `pvc` mount for NFS storage on all functions. The following code demonstrates how to apply the `v3io` FUSE driver by default:

```
# Change MLRun auto-mount configuration
import mlrun.mlconf

mlrun.mlconf.storage.auto_mount_type = "v3io_fuse"
```

Each of the auto-mount supported methods applies a specific modifier function. The supported methods are:

- `v3io_credentials` — apply `v3io` credentials needed for `v3io` API usage. Applies the *v3io_cred()* modifier.
- `v3io_fuse` — create Fuse driver mount. Applies the *mount_v3io()* modifier.
- `pvc` — create a `pvc` mount. Applies the *mount_pvc()* modifier.
- `auto` — the default auto-mount logic as described above (either `v3io_credentials` or `pvc`).
- `none` — perform no auto-mount (same as using `disable_auto_mount = True`).

The modifier functions executed by auto-mount can be further configured by specifying their parameters. These can be provided in the `storage.auto_mount_params` configuration parameters. Parameters can be passed as a string made of `key=value` pairs separated by commas. For example, the following code runs a `pvc` mount with specific parameters:

```
mlrun.mlconf.storage.auto_mount_type = "pvc"
pvc_params = {
    "pvc_name": "my_pvc_mount",
    "volume_name": "pvc_volume",
    "volume_mount_path": "/mnt/storage/nfs",
}
mlrun.mlconf.storage.auto_mount_params = ",".join(
    [f"{key}={value}" for key, value in pvc_params.items()]
)
```

Alternatively, the parameters can be provided as a base64-encoded JSON object, which can be useful when passing complex parameters or strings that contain special characters:

```
pvc_params_str = base64.b64encode(json.dumps(pvc_params).encode())
mlrun.mlconf.storage.auto_mount_params = pvc_params_str
```

6.6 Images and their usage in MLRun

Every release of MLRun includes several images for different usages. The build and the infrastructure images are described, and located, in the README. They are also published to dockerhub and quay.io.

In this section

- *Using images*
- *MLRun images and how to build them*
- *MLRun images and external docker images*

6.6.1 Using images

See Kubernetes Jobs & Images.

6.6.2 MLRun images and how to build them

See README.

6.6.3 MLRun images and external docker images

There is no difference in the usage between the MLRun images and external docker images. However:

- MLRun images resolve auto tags: If you specify `image="mlrun/mlrun"` the API fills in the tag by the client version, e.g. changes it to `mlrun/mlrun:1.2.0`. So, if the client gets upgraded you'll automatically get a new image tag.
- Where the data node registry exists, MLRun Appends the registry prefix, so the image loads from the datanode registry. This pulls the image more quickly, and also supports air-gapped sites. When you specify an MLRun image, for example `mlrun/mlrun:1.2.0`, the actual image used is similar to `datanode-registry.iguazio-platform.app.vm/mlrun/mlrun:1.2.0`.

These characteristics are great when you're working in a POC or development environment. But MLRun typically upgrades packages as part of the image, and therefore the default MLRun images can break your product flow.

Working with images in production

For production you should create your own images to ensure that the image is fixed.

- Pin the image tag, e.g. `image="mlrun/mlrun:1.2.0"`. This maintains the image tag at 1.1.0 even when the client is upgraded. Otherwise, an upgrade of the client would also upgrade the image. (If you specify an external (not MLRun images) docker image, like python, the result is the docker/k8s default behavior, which defaults to `latest` when the tag is not provided.)
- Pin the versions of requirements, again to avoid breakages, e.g. `pandas==1.4.0`. (If you only specify the package name, e.g. pandas, then pip/conda (python's package managers) just pick up the latest version.)

6.7 Build function image

As discussed in *Images and their usage in MLRun*, MLRun provides pre-built images which contain the components necessary to execute an MLRun runtime. In some cases, however, custom images need to be created. This page details this process and the available options.

6.7.1 When is a build required?

In many cases an MLRun runtime can be executed without having to build an image. This will be true when the basic MLRun images fulfill all the requirements for the code to execute. It is required to build an image if one of the following is true:

- The code uses additional Python packages, OS packages, scripts or other configurations that need to be applied
- The code uses different base-images or different versions of MLRun images than provided by default
- Executed source code has changed, and the image has the code packaged in it - see here for more details on source code, and using *with_code()* to avoid re-building the image when the code has changed
- The code runs nuclio functions, which are packaged as images (the build is triggered by MLRun and executed by nuclio)

The build process in MLRun is based on Kaniko and automated by MLRun - MLRun generates the dockerfile for the build process, and configures Kaniko with parameters needed for the build.

Building images is done through functions provided by the *MlrunProject* class. By using project functions, the same process is used to build and deploy a stand-alone function or functions serving as steps in a pipeline.

6.7.2 Automatically building images

MLRun has the capability to auto-detect when a function image needs to first be built. Following is an example that will require building of the image:

```
project = mlrun.new_project(project_name, "./proj")

project.set_function(
    "train_code.py",
    name="trainer",
    kind="job",
    image="mlrun/mlrun",
    handler="train_func",
    requirements=["pandas"]
)

# auto_build will trigger building the image before running,
# due to the additional requirements.
project.run_function("trainer", auto_build=True)
```

Using the `auto_build` option is only suitable when the build configuration does not change between runs of the runtime. For example, if during the development process new requirements were added, the `auto_build` parameter should not be used, and manual build is needed to re-trigger a build of the image.

In the example above, the `requirements` parameter was used to specify a list of additional Python packages required by the code. This option directly affects the image build process - each requirement is installed using `pip` as part of the docker-build process. The `requirements` parameter can also contain a path to a requirements file, making it easier to reuse an existing configuration rather than specify a list of packages.

6.7.3 Manually building an image

To manually build an image, use the *build_function()* function, which provides multiple options that control and configure the build process.

Specifying base image

To use an existing image as the base image for building the image, set the image name in the `base_image` parameter. Note that this image serves as the base (dockerfile `FROM` property), and should not to be confused with the resulting image name, as specified in the `image` parameter.

```
project.build_function(
   "trainer",
   base_image="myrepo/my_base_image:latest",
)
```

Running commands

To run arbitrary commands during the image build, pass them in the `commands` parameter of *build_function()*. For example:

```
github_repo = "myusername/myrepo.git@mybranch"

project.build_function(
   "trainer",
   base_image="myrepo/base_image:latest",
   commands= [
        "pip install git+https://github.com/" + github_repo,
        "mkdir -p /some/path && chmod 0777 /some/path",
   ]
)
```

These commands are added as `RUN` operations to the dockerfile generating the image.

MLRun package deployment

The `with_mlrun` and `mlrun_version_specifier` parameters allow control over the inclusion of the MLRun package in the build process. Depending on the base-image used for the build, the MLRun package may already be available in which case use `with_mlrun=False`. If not specified, MLRun will attempt to detect this situation - if the image used is one of the default MLRun images released with MLRun, `with_mlrun` is automatically set to `False`. If the code execution requires a different version of MLRun than the one used to deploy the function, set the `mlrun_version_specifier` to point at the specific version needed. This uses the published MLRun images of the specified version instead. For example:

```
project.build_function(
   "trainer",
   with_mlrun=True,
   mlrun_version_specifier="1.0.0"
)
```

Working with code repository

As the code matures and evolves, the code will usually be stored in a git code repository. When the MLRun project is associated with a git repo (see *Create, save, and use projects* for details), functions can be added by calling *set_function()* and setting `with_repo=True`. This indicates that the code of the function should be retrieved from the project code repository.

In this case, the entire code repository will be retrieved from git as part of the image-building process, and cloned into the built image. This is recommended when the function relies on code spread across multiple files and also is usually preferred for production code, since it means that the code of the function is stable, and further modifications to the code will not cause instability in deployed images.

During the development phase it may be desired to retrieve the code in runtime, rather than re-build the function image every time the code changes. To enable this, use *set_source()* which gets a path to the source (can be a git repository or a tar or zip file) and set `pull_at_runtime=True`.

Using a private Docker registry

By default, images are pushed to the registry configured during MLRun deployment, using the configured registry credentials.

To push resulting images to a different registry, specify the registry URL in the `image` parameter. If the registry requires credentials, create a k8s secret containing these credentials, and pass its name in the `secret_name` parameter.

When using ECR as registry, MLRun uses Kaniko's ECR credentials helper, in which case the secret provided should contain AWS credentials needed to create ECR repositories, as described here. MLRun detects automatically that the registry is an ECR registry based on its URL and configures Kaniko to use the ECR helper. For example:

```
# AWS credentials stored in a k8s secret -
# kubectl create secret generic ecr-credentials --from-file=<path to .aws/credentials>

project.build_function(
    "trainer",
    image="<aws_account_id>.dkr.ecr.us-east-2.amazonaws.com/myrepo/image:v1",
    secret_name="ecr-credentials",
)
```

Build environment variables

It is possible to pass environment variables that will be set in the Kaniko pod that executes the build. This may be useful to pass important information needed for the build process. The variables are passed as a dictionary in the `builder_env` parameter, for example:

```
project.build_function(
   ...
   builder_env={"GIT_TOKEN": token},
)
```

6.7.4 Deploying nuclio functions

When using nuclio functions, the image build process is done by nuclio as part of the deployment of the function. Most of the configurations mentioned in this page are available for nuclio functions as well. To deploy a nuclio function, use *deploy_function()* instead of using *build_function()* and *run_function()*.

6.7.5 Creating default Spark runtime images

When using Spark to execute code, either using a Spark service (remote-spark) or the Spark operator, an image is required that contains both Spark binaries and dependencies, and MLRun code and dependencies. This image is used in the following scenarios:

1. For remote-spark, the image is used to run the initial MLRun code which will submit the Spark job using the remote Spark service
2. For Spark operator, the image is used for both the driver and the executor pods used to execute the Spark job

This image needs to be created any time a new version of Spark or MLRun is being used, to ensure that jobs are executed with the correct versions of both products.

To prepare this image, MLRun provides the following facilities:

```
# For remote Spark
from mlrun.runtimes import RemoteSparkRuntime
RemoteSparkRuntime.deploy_default_image()

# For Spark operator
from mlrun.runtimes import Spark3Runtime
Spark3Runtime.deploy_default_image()
```

6.8 Node affinity

You can assign a node or a node group for services or for jobs executed by a service. When specified, the service or the pods of a function can only run on nodes whose labels match the node selector entries configured for the specific service. If node selection for the service is not specified, the selection criteria defaults to the Kubernetes default behavior, and jobs run on a random node.

For MLRun and Nuclio, you can also specify node selectors on a per-job basis. The default node selectors (defined at the service level) are applied to all jobs unless you specifically override them for an individual job.

You can configure node affinity for:

- Jupyter
- Presto (The node selection also affects any additional services that are directly affected by Presto, for example hive and mariadb, which are created if Enable hive is checked in the Presto service.)
- Grafana
- Shell
- MLRun (default value applied to all jobs that can be overwritten for individual jobs)
- Nuclio (default value applied to all jobs that can be overwritten for individual jobs)

See more about Kubernetes nodeSelector.

6.8.1 UI configuration

Configure node selection on the service level in the service's **Custom Parameters** tab, under **Resources**, by adding or removing Key:Value pairs. For MLRun and Nuclio, this is the default node selection for all MLRun jobs and Nuclio functions.

You can also configure the node selection for individual MLRun jobs by going to **Platform dashboard | Projects | New Job | Resources | Node selector**, and adding or removing Key:Value pairs. Configure the node selection for individual Nuclio functions when creating a function in the **Confguration** tab, under **Resources**, by adding Key:Value pairs.

6.8.2 SDK configuration

Configure node selection by adding the key:value pairs in your Jupyter notebook formatted as a Python dictionary. For example:

```
import mlrun
import os
train_fn = mlrun.code_to_function('training',
                            kind='job',
                            handler='my_training_function')
train_fn.with_preemption_mode(mode="prevent")
train_fn.run(inputs={"dataset" :my_data})


# Add node selection
func.with_node_selection(node_selector={name})
```

See with_node_selection.

6.9 Managing job resources

MLRun orchestrates serverless functions over Kubernetes. You can specify the resource requirements (CPU, memory, GPUs), preferences, and priorities in the logical function object. These are used during the function deployment.

Configuration of job resources is relevant for all supported cloud platforms.

In this section

6.9.1 Replicas

Some runtimes can scale horizontally, configured either as a number of replicas: `spec.replicas` or a range (for auto scaling in Dask or Nuclio:

```
spec.min_replicas = 1
spec.max_replicas = 4
```

See more details in Dask, MPIJob and Horovod, Spark, Nuclio.

6.9.2 CPU, GPU, and memory limits for user jobs

When you create a pod in an MLRun job or Nuclio function, the pod has default CPU and memory limits. When the job runs, it can consume resources up to the limits defined. The default limits are set at the service level. You can change the default limit for the service, and also overwrite the default when creating a job, or a function.

See more about Kubernetes Resource Management for Pods and Containers.

UI configuration

When creating a service, set the **Memory** and **CPU** in the **Common Parameters** tab, under **User jobs defaults**. When creating a job or a function, overwrite the default **Memory**, **CPU**, or **GPU** in the **Configuration** tab, under **Resources**.

SDK configuration

Configure the limits assigned to a function by using `with_limits`. For example:

```
training_function = mlrun.code_to_function("training.py", name="training", handler="train
↪",
                                                                    kind="mpijob",␣
↪image="mlrun/ml-models-gpu")
training_function.spec.replicas = 2
training_function.with_requests(cpu=2)
training_function.gpus(1)
```

Note

When specifying GPUs, MLRun uses `nvidia.com/gpu` as default GPU type. To use a different type of GPU, specify it using the optional `gpu_type` parameter.

6.9.3 Volumes

When you create a pod in an MLRun job or Nuclio function, the pod by default has access to a file-system which is ephemeral, and gets deleted when the pod completes its execution. In many cases, a job requires access to files residing on external storage, or to files containing configurations and secrets exposed through Kubernetes config-maps or secrets. Pods can be configured to consume the following types of volumes, and to mount them as local files in the local pod file-system:

- V3IO containers: when running on the Iguazio system, pods have access to the underlying V3IO shared storage. This option mounts a V3IO container or a subpath within it to the pod through the V3IO FUSE driver.

- PVC: Mount a Kubernetes persistent volume claim (PVC) to the pod. The persistent volume and the claim need to be configured beforehand.
- Config Map: Mount a Kubernetes Config Map as local files to the pod.
- Secret: Mount a Kubernetes secret as local files to the pod.

For each of the options, a name needs to be assigned to the volume, as well as a local path to mount the volume at (using a Kubernetes Volume Mount). Depending on the type of the volume, other configuration options may be needed, such as an access-key needed for V3IO volume.

See more about Kubernetes Volumes.

MLRun supports the concept of volume auto-mount which automatically mounts the most commonly used type of volume to all pods, unless disabled. See more about MLRun auto mount.

UI configuration

You can configure Volumes when creating a job, rerunning an existing job, and creating an ML function. Modify the Volumes for an ML function by pressing **ML functions**, then of the function, **Edit | Resources | Volumes** drop-down list.

Select the volume mount type: either Auto (using auto-mount), Manual or None. If selecting Manual, fill in the details in the volumes list for each volume to mount to the pod. Multiple volumes can be configured for a single pod.

SDK configuration

Configure volumes attached to a function by using the `apply` function modifier on the function.

For example, using v3io storage:

```
# import the training function from the Function Hub (hub://)
train = mlrun.import_function('hub://sklearn_classifier')# Import the function:
open_archive_function = mlrun.import_function("hub://open_archive")

# use mount_v3io() for iguazio volumes
open_archive_function.apply(mount_v3io())
```

You can specify a list of the v3io path to use and how they map inside the container (using volume_mounts). For example:

```
mlrun.mount_v3io(name='data',access_key='XYZ123..',volume_mounts=[mlrun.VolumeMount("/
↪data", "projects/proj1/data")])
```

See full details in mount_v3io.

Alternatively, using a PVC volume:

```
mount_pvc(pvc_name="data-claim", volume_name="data", volume_mount_path="/data")
```

See full details in mount_pvc.

6.9.4 Preemption mode: Spot vs. On-demand nodes

Node selector is supported for all cloud platforms. It is relevant for MLRun and Nuclio only.

When running ML functions you might want to control whether to run on spot nodes or on-demand nodes. Preemption mode controls whether pods can be scheduled on preemptible (spot) nodes. Preemption mode is supported for all functions.

Preemption mode uses Kubernets Taints and Toleration to enforce the mode selected. Read more in Kubernetes Taints and Tolerations.

Why preemption mode

On-demand instances provide full control over the instance lifecycle. You decide when to launch, stop, hibernate, start, reboot, or terminate it. With Spot instances you request capacity from specific available zones, though it is susceptible to spot capacity availability. This is a good choice if you can be flexible about when your applications run and if your applications can be interrupted.

Here are some questions to consider when choosing the type of node:

- Is the function mission critical and must be operational at all times?
- Is the function a stateful function or stateless function?
- Can the function recover from unexpected failure?
- Is this a job that should run only when there are available inexpensive resources?

Important

When an MLRun job is running on a spot node and it fails, it won't get back up again. However, if Nuclio goes down due to a spot issue, it is brought up by Kubernetes.

Kuberenetes has a few methods for configuring which nodes to run on. To get a deeper understanding, see Pod Priority and Preemption. Also, you must understand the configuration of the spot nodes as specified by the cloud provider.

Stateless and Stateful Applications

When deploying your MLRun jobs to specific nodes, take into consideration that on-demand nodes are designed to run stateful applications while spot nodes are designed for stateless applications. MLRun jobs are more stateful by nature. An MLRun job that is assigned to run on a spot node might be subject to interruption; it would have to be designed so that the job/function state will be saved when scaling to zero.

Supported preemption modes

Preemption mode has three values:

- Allow: The function pod can run on a spot node if one is available.
- Constrain: The function pod only runs on spot nodes, and does not run if none is available.
- Prevent: Default. The function pod cannot run on a spot node.

UI configuration

Note

Relevant when MLRun is executed in the Iguazio platform.

You can configure Spot node support when creating a job, rerunning an existing job, and creating an ML function. The **Run on Spot nodes** drop-down list is in the **Resources** section of jobs. Configure the Spot node support for individual Nuclio functions when creating a function in the **Configuration** tab, under **Resources**.

SDK configuration

Configure preemption mode by adding the `with_preemption_mode` parameter in your Jupyter notebook, and specifying a mode from the list of values above. This example illustrates a function that cannot be scheduled on preemptible nodes:

```
import mlrun
import os

train_fn = mlrun.code_to_function('training',
                            kind='job',
                            handler='my_training_function')
train_fn.with_preemption_mode(mode="prevent")
train_fn.run(inputs={"dataset" :my_data})
```

See `with_preemption_mode`.

Alternatively, you can specify the preemption using `with_priority_class` and `fn.with_priority_class(name="default-priority")node_selector`. This example specifies that the pod/function runs only on non-preemptible nodes:

```
import mlrun
import os
train_fn = mlrun.code_to_function('training',
                            kind='job',
                            handler='my_training_function')
train_fn.with_preemption_mode(mode="prevent")
train_fn.run(inputs={"dataset" :my_data})

fn.with_priority_class(name="default-priority")
fn.with_node_selection(node_selector={"app.iguazio.com/lifecycle":"non-preemptible"})
```

See `with_node_selection`.

6.9.5 Pod priority for user jobs

Pods (services, or jobs created by those services) can have priorities, which indicate the relative importance of one pod to the other pods on the node. The priority is used for scheduling: a lower priority pod can be evicted to allow scheduling of a higher priority pod. Pod priority is relevant for all pods created by the service. For MLRun, it applies to the jobs created by MLRun. For Nuclio it applies to the pods of the Nuclio-created functions.

Eviction uses these values to determine what to evict with conjunction to the pods priority Pod Priority and Preemption.

Pod priority is specified through Priority classes, which map to a priority value. The priority values are: High, Medium, Low. The default is Medium. Pod priority is supported for:

- MLRun jobs: the default priority class for the jobs that MLRun creates.
- Nuclio functions: the default priority class for the user-created functions.
- Jupyter
- Presto (The pods priority also affects any additional services that are directly affected by Presto, for example like hive and mariadb, which are created if Enable hive is checked in the Presto service.)
- Grafana
- Shell

UI configuration

Note

Relevant when MLRun is executed in the Iguazio platform.

Configure the default priority for a service, which is applied to the service itself or to all subsequently created user-jobs in the service's **Common Parameters** tab, **User jobs defaults** section, **Priority class** drop-down list.

Modify the priority for an ML function by pressing **ML functions**, then of the function, **Edit | Resources | Pods Priority** drop-down list.

SDK configuration

Configure pod priority by adding the priority class parameter in your Jupyter notebook. For example:

```
import mlrun
import os
train_fn = mlrun.code_to_function('training',
                            kind='job',
                            handler='my_training_function')
train_fn.with_priority_class(name={value})
train_fn.run(inputs={"dataset" :my_data})
```

See with_priority_class.

6.10 Function Hub

This section demonstrates how to import a function from the Hub into your project, and provides some basic instructions on how to run the function and view the results.

In this section

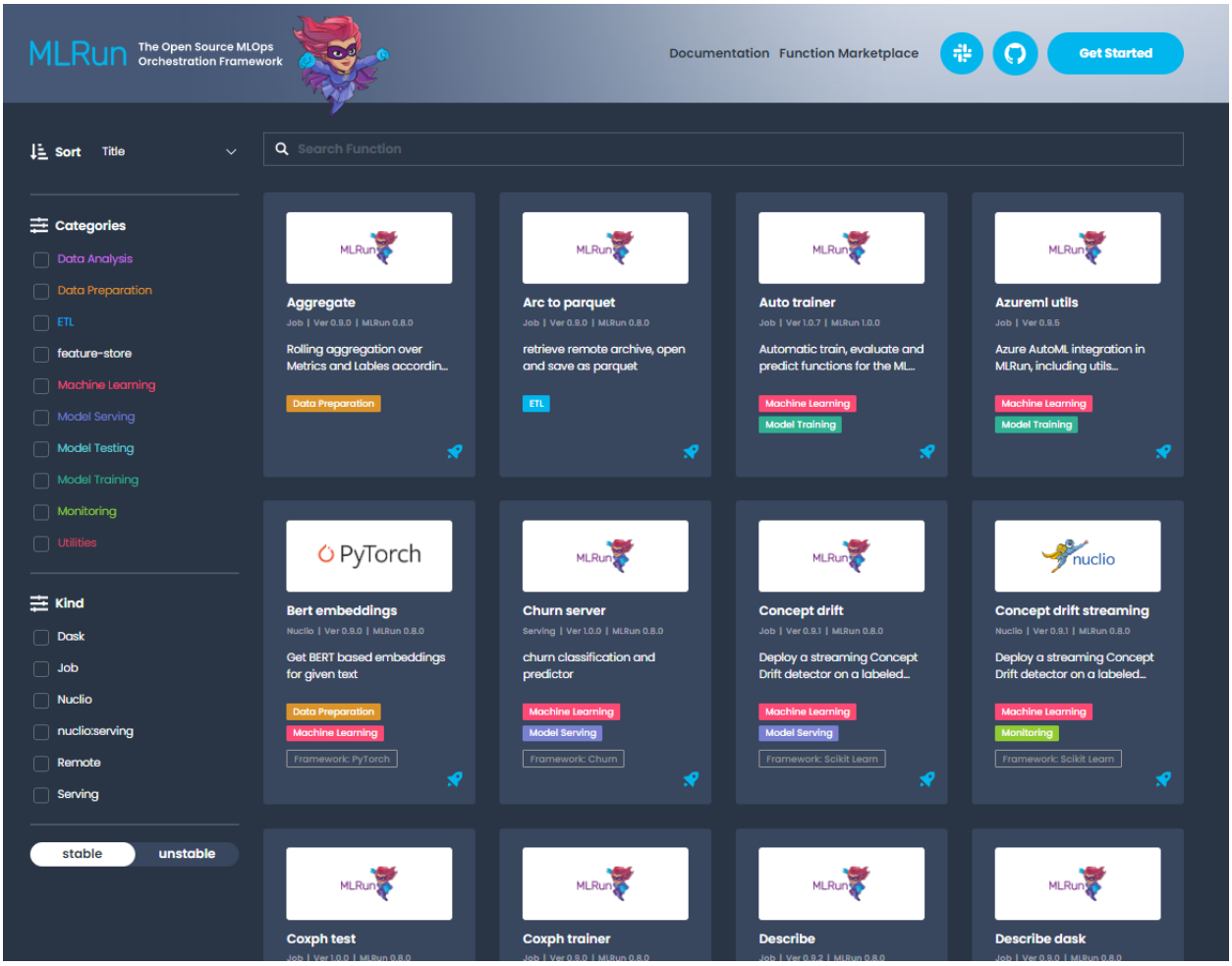
6.10.1 Function Hub

The MLRun Function Hub has a wide range of functions that can be used for a variety of use cases. There are functions for ETL, data preparation, training (ML & Deep learning), serving, alerts and notifications and more. Each function has a docstring that explains how to use it. In addition, the functions are associated with categories to make it easier for you to find the relevant one.

Functions can be easily imported into your project and therefore help you to speed up your development cycle by reusing built-in code.

6.10.2 Searching for functions

The Function Hub is located here. You can search and filter the categories and kinds to find a function that meets your needs.



6.10.3 Setting the project configuration

The first step for each project is to set the project name and path:

```
from os import path, getenv
from mlrun import new_project

project_name = 'load-func'
project_path = path.abspath('conf')
project = new_project(project_name, project_path, init_git=True)

print(f'Project path: {project_path}\nProject name: {project_name}')
```

Set the artifacts path

The artifact path is the default path for saving all the artifacts that the functions generate:

```
from mlrun import run_local, mlconf, import_function, mount_v3io

# Target location for storing pipeline artifacts
artifact_path = path.abspath('jobs')
# MLRun DB path or API service URL
mlconf.dbpath = mlconf.dbpath or 'http://mlrun-api:8080'

print(f'Artifacts path: {artifact_path}\nMLRun DB path: {mlconf.dbpath}')
```

6.10.4 Loading functions from the Hub

Run `project.set_function` to load a functions. `set_function` updates or adds a function object to the project.

`set_function(func, name='', kind='', image=None, with_repo=None)`

Parameters:

- **func** — function object or spec/code url.
- **name** — name of the function (under the project).
- **kind** — runtime kind e.g. job, nuclio, spark, dask, mpijob. Default: job.
- **image** — docker image to be used, can also be specified in the function object/yaml.
- **with_repo** — add (clone) the current repo to the build source.

Returns: project object

For more information see the *set_function()*API documentation.

Load function example

This example loads the describe function. This function analyzes a csv or parquet file for data analysis.

```
project.set_function('hub://describe', 'describe')
```

Create a function object called my_describe:

```
my_describe = project.func('describe')
```

6.10.5 View the function params

To view the parameters, run the function with .doc():

```
my_describe.doc()
```

```
    function: describe
    describe and visualizes dataset stats
    default handler: summarize
    entry points:
```

(continues on next page)

(continued from previous page)

```
    summarize: Summarize a table
      context(MLClientCtx)  - the function context, default=
      table(DataItem)  - MLRun input pointing to pandas dataframe (csv/parquet file␣
↪path), default=
      label_column(str)  - ground truth column label, default=None
      class_labels(List[str])  - label for each class in tables and plots, default=[]
      plot_hist(bool)  - (True) set this to False for large tables, default=True
      plots_dest(str)  - destination folder of summary plots (relative to artifact_
↪path), default=plots
      update_dataset  - when the table is a registered dataset update the charts in-
↪place, default=False
```

6.10.6 Running the function

Use the `run` method to to run the function.

When working with functions pay attention to the following:

- Input vs. params — for sending data items to a function, send it via "inputs" and not as params.
- Working with artifacts — Artifacts from each run are stored in the `artifact_path`, which can be set globally with the environment variable (MLRUN_ARTIFACT_PATH) or with the config. If it's not already set you can create a directory and use it in the runs. Using `{{run.uid}}` in the path creates a unique directory per run. When using pipelines you can use the `{{workflow.uid}}` template option.

This example runs the describe function. This function analyzes a dataset (in this case it's a csv file) and generates HTML files (e.g. correlation, histogram) and saves them under the artifact path.

```
DATA_URL = 'https://s3.wasabisys.com/iguazio/data/iris/iris_dataset.csv'

my_describe.run(name='describe',
                inputs={'table': DATA_URL},
                artifact_path=artifact_path)
```

Saving the artifacts in a unique folder for each run

```
out = mlconf.artifact_path or path.abspath('./data')
my_describe.run(name='describe',
                inputs={'table': DATA_URL},
                artifact_path=path.join(out, '{{run.uid}}'))
```

Viewing the jobs & the artifacts

There are few options to view the outputs of the jobs we ran:

- In Jupyter the result of the job is displayed in the Jupyter notebook. When you click on the artifacts it displays its content in Jupyter.
- In the MLRun UI, under the project name, you can view the job that was running as well as the artifacts it generated.

CHAPTER

SEVEN

DATA AND ARTIFACTS

One of the biggest challenge in distributed systems is handling data given the different access methods, APIs, and authentication mechanisms across types and providers.

Working with the abstractions enable you to securely access different data sources through a single API, many continuance methods (e.g. to/from DataFrame, get, download, list, ...), automated data movement, and versioning.

MLRun provides these main abstractions to access structured and unstructured data:

In this section

7.1 Data stores

A data store defines a storage provider (e.g. file system, S3, Azure blob, Iguazio v3io, etc.).

In this section

7.1.1 Shared data stores

MLRun supports multiple data stores. (More can easily added by extending the `DataStore` class.) Data stores are referred to using the schema prefix (e.g. `s3://my-bucket/path`). The currently supported schemas and their urls:

- **files** — local/shared file paths, format: `/file-dir/path/to/file`
- **http, https** — read data from HTTP sources (read-only), format: `https://host/path/to/file`
- **s3** — S3 objects (AWS or other endpoints), format: `s3://<bucket>/path/to/file`
- **v3io, v3ios** — Iguazio v3io data fabric, format: `v3io://[<remote-host>]/<data-container>/path/to/file`
- **az** — Azure Blob storage, format: `az://<container>/path/to/file`
- **gs, gcs** — Google Cloud Storage objects, format: `gs://<bucket>/path/to/file`
- **store** — MLRun versioned artifacts (see Artifacts), format: `store://artifacts/<project>/<artifact-name>[:tag]`
- **memory** — in memory data registry for passing data within the same process, format `memory://key`, use `mlrun.datastore.set_in_memory_item(key, value)` to register in memory data items (byte buffers or DataFrames).

7.1.2 Storage credentials and parameters

Data stores might require connection credentials. These can be provided through environment variables or project/job context secrets. The exact credentials depend on the type of the data store and are listed in the following table. Each parameter specified can be provided as an environment variable, or as a project-secret that has the same key as the name of the parameter.

MLRun jobs executed remotely run in independent pods, with their own environment. When setting an environment variable in the development environment (for example Jupyter), this has no effect on the executing pods. Therefore, before executing jobs that require access to storage credentials, these need to be provided by assigning environment variables to the MLRun runtime itself, assigning secrets to it, or placing the variables in project-secrets.

Warning: Passing secrets as environment variables to runtimes is discouraged, as they are exposed in the pod spec. Refer to Working with secrets for details on secret handling in MLRun.

For example, running a function locally:

```
# Access object in AWS S3, in the "input-data" bucket
source_url = "s3://input-data/input_data.csv"

os.environ["AWS_ACCESS_KEY_ID"] = "<access key ID>"
os.environ["AWS_SECRET_ACCESS_KEY"] = "<access key>"

# Execute a function that reads from the object pointed at by source_url.
# When running locally, the function can use the local environment variables.
local_run = func.run(name='aws_test', inputs={'source_url': source_url}, local=True)
```

Running the same function remotely:

```
# Executing the function remotely using env variables (not recommended!)
func.set_env("AWS_ACCESS_KEY_ID", "<access key ID>").set_env("AWS_SECRET_ACCESS_KEY", "
↪<access key>")
remote_run = func.run(name='aws_test', inputs={'source_url': source_url})

# Using project-secrets (recommended) - project secrets are automatically mounted to␣
↪project functions
secrets = {"AWS_ACCESS_KEY_ID": "<access key ID>", "AWS_SECRET_ACCESS_KEY": "<access key>
↪"}
db = mlrun.get_run_db()
db.create_project_secrets(project=project_name, provider="kubernetes", secrets=secrets)

remote_run = func.run(name='aws_test', inputs={'source_url': source_url})
```

The following sections list the credentials and configuration parameters applicable to each storage type.

v3io

When running in an Iguazio system, MLRun automatically configures executed functions to use `v3io` storage, and passes the needed parameters (such as access-key) for authentication. Refer to the auto-mount section for more details on this process.

In some cases, the v3io configuration needs to be overridden. The following parameters can be configured:

- `V3IO_API` — URL pointing to the v3io web-API service.
- `V3IO_ACCESS_KEY` — access key used to authenticate with the web API.
- `V3IO_USERNAME` — the user-name authenticating with v3io. While not strictly required when using an access-key to authenticate, it is used in several use-cases, such as resolving paths to the home-directory.

S3

- `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` — access key parameters
- `S3_ENDPOINT_URL` — the S3 endpoint to use. If not specified, it defaults to AWS. For example, to access a storage bucket in Wasabi storage, use `S3_ENDPOINT_URL = "https://s3.wasabisys.com"`
- `MLRUN_AWS_ROLE_ARN` — IAM role to assume. Connect to AWS using the secret key and access key, and assume the role whose ARN is provided. The ARN must be of the format `arn:aws:iam::<account-of-role-to-assume>:role/<name-of-role>`
- `AWS_PROFILE` — name of credentials profile from a local AWS credentials file. When using a profile, the authentication secrets (if defined) are ignored, and credentials are retrieved from the file. This option should be used for local development where AWS credentials already exist (created by `aws` CLI, for example)

Azure Blob storage

The Azure Blob storage can utilize several methods of authentication. Each requires a different set of parameters as listed here:

Authentication method	Parameters
Connection string	`AZURE_STORAGE_CONNECTION_STRING`
SAS token	`AZURE_STORAGE_ACCOUNT_NAMEAZURE_STORAGE_SAS_TOKEN`
Account key	`AZURE_STORAGE_ACCOUNT_NAMEAZURE_STORAGE_KEY`
Service principal with a client secret	`AZURE_STORAGE_ACCOUNT_NAMEAZURE_STORAGE_CLIENT_IDAZURE_STORAGE_CLIENT_SECRETAZURE`

Note: The `AZURE_STORAGE_CONNECTION_STRING` configuration uses the `BlobServiceClient` to access objects. This has limited functionality and cannot be used to access Azure Datalake storage objects. In this case use one of the other authentication methods that use the `fsspec` mechanism.

Google cloud storage

- GOOGLE_APPLICATION_CREDENTIALS — path to the application credentials to use (in the form of a JSON file). This can be used if this file is located in a location on shared storage, accessible to pods executing MLRun jobs.
- GCP_CREDENTIALS — when the credentials file cannot be mounted to the pod, this environment variable may contain the contents of this file. If configured in the function pod, MLRun dumps its contents to a temporary file and points GOOGLE_APPLICATION_CREDENTIALS at it.

7.2 Data items

A data item can be one item or a or collection of items (file, dir, table, etc.).

When running jobs or pipelines, data is passed using the *DataItem* objects. Data items objects abstract away the data backend implementation, provide a set of convenience methods (.as_df, .get, .show, ...), and enable auto logging/versioning of data and metadata.

Example function:

```
def prep_data(context, source_url: mlrun.DataItem, label_column='label'):
    # Convert the DataItem to a Pandas DataFrame
    df = source_url.as_df()
    df = df.drop(label_column, axis=1).dropna()
    context.log_dataset('cleaned_data', df=df, index=False, format='csv')
```

Running the function:

```
prep_data_run = data_prep_func.run(name='prep_data',
                                   handler=prep_data,
                                   inputs={'source_url': source_url},
                                   params={'label_column': 'userid'})
```

In order to call the function with an input you can use the inputs dictionary attribute. In order to pass a simple parameter, use the params dictionary attribute. The input value is the specific item uri (per data store schema) as explained in Shared data stores.

Reading the data results from the run, you can easily get a run output artifact as a DataItem (so that you can view/use the artifact) using:

```
# read the data locally as a Dataframe
prep_data_run.artifact('cleaned_data').as_df()
```

The *DataItem* supports multiple convenience methods such as:

- **get**(), **put**() - to read/write data
- **download**(), **upload**() - to download/upload files
- **as_df**() - to convert the data to a DataFrame object
- **local** - to get a local file link to the data (that is downloaded locally if needed)
- **listdir**(), **stat** - file system like methods
- **meta** - access to the artifact metadata (in case of an artifact uri)
- **show**() - visualizes the data in Jupyter (as image, html, etc.)

See the *DataItem* class documentation for details. mlrun.datastore.DataItem

In order to get a DataItem object from a url use *get_dataitem()* or *get_object()* (returns the `DataItem.get()`).

For example:

```
df = mlrun.get_dataitem('s3://demo-data/mydata.csv').as_df()
print(mlrun.get_object('https://my-site/data.json'))
```

7.3 Artifacts

An artifact is any data that is produced and/or consumed by functions, jobs, or pipelines.

Artifacts metadata is stored in the project's database. The main types of artifacts are:

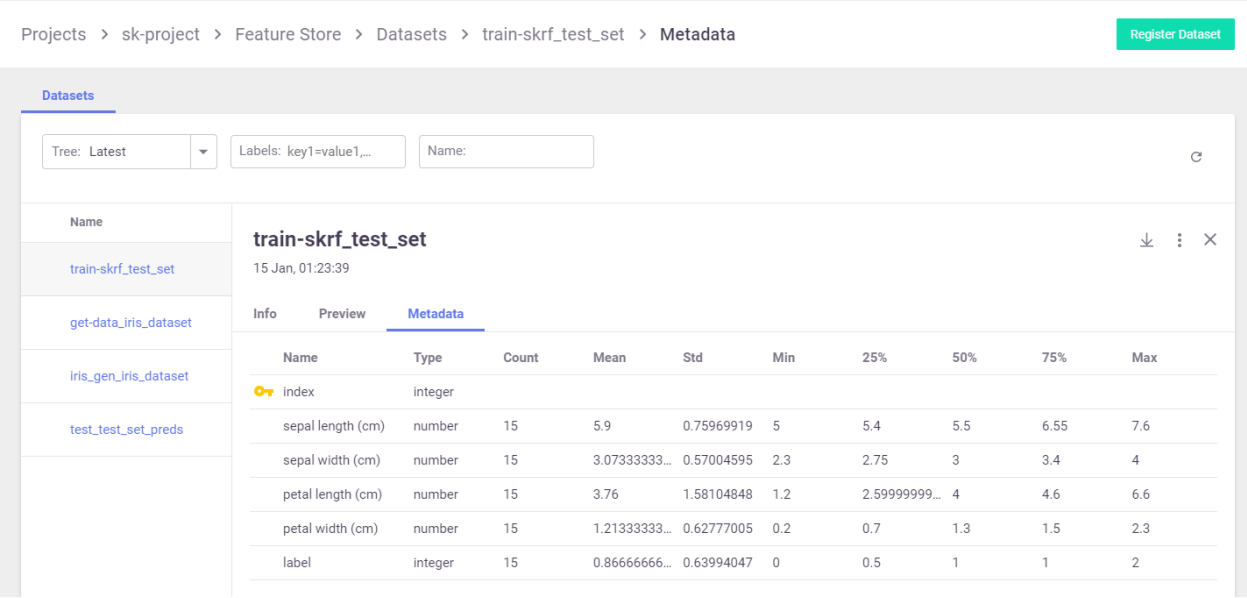
- **Files** — files, directories, images, figures, and plotlines
- **Datasets** — any data, such as tables and DataFrames
- **Models** — all trained models
- **Feature Store Objects** — Feature sets and feature vectors

In this section

7.3.1 Viewing artifacts

Artifacts that are stored in certain paths (see *Artifact path*) can be viewed and managed in the UI. In the **Project** page, select the type of artifact you want to view from the left-hand menu: Feature Store (for feature-sets, feature-vectors and features), Datasets, Artifacts, or Models.

Example dataset artifact screen:



Artifacts that were generated by an MLRun job can also be viewed from the **Jobs > Artifacts** tab.

You can search the artifacts based on time and labels, and you can filter the artifacts by tag type. For each artifact, you can view its content, its location, the artifact type, labels, the producer of the artifact, the artifact owner, last update date, and type-specific information. You can download the artifact. You can also tag and remove tags from artifacts using the UI.

7.3.2 Artifact path

Any path that is supported by MLRun can be used to store artifacts. However, only artifacts that are stored in paths that are system-configured as "allowed" in the MLRun service are visible in the UI. These are:

- MLRun < 1.2: The allowed paths include only v3io paths
- MLRun 1.2 and higher: Allows cloud storage paths — `v3io://`, `s3://`, `az://`, `gcs://`, `gs://`

Jobs use the default or job specific `artifact_path` parameter to determine where the artifacts are stored. The default `artifact_path` can be specified at the cluster level, client level, project level, or job level (at that precedence order), or can be specified as a parameter in the specific `log` operation.

You can set the default `artifact_path` for your environment using the *`set_environment()`* function.

You can override the default `artifact_path` configuration by setting the `artifact_path` parameter of the *`set_environment()`* function. You can use variables in the artifacts path, such as `{{project}}` for the name of the running project or `{{run.uid}}` for the current job/pipeline run UID. (The default artifacts path uses `{{project}}`.) The following example configures the artifacts path to an artifacts directory in the current active directory (`./artifacts`)

```
set_environment(project=project_name, artifact_path='./artifacts')
```

For Iguazio MLOps Platform users

In the platform, the default artifacts path is a /artifacts directory in the predefined "projects" data container: `/v3io/projects/<project name>/artifacts` (for example, `/v3io/projects/myproject/artifacts` for a "myproject" project).

7.3.3 Saving artifacts in run-specific paths

When you specify `{{run.uid}}`, the artifacts for each job are stored in a dedicated directory for each executed job. Under the artifact path, you should see the source-data file in a new directory whose name is derived from the unique run ID. Otherwise, the same artifacts directory is used in all runs, and the artifacts for newer runs override those from the previous runs.

As previously explained, `set_environment` returns a tuple with the project name and artifacts path. You can optionally save your environment's artifacts path to a variable, as demonstrated in the previous steps. You can then use the artifacts-path variable to extract paths to task-specific artifact subdirectories. For example, the following code extracts the path to the artifacts directory of a `training` task, and saves the path to a `training_artifacts` variable:

```
from os import path
training_artifacts = path.join(artifact_path, 'training')
```

Note

The artifacts path uses data store URLs, which are not necessarily local file paths (for example, `s3://bucket/path`). Be careful not to use such paths with general file utilities.

7.3.4 Artifact URIs, versioning, and metadata

Artifacts have unique URIs in the form `store://<type>/<project>/<key/path>[:tag]`. The URI is automatically generated by `log_artifact` and can be used as input to jobs, functions, pipelines, etc.

Artifacts are versioned. Each unique version has a unique IDs (uid) and can have a `tag` label.
When the tag is not specified, it uses the `latest` version.

Artifact metadata and objects can be accessed through the SDK or downloaded from the UI (as YAML files). They host common and object specific metadata such as:

- Common metadata: name, project, updated, version info
- How they were produced (user, job, pipeline, etc.)
- Lineage data (sources used to produce that artifact)
- Information about formats, schema, sample data
- Links to other artifacts (e.g. a model can point to a chart)
- Type-specific attributes

Artifacts can be obtained via the SDK through type specific APIs or using generic artifact APIs such as:

- *`get_dataitem()`* - get the *`DataItem`* object for reading/downloading the artifact content
- *`get_store_resource()`* - get the artifact object

Example artifact URLs:

```
store://artifacts/default/my-table
store://artifacts/sk-project/train-model:e95f757e-7959-4d66-b500-9f6cdb1f0bc7
store://feature-sets/stocks/quotes:v2
store://feature-vectors/stocks/enriched-ticker
```

7.3.5 See also

- *Working with data and model artifacts*
- *Model Artifacts*
- *Logging datasets*

Back to top

7.4 Model Artifacts

An essential piece of artifact management and versioning is storing a model version. This allows the users to experiment with different models and compare their performance, without having to worry about losing their previous results.

The simplest way to store a model named `my_model` is with the following code:

```
from pickle import dumps
model_data = dumps(model)
context.log_model(key='my_model', body=model_data, model_file='my_model.pkl')
```

You can also store any related metrics by providing a dictionary in the `metrics` parameter, such as `metrics={'accuracy': 0.9}`. Furthermore, any additional data that you would like to store along with the model can be specified in the `extra_data` parameter. For example `extra_data={'confusion': confusion.target_path}`

A convenient utility method, `eval_model_v2`, which calculates mode metrics is available in `mlrun.utils`.

See example below for a simple model trained using scikit-learn (normally, you would send the data as input to the function). The last 2 lines evaluate the model and log the model.

```
from sklearn import linear_model
from sklearn import datasets
from sklearn.model_selection import train_test_split
from pickle import dumps

from mlrun.execution import MLClientCtx
from mlrun.mlutils import eval_model_v2

def train_iris(context: MLClientCtx):

    # Basic scikit-learn iris SVM model
    X, y = datasets.load_iris(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42)
    model = linear_model.LogisticRegression(max_iter=10000)
    model.fit(X_train, y_train)

    # Evaluate model results and get the evaluation metrics
    eval_metrics = eval_model_v2(context, X_test, y_test, model)

    # Log model
    context.log_model("model",
                      body=dumps(model),
                      artifact_path=context.artifact_subpath("models"),
```

(continues on next page)

(continued from previous page)

```
                      extra_data=eval_metrics,
                      model_file="model.pkl",
                      metrics=context.results,
                      labels={"class": "sklearn.linear_model.LogisticRegression"})
```

Save the code above to `train_iris.py`. The following code loads the function and runs it as a job. See the quick-start page to learn how to create the project and set the artifact path.

```
from mlrun import code_to_function

gen_func = code_to_function(name='train_iris',
                            filename='train_iris.py',
                            handler='train_iris',
                            kind='job',
                            image='mlrun/ml-models')

train_iris_func = project.set_function(gen_func).apply(auto_mount())

train_iris = train_iris_func.run(name='train_iris',
                                 handler='train_iris',
                                 artifact_path=artifact_path)
```

You can now use `get_model` to read the model and run it. This function will get the model file, metadata, and extra data. The input can be either the path of the model, or the directory where the model resides. If you provide a directory, the function will search for the model file (by default it searches for .pkl files)

The following example gets the model from `models_path` and test data in `test_set` with the expected label provided as a column of the test data. The name of the column containing the expected label is provided in `label_column`. The example then retrieves the models, runs the model with the test data and updates the model with the metrics and results of the test data.

```
from pickle import load

from mlrun.execution import MLClientCtx
from mlrun.datastore import DataItem
from mlrun.artifacts import get_model, update_model
from mlrun.mlutils import eval_model_v2

def test_model(context: MLClientCtx,
               models_path: DataItem,
               test_set: DataItem,
               label_column: str):

    if models_path is None:
        models_path = context.artifact_subpath("models")
    xtest = test_set.as_df()
    ytest = xtest.pop(label_column)

    model_file, model_obj, _ = get_model(models_path)
    model = load(open(model_file, 'rb'))

    extra_data = eval_model_v2(context, xtest, ytest.values, model)
    update_model(model_artifact=model_obj, extra_data=extra_data,
```

(continues on next page)

(continued from previous page)

```
                    metrics=context.results, key_prefix='validation-')
```

To run the code, place the code above in `test_model.py` and use the following snippet. The model from the previous step is provided as the `models_path`:

```
from mlrun.platforms import auto_mount
gen_func = code_to_function(name='test_model',
                            filename='test_model.py',
                            handler='test_model',
                            kind='job',
                            image='mlrun/ml-models')

func = project.set_function(gen_func).apply(auto_mount())

run = func.run(name='test_model',
                handler='test_model',
                params={'label_column': 'label'},
                inputs={'models_path': train_iris.outputs['model'],
                        'test_set': 'https://s3.wasabisys.com/iguazio/data/iris/iris_
↪dataset.csv'}),
                artifact_path=artifact_path)
```

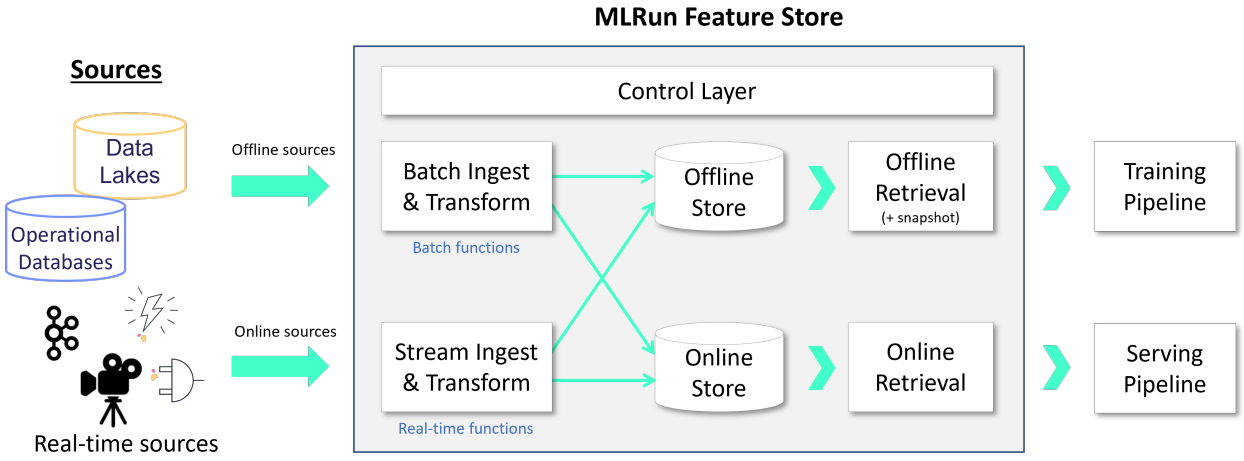
CHAPTER

EIGHT

FEATURE STORE

A feature store provides a single pane of glass for sharing all available features across the organization along with their metadata. MLRun Feature store support security, versioning, and data snapshots, enabling better data lineage, compliance, and manageability.

As illustrated in the diagram below, feature stores provide a mechanism (**`Feature Sets`**) to read data from various online or offline sources, conduct a set of data transformations, and persist the data in online and offline storage. Features are stored and cataloged along with all their metadata (schema, labels, statistics, etc.), allowing users to compose **`Feature Vectors`** and use them for training or serving. The Feature Vectors are generated when needed, taking into account data versioning and time correctness (time traveling). Different function kinds (Nuclio, Spark, Dask) are used for feature retrieval, real-time engine for serving, and batch one for training.



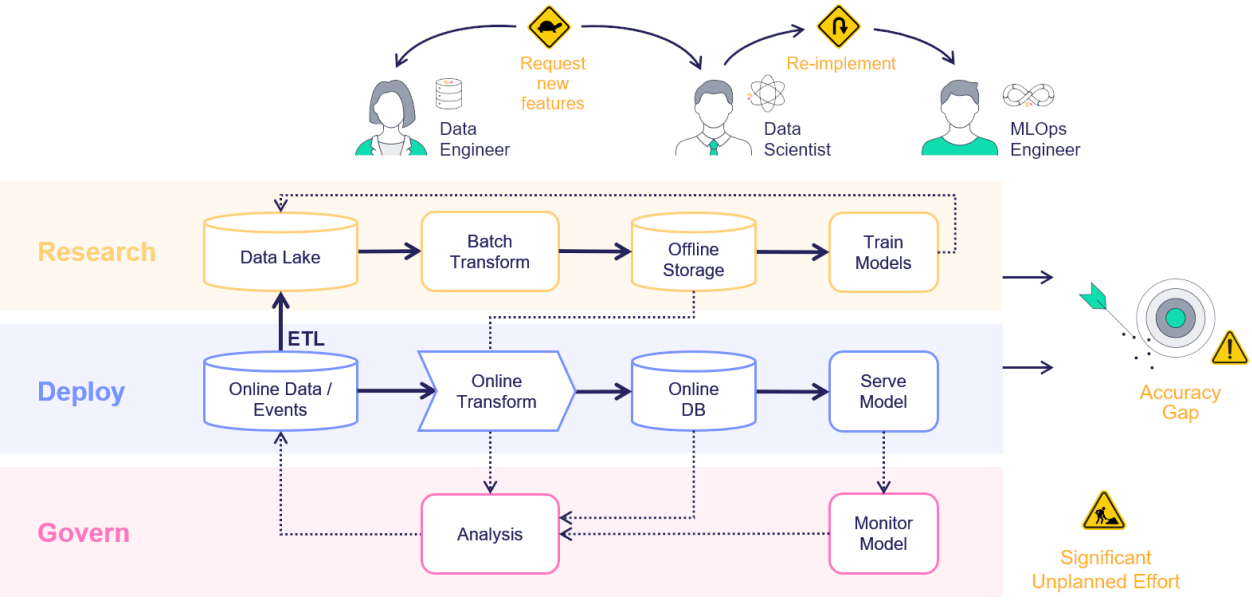
In this section

8.1 Feature store overview

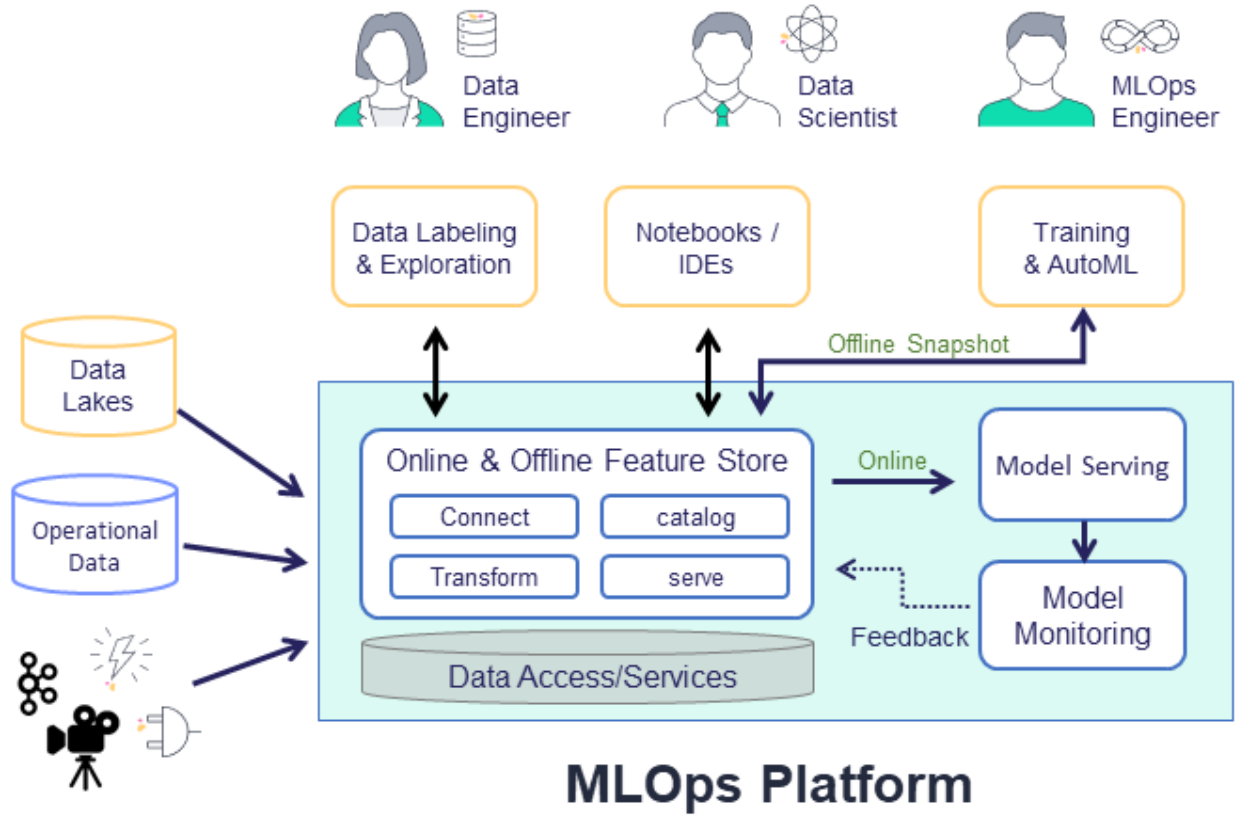
In machine-learning scenarios, generating a new feature, called feature engineering, takes a tremendous amount of work. The same features must be used both for training, based on historical data, and for the model prediction based on the online or real-time data. This creates a significant additional engineering effort, and leads to model inaccuracy when the online and offline features do not match. Furthermore, monitoring solutions must be built to track features and results and send alerts of data or model drift.

Consider a scenario in which you train a model and one of its features is a comparison of the current amount to the average amount spent during the last 3 months by the same person. Creating such a feature is easy when you have the full dataset in training, but in serving, this feature must be calculated in an online manner. The "brute-force" way to address this is to have an ML engineer create an online pipeline that reimplements all the feature calculations done in

the offline process. This is not just time-consuming and error-prone, but very difficult to maintain over time, and results in a lengthy deployment time. This is exacerbated when having to deal with thousands of features with an increasing number of data engineers and data scientists that are creating and using the features.



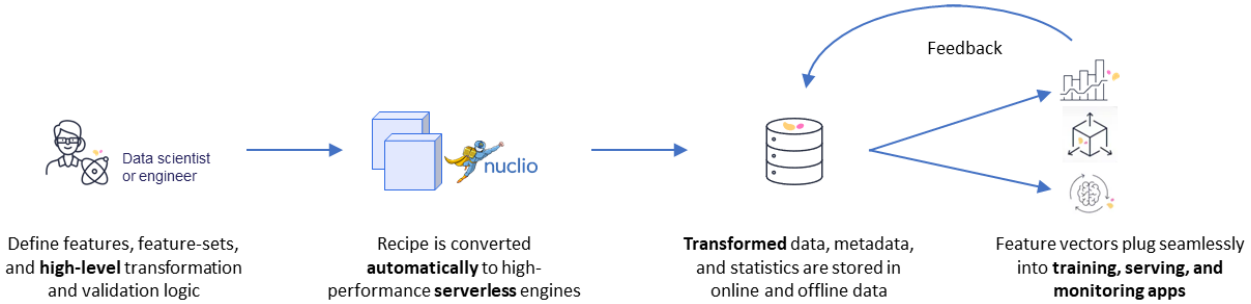
With MLRun's feature store you can easily define features during the training, that are deployable to serving, without having to define all the "glue" code. You simply create the necessary building blocks to define features and integration, with offline and online storage systems to access the features.



The feature store is comprised of the following:

- **Feature** — In machine-learning, a feature is an individual measurable property or characteristic of a phenomenon being observed. This can be raw data (e.g., transaction amount, image pixel, etc.) or a calculation derived from one or more other features (e.g., deviation from average, pattern on image, etc.).
- ***Feature sets*** — A grouping of features that are ingested together and stored in a logical group. Feature sets take data from offline or online sources, build a list of features through a set of transformations, and store the resulting features, along with the associated metadata and statistics. For example, a transaction may be grouped by the ID of a person performing the transfer or by the device identifier used to perform the transaction. You can also define in the timestamp source in the feature set, and ingest data into a feature set.
- **Execution** — A set of operations performed on the data while it is ingested. The graph contains steps that represent data sources and targets, and can also contain steps that transform and enrich the data that is passed through the feature set. For a deeper dive, see *Feature set transformations*.
- ***Feature vectors*** — A set of features, taken from one or more feature sets. The feature vector is defined prior to model training and serves as the input to the model training process. During model serving, the feature values in the vector are obtained from an online service.

8.1.1 How the feature store works



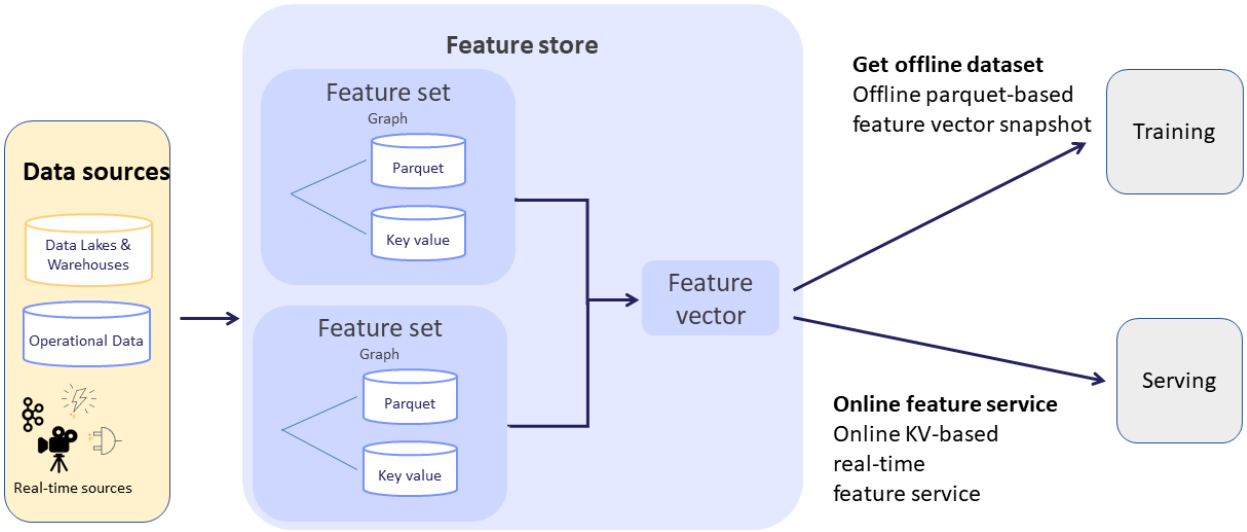
The common flow when working with the feature store is to first define the feature set with its source, transformation graph, and targets. MLRun's robust transformation engine performs complex operations with just a few lines of Python code. To test the execution process, call the `infer` method with a sample DataFrame. This runs all operations in memory without storing the results. Once the graph is defined, it's time to ingest the data.

You can ingest data directly from a DataFrame, by calling the feature set `ingest` method. You can also define an ingestion process that runs as a Kubernetes job. This is useful if there is a large ingestion process, or if there is a recurrent ingestion and you want to schedule the job.

MLRun can also leverage Nuclio to perform real-time ingestion by calling the `deploy_ingestion_service` function. This means that during serving you can update feature values, and not just read them. For example, you can update a sliding window aggregation as part of a model serving process.

The next step is to define the feature vector. Call the `get_offline_features` function to join together features across different feature sets.

8.1.2 Training and serving using the feature store



Next, extract a versioned **offline** static dataset for training, based on the parquet target defined in the feature sets. You can train a model with the feature vector data by providing the input in the form of `'store://feature-vectors/{project}/{feature_vector_name}'`.

Training functions generate models and various model statistics. Use MLRun's auto logging capabilities to store the models along with all the relevant data, metadata and measurements.

MLRun can apply all the MLOps functionality by using the framework specific `apply_mlrun()` method, which manages the training process and automatically logs all the framework specific model details, data, metadata and metrics.

The training job automatically generates a set of results and versioned artifacts (run `train_run.outputs` to view the job outputs).

For serving, once you validate the feature vector, use the **online** feature service, based on the nosql target defined in the feature set for real-time serving. For serving, you define a serving class derived from `mlrun.serving.V2ModelServer`. In the class `load` method, call the `get_online_feature_service` function with the vector name, which returns a feature service object. In the class `preprocess` method, call the feature service `get` method to get the values of those features.

Using this feature store centric process, using one computation graph definition for a feature set, you receive an automatic online and offline implementation for the feature vectors, with data versioning both in terms of the actual graph that was used to calculate each data point, and the offline datasets that were created to train each model.

See more information in *training with the feature store* and *Serving with the feature store*.

8.2 Feature sets

In MLRun, a group of features can be ingested together and stored in logical group called feature set. Feature sets take data from offline or online sources, build a list of features through a set of transformations, and store the resulting features along with the associated metadata and statistics. A feature set can be viewed as a database table with multiple material implementations for batch and real-time access, along with the data pipeline definitions used to produce the features.

The feature set object contains the following information:

- **Metadata** — General information which is helpful for search and organization. Examples are project, name, owner, last update, description, labels, etc.

- **Key attributes** — Entity (the join key), timestamp key (optional), label column.
- **Features** — The list of features along with their schema, metadata, validation policies and statistics.
- **Source** — The online or offline data source definitions and ingestion policy (file, database, stream, http endpoint, etc.).
- **Transformation** — The data transformation pipeline (e.g. aggregation, enrichment etc.).
- **Target stores** — The type (i.e. parquet/csv or key value), location and status for the feature set materialized data.
- **Function** — The type (storey, pandas, spark) and attributes of the data pipeline serverless functions.

In this section

- *Create a Feature Set*
- *Add transformations*
- *Simulate and debug the data pipeline with a small dataset*

See also *Ingest data using the feature store*

8.2.1 Create a feature set

Create a new `FeatureSet` with the base definitions:

- **name**—The feature set name is a unique name within a project.
- **entities**—Each feature set must be associated with one or more index column. When joining feature sets, the entity is used as the key column.
- **timestamp_key**—(optional) Used for specifying the time field when joining by time.
- **engine**—The processing engine type:
  - Spark
  - pandas
  - storey (some advanced functionalities are in the Beta state)

Example:

```
#Create a basic feature set example
stocks_set = FeatureSet("stocks", entities=[Entity("ticker")])
```

To learn more about Feature Sets go to `FeatureSet`.

Note

Feature sets can also be created in the UI. To create a feature set:

1. Select a project and press **Feature store**, then press **Create Set**.
2. After completing the form, press **Save and Ingest** to start the process, or **Save** to save the set for later ingestion.

8.2.2 Add transformations

Define the data processing steps using a transformations graph (DAG).

A feature set data pipeline takes raw data from online or offline sources and transforms it to meaningful features. The MLRun feature store supports three processing engines (storey, pandas, spark) that can run in the client (e.g. Notebook) for interactive development or in elastic serverless functions for production and scale.

The data pipeline is defined using MLRun graph (DAG) language. Graph steps can be pre-defined operators (such as aggregate, filter, encode, map, join, impute, etc.) or custom python classes/functions. Read more about the graph in Real-time serving pipelines (graphs).

The `pandas` and `spark` engines are good for simple batch transformations, while the `storey` stream processing engine (the default engine) can handle complex workflows and real-time sources.

The results from the transformation pipeline are stored in one or more material targets. Data for offline access, such as training, is usually stored in Parquet files. Data for online access such as serving is stored in the Iguazio NoSQL DB (`NoSqlTarget`). You can use the default targets or add/replace with additional custom targets. See Target stores(#target-stores).

Graph example (storey engine):

```
import mlrun.feature_store as fstore
feature_set = fstore.FeatureSet("measurements", entities=[Entity(key)], timestamp_key=
↪"timestamp")
# Define the computational graph including the custom functions
feature_set.graph.to(DropColumns(drop_columns))\
                 .to(RenameColumns(mapping={'bad': 'bed'}))
feature_set.add_aggregation('hr', ['avg'], ["1h"])
feature_set.plot()
fstore.ingest(feature_set, data_df)
```

Graph example (pandas engine):

```
def myfunc1(df, context=None):
    df = df.drop(columns=["exchange"])
    return df

stocks_set = fstore.FeatureSet("stocks", entities=[Entity("ticker")], engine="pandas")
stocks_set.graph.to(name="s1", handler="myfunc1")
df = fstore.ingest(stocks_set, stocks_df)
```

The graph steps can use built-in transformation classes, simple python classes, or function handlers.

See more details in Feature set transformations.

8.2.3 Simulate and debug the data pipeline with a small dataset

During the development phase it's pretty common to check the feature set definition and to simulate the creation of the feature set before ingesting the entire dataset, since ingesting the entire feature set can take time. This allows you to get a preview of the results (in the returned dataframe). The simulation method is called `infer`. It infers the source data schema as well as processing the graph logic (assuming there is one) on a small subset of data. The infer operation also learns the feature set schema and does statistical analysis on the result by default.

```
df = fstore.preview(quotes_set, quotes)
```

(continues on next page)

(continued from previous page)

```
# print the featue statistics
print(quotes_set.get_stats_table())
```

8.3 Feature set transformations

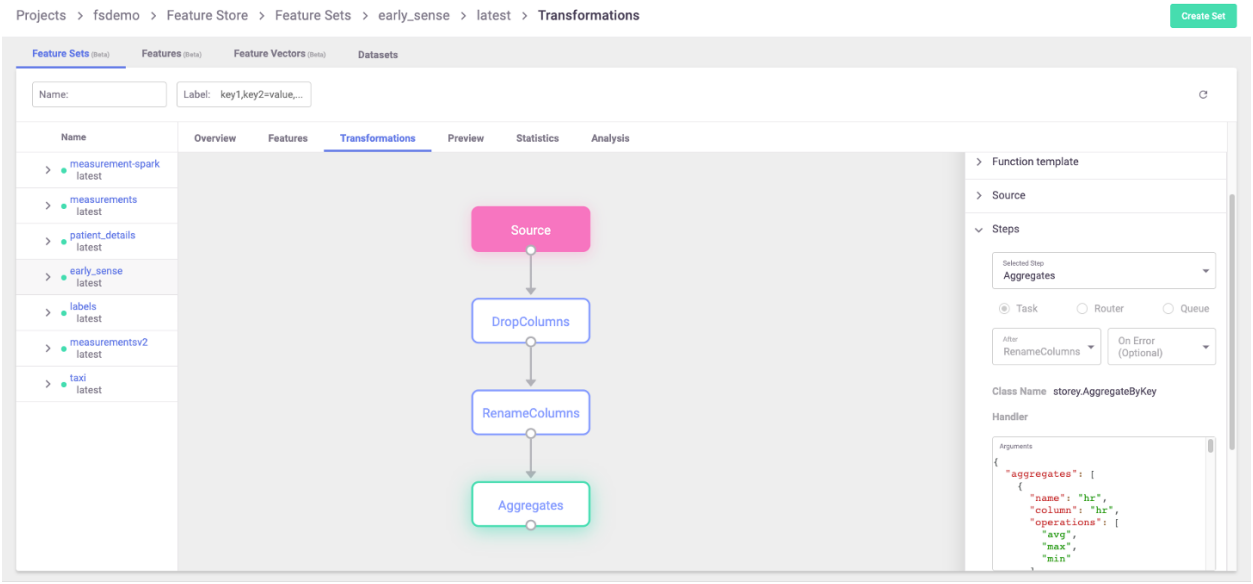
A feature set contains an execution graph of operations that are performed when data is ingested, or when simulating data flow for inferring its metadata. This graph utilizes MLRun's *Real-time serving pipelines (graphs)*.

The graph contains steps that represent data sources and targets, and may also contain steps whose purpose is transformations and enrichment of the data passed through the feature set. These transformations can be provided in one of three ways:

- ***Aggregations*** — MLRun supports adding aggregate features to a feature set through the `add_aggregation()` function.
- ***Built-in transformations*** — MLRun is equipped with a set of transformations provided through the `storey.transformations` package. These transformations can be added to the execution graph to perform common operations and transformations.
- ***Custom transformations*** — You can extend the built-in functionality by adding new classes that perform any custom operation and use them in the serving graph.

Once a feature-set is created, its internal execution graph can be observed by calling the feature-set's `plot()` function, which generates a `graphviz` plot based on the internal graph. This is very useful when running within a Jupyter notebook, and produces a graph such as the following example:

This plot shows various transformations and aggregations being used as part of the feature-set processing, as well as the targets where results are saved to (in this case two targets). Feature-sets can also be observed in the MLRun UI, where the full graph can be seen and specific step properties can be observed:



For a full end-to-end example of feature-store and usage of the functionality described in this page, refer to the feature store example.

In this section

- *Aggregations*
- *Built-in transformations*
- *Custom transformations*

8.3.1 Aggregations

Aggregations, being a common tool in data preparation and ML feature engineering, are available directly through the MLRun *FeatureSet* class. These transformations allow adding a new feature to the feature-set that is created by performing some aggregate function over feature's values within a time-based sliding window.

For example, if a feature-set contains stock trading data including the specific bid price for each bid at any given time, you could introduce aggregate features that show the minimal and maximal bidding price over all the bids in the last hour, per stock ticker (which is the entity in question). To do that, use the code:

```
import mlrun.feature_store as fstore
# create a new feature set
quotes_set = fstore.FeatureSet("stock-quotes", entities=[fstore.Entity("ticker")])
quotes_set.add_aggregation("bid", ["min", "max"], ["1h"], "10m")
```

Once this is executed, the feature-set has new features introduced, with their names produced from the aggregate parameters, using this format: `{column}_{operation}_{window}`. Thus, the example above generates two new features: `bid_min_1h` and `bid_max_1h`. If the function gets an optional `name` parameter, features are produced in `{name}_{operation}_{window}` format. If the `name` parameter is not specified, features are produced in `{column_name}_{operation}_{window}` format. These features can then be fed into predictive models or be used for additional processing and feature generation.

Notes

- Internally, the graph step that is created to perform these aggregations is named `"Aggregates"`. If more than one aggregation steps are needed, a unique name must be provided to each, using the `state_name` parameter.
- The timestamp column must be part of the feature set definition (for aggregation).

Aggregations that are supported using this function are:

- `count`
- `sum`
- `sqr` (sum of squares)
- `max`
- `min`
- `first`
- `last`
- `avg`
- `stdvar`
- `stddev`

For a full documentation of this function, see the *add_aggregation()* documentation.

8.3.2 Built-in transformations

MLRun, and the associated `storey` package, have a built-in library of transformation functions that can be applied as steps in the feature-set's internal execution graph. In order to add steps to the graph, it should be referenced from the *`FeatureSet`* object by using the *`graph`* property. Then, new steps can be added to the graph using the functions in *`storey.transformations`* (follow the link to browse the documentation and the list of existing functions). The transformations are also accessible directly from the `storey` module.

See the built-in steps.

Note

Internally, MLRun makes use of functions defined in the `storey` package for various purposes. When creating a feature-set and configuring it with sources and targets, what MLRun does behind the scenes is to add steps to the execution graph that wraps methods and classes, which perform the actions. When defining an async execution graph, `storey` classes are used. For example, when defining a Parquet data-target in MLRun, a graph step is created that wraps storey's `WriteToParquet()` function.

To use a function:

1. Access the graph from the feature-set object, using the *`graph`* property.
2. Add steps to the graph using the various graph functions, such as `to()`. The function object passed to the step should point at the transformation function being used.

The following is an example for adding a simple `filter` to the graph, that drops any bid which is lower than 50USD:

```
quotes_set.graph.to("storey.Filter", "filter", _fn="(event['bid'] > 50)")
```

In the example above, the parameter `_fn` denotes a callable expression that is passed to the `storey.Filter` class as the parameter `fn`. The callable parameter can also be a Python function, in which case there's no need for parentheses around it. This call generates a step in the graph called `filter` that calls the expression provided with the event being propagated through the graph as data is fed to the feature-set.

8.3.3 Custom transformations

When a transformation is needed that is not provided by the built-in functions, new classes that implement transformations can be created and added to the execution graph. Such classes should extend the *`MapClass`* class, and the actual transformation should be implemented within their `do()` function, which receives an event and returns the event after performing transformations and manipulations on it. For example, consider the following code:

```
class MyMap(MapClass):
    def __init__(self, multiplier=1, **kwargs):
        super().__init__(**kwargs)
        self._multiplier = multiplier

    def do(self, event):
        event["multi"] = event["bid"] * self._multiplier
        return event
```

The `MyMap` class can then be used to construct graph steps, in the same way as shown above for built-in functions:

```
quotes_set.graph.add_step("MyMap", "multi", after="filter", multiplier=3)
```

This uses the `add_step` function of the graph to add a step called `multi` utilizing `MyMap` after the `filter` step that was added previously. The class is initialized with a multiplier of 3.

8.4 Creating and using feature vectors

You can define a group of features from different feature sets as a *FeatureVector*.
Feature vectors are used as an input for models, allowing you to define the feature vector once, and in turn create and track the *datasets* created from it or the online manifestation of the vector for real-time prediction needs.

The feature vector handles all the merging logic for you using an `asof merge` type merge that accounts for both the time and the entity. It ensures that all the latest relevant data is fetched, without concerns about "seeing the future" or other types of common time related errors.

In this section

8.4.1 Creating a feature vector

The feature vector object holds the following information:

- Name — the feature vector's name as will be later addressed in the store reference `store://feature_vectors/<project>/<feature-vector-name>` and the UI (after saving the vector).
- Description — a string description of the feature vector.
- Features — a list of features that comprise the feature vector.
  The feature list is defined by specifying the `<feature-set>.<feature-name>` for specific features or `<feature-set>.*` for all the feature set's features.
- Label feature — the feature that is the label for this specific feature vector, as a `<feature-set>.<feature-name>` string specification.

Example of creating a feature vector:

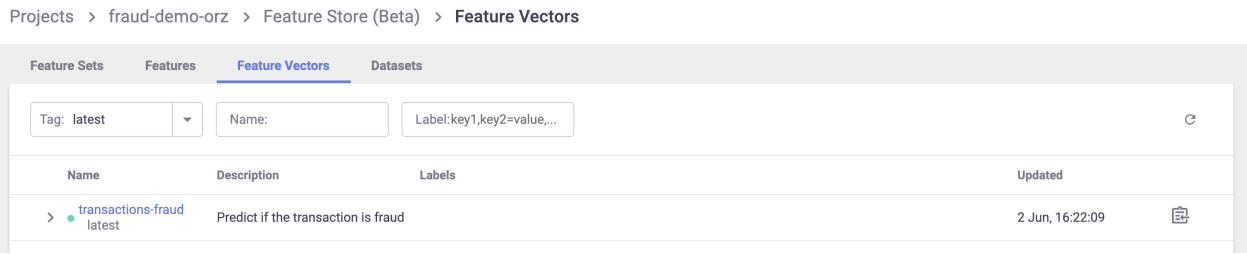
```
import mlrun.feature_store as fstore

# Feature vector definitions
feature_vector_name = 'example-fv'
feature_vector_description = 'Example feature vector'
features = ['data_source_1.*',
            'data_source_2.feature_1',
            'data_source_2.feature_2',
            'data_source_3.*']
label_feature = 'label_source_1.label_feature'

# Feature vector creation
fv = fstore.FeatureVector(name=feature_vector_name,
                          features=features,
                          label_feature=label_feature,
                          description=feature_vector_description)

# Save the feature vector in the MLRun DB
```

(continues on next page)

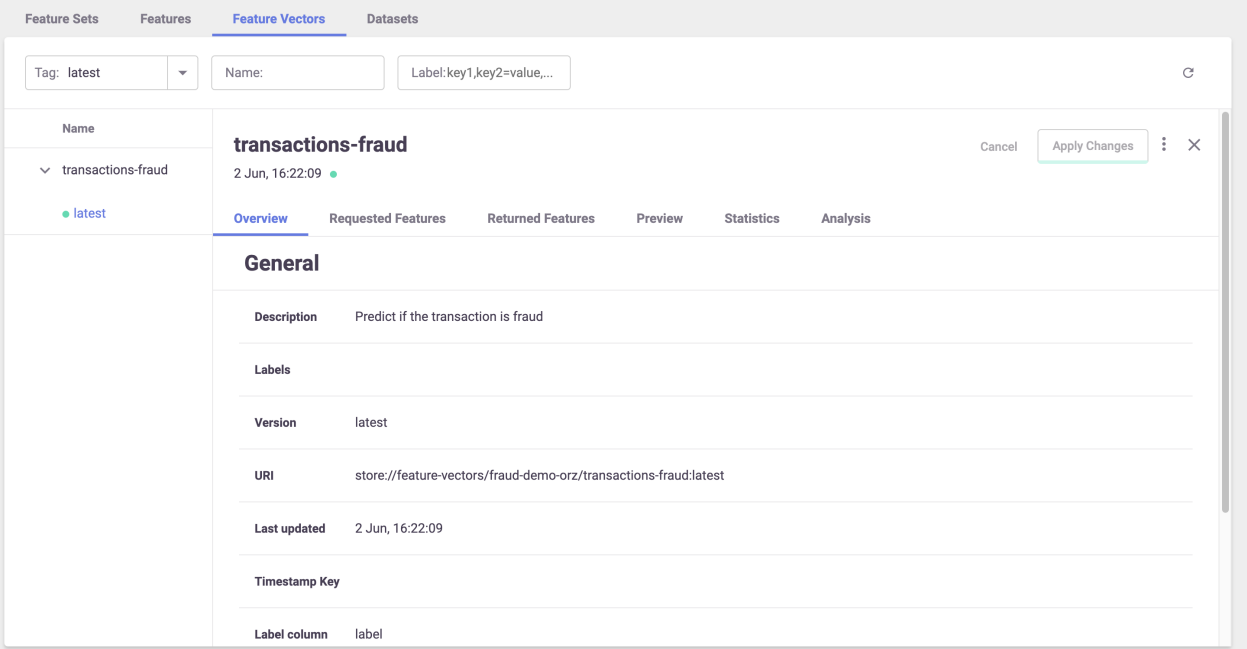
(continued from previous page)

```
# so it will could be referenced by the `store://`
# and show in the UI
fv.save()
```

After saving the feature vector, it appears in the UI:



You can also view some metadata about the feature vector, including all the features, their types, a preview and statistics:



8.4.2 Using a feature vector

After a feature vector is saved, it can be used to create both offline (static) datasets and online (real-time) instances to supply as input to a machine learning model.

Creating an offline feature vector

Use the feature store's *get_offline_features()* function to produce a `dataset` from the feature vector. It creates the dataset (asynchronously if possible), saves it to the requested target, and returns a *OfflineVectorResponse*. Due to the async nature of this action, the response object contains an `fv_response.status` indicator that, once completed, could be directly turned into a `dataframe`, `parquet` or a `csv`.

`get_offline_features` expects to receive:

- **feature_vector** — a feature vector store reference or object.
- **entity_rows** — an optional dataframe that the features will be joined to.
  Defaults to the first feature set defined in the features vector's features list, and acts as the base for the vector's joins.
- **entity_timestamp_column** — an optional specific timestamp column (from the defined features) to act as the base timestamp column.
  Defaults to the base feature set's timestamp entity.
- **target** — a Feature Store target to write the results to.
  Defaults to return as a return value to the caller.
- **run_config** — an optional function or a *RunConfig* to run the feature vector creation process in a remote function.
- **drop_columns** — a list of columns to drop from the resulting feature vector. Optional.
- **start_time** — datetime, low limit of time needed to be filtered. Optional.
- **end_time** — datetime, high limit of time needed to be filtered. Optional.

You can add a time-based filter condition when running `get_offline_feature` with a given vector. You can also filter with the query argument on all the other features as relevant.

Here's an example of a new dataset from a parquet target:

```
# Import the Parquet Target, so you can build your dataset from a parquet file
from mlrun.datastore.targets import ParquetTarget

# Get offline feature vector based on vector and parquet target
offline_fv = fstore.get_offline_features(feature_vector_name, target=ParquetTarget())

# Return dataset
dataset = offline_fv.to_dataframe()
```

Once an offline feature vector is created with a static target (such as *ParquetTarget()*) the reference to this dataset is saved as part of the feature vector's metadata and can now be referenced directly through the store as a function input using `store://feature-vectors/{project}/{feature_vector_name}`.

For example:

```
fn = mlrun.import_function('hub://sklearn-classifier').apply(auto_mount())

# Define the training task, including the feature vector and label
task = mlrun.new_task('training',
                      inputs={'dataset': f'store://feature-vectors/{project}/{feature_
↪vector_name}'},
                      params={'label_column': 'label'}
                     )
```

(continues on next page)

(continued from previous page)

```
# Run the function
run = fn.run(task)
```

You can see a full example of using the offline feature vector to create an ML model in part 2 of the end-to-end demo.

Creating an online feature vector

The online feature vector provides real-time feature vectors to the model using the latest data available.

First create an `Online Feature Service` using *get_online_feature_service()*. Then feed the `Entity` of the feature vector to the service and receive the latest feature vector.

To create the *OnlineVectorService* you only need to pass it the feature vector's store reference.

```
import mlrun.feature_store as fstore

# Create the Feature Vector Online Service
feature_vector = 'store://feature-vectors/{project}/{feature_vector_name}'
svc = fstore.get_online_feature_service(feature_vector)
```

The online feature service supports value imputing (substitute NaN/Inf values with statistical or constant value). You can set the `impute_policy` parameter with the imputing policy, and specify which constant or statistical value will be used instead of NaN/Inf value. This can be defined per column or for all the columns (`"*"`). The replaced value can be a fixed number for constants or `$mean`, `$max`, `$min`, `$std`, `$count` for statistical values. `"*"` is used to specify the default for all features, for example:

```
svc = fstore.get_online_feature_service(feature_vector, impute_policy={"*": "$mean", "age
↪": 33})
```

To use the online feature service you need to supply a list of entities you want to get the feature vectors for. The service returns the feature vectors as a dictionary of `{<feature-name>: <feature-value>}` or simply a list of values as numpy arrays.

For example:

```
# Define the wanted entities
entities = [{<feature-vector-entity-column-name>: <entity>}]

# Get the feature vectors from the service
svc.get(entities)
```

The `entities` can be a list of dictionaries as shown in the example, or a list of lists where the values in the internal list correspond to the entity values (e.g. `entities = [["Joe"], ["Mike"]]`). The `.get()` method returns a dict by default. If you want to return an ordered list of values, set the `as_list` parameter to `True`. The list input is required by many ML frameworks and this eliminates additional glue logic.

See a full example of using the online feature service inside a serving function in part 3 of the end-to-end demo.

8.5 Feature store end-to-end demo

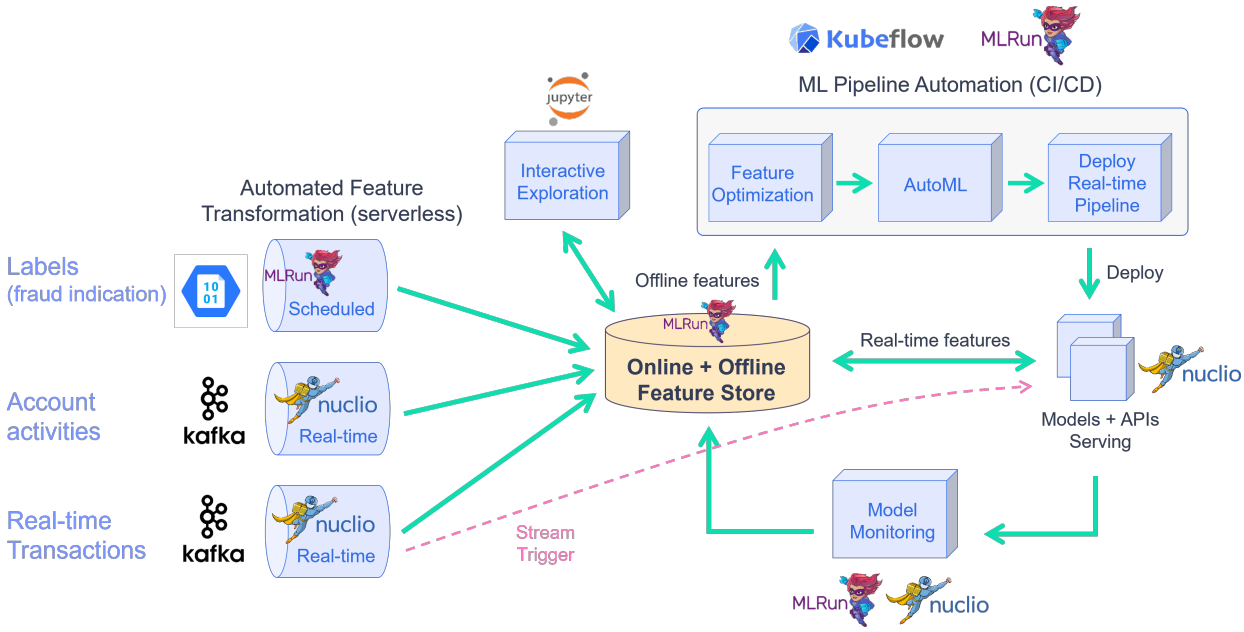
This demo shows the usage of MLRun and the feature store:

- **Data ingestion & preparation**
- **Model training & testing**
- **Model serving**
- **Building an automated ML pipeline**

Fraud prevention, specifically, is a challenge since it requires processing raw transactions and events in real-time and being able to quickly respond and block transactions before they occur. Consider, for example, a case where you would like to evaluate the average transaction amount. When training the model, it is common to take a DataFrame and just calculate the average. However, when dealing with real-time/online scenarios, this average has to be calculated incrementally.

This demo illustrates how to **Ingest** different data sources to the **Feature Store**. Specifically, it covers two types of data:

- **Transactions**: Monetary activity between two parties to transfer funds.
- **Events**: Activity performed by a party, such as login or password change.



The demo walks through creation of an ingestion pipeline for each data source with all the needed preprocessing and validation. It runs the pipeline locally within the notebook and then launches a real-time function to **ingest live data** or schedule a cron to run the task when needed.

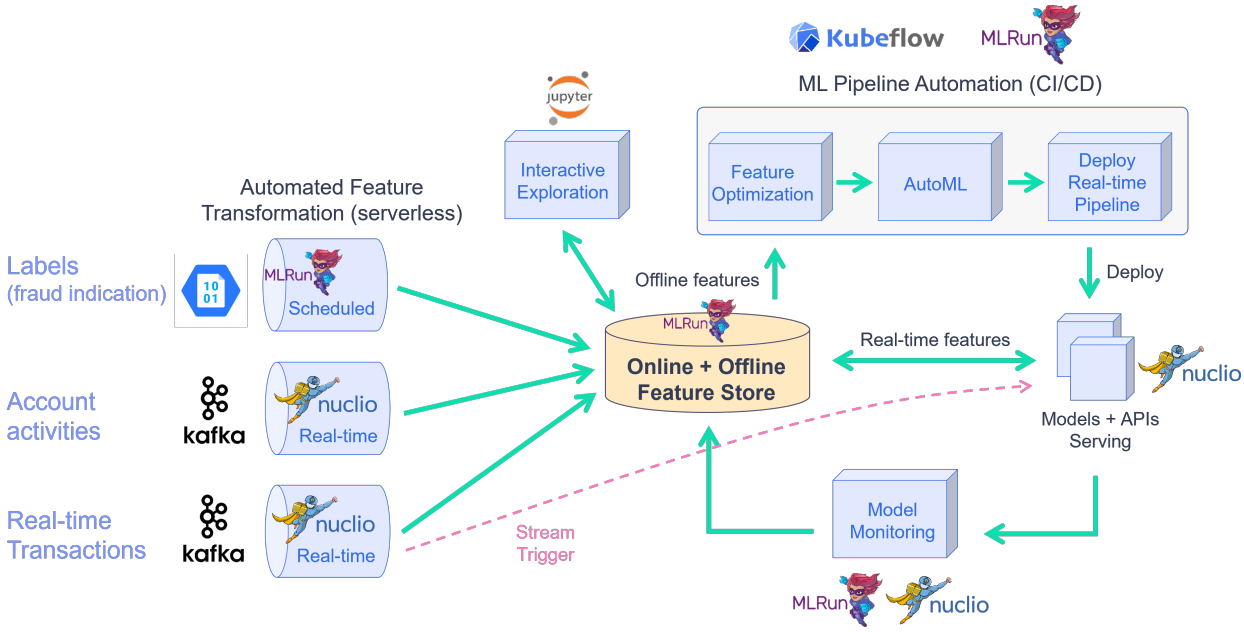
Following the ingestion, you create a feature vector, select the most relevant features and create a final model. Then you deploy the model and showcase the feature vector and model serving.

8.5.1 Part 1: Data Ingestion

This demo showcases financial fraud prevention. It uses the MLRun feature store to define complex features that help identify fraud.

Fraud prevention is a special challenge since it requires processing raw transaction and events in real-time and being able to quickly respond and block transactions before they occur.

To address this, you'll create a development pipeline and a production pipeline. Both pipelines share the same feature engineering and model code, but serve data very differently. Furthermore, MLRun automates the data and model monitoring process, drift identification, and trigger retraining in a CI/CD pipeline. This process is described in the diagram below:



The raw data is described as follows:

TRANSACTIONS			USER EVENTS	
age	age group value 0-6. Some values are marked as U for unknown		**source**	The party/entity related to the event
gender	A character to define the age		**event**	event, such as login or password change
zipcodeOri	ZIP code of the person originating the transaction		**timestamp**	The date and time of the event
zipMerchant	ZIP code of the merchant receiving the transaction			
category	category of the transaction (e.g., transportation, food, etc.)			
amount	the total amount of the transaction			
fraud	whether the transaction is fraudulent			
timestamp	the date and time in which the transaction took place			
source	the ID of the party/entity performing the transaction			
target	the ID of the party/entity receiving the transaction			
device	the device ID used to perform the transaction			

This notebook introduces how to **Ingest** different data sources to the **Feature Store**.

The following FeatureSets are created:

- **Transactions**: Monetary transactions between a source and a target.
- **Events**: Account events such as account login or a password change.
- **Label**: Fraud label for the data.

By the end of this tutorial you'll know how to:

- Create an ingestion pipeline for each data source.
- Define preprocessing, aggregation, and validation of the pipeline.
- Run the pipeline locally within the notebook.
- Launch a real-time function to ingest live data.
- Schedule a cron to run the task when needed.

```
project_name = 'fraud-demo'
```

```
import mlrun

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name, context="./", user_project=True)
```

```
> 2022-03-16 05:45:07,703 [info] loaded project fraud-demo from MLRun DB
```

Step 1 - Fetch, process and ingest the datasets

1.1 - Transactions

Transactions

```
# Helper functions to adjust the timestamps of our data
# while keeping the order of the selected events and
# the relative distance from one event to the other

def date_adjustment(sample, data_max, new_max, old_data_period, new_data_period):
    '''
        Adjust a specific sample's date according to the original and new time periods
    '''
    sample_dates_scale = ((data_max - sample) / old_data_period)
    sample_delta = new_data_period * sample_dates_scale
    new_sample_ts = new_max - sample_delta
    return new_sample_ts

def adjust_data_timespan(dataframe, timestamp_col='timestamp', new_period='2d', new_max_
↪date_str='now'):
    '''
        Adjust the dataframe timestamps to the new time period
    '''
    # Calculate old time period
    data_min = dataframe.timestamp.min()
    data_max = dataframe.timestamp.max()
    old_data_period = data_max-data_min

    # Set new time period
    new_time_period = pd.Timedelta(new_period)
    new_max = pd.Timestamp(new_max_date_str)
    new_min = new_max-new_time_period
    new_data_period = new_max-new_min

    # Apply the timestamp change
    df = dataframe.copy()
    df[timestamp_col] = df[timestamp_col].apply(lambda x: date_adjustment(x, data_max,␣
↪new_max, old_data_period, new_data_period))
    return df
```

```
import pandas as pd

# Fetch the transactions dataset from the server
transactions_data = pd.read_csv('https://s3.wasabisys.com/iguazio/data/fraud-demo-mlrun-
↪fs-docs/data.csv', parse_dates=['timestamp'], nrows=500)

# Adjust the samples timestamp for the past 2 days
transactions_data = adjust_data_timespan(transactions_data, new_period='2d')

# Preview
transactions_data.head(3)
```

```
   step age gender  zipcodeOri  zipMerchant           category  amount  fraud  \
0     0   4      M       28007        28007  es_transportation    4.55      0
1     0   2      M       28007        28007  es_transportation   39.68      0
2     0   4      F       28007        28007  es_transportation   26.89      0

                       timestamp       source       target  \
0 2022-03-15 16:13:54.851486383  C1093826151   M348934600
1 2022-03-14 09:21:09.710448366   C352968107   M348934600
2 2022-03-15 22:41:20.666966912  C2054744914  M1823072687

                             device
0  f802e61d76564b7a89a83adcdfa573da
1  38ef7fc3eb7442c8ae64579a483f1d2b
2  7a851d0758894078b5846851ae32d5e3
```

Transactions - create a feature set and preprocessing pipeline

Create the feature set (data pipeline) definition for the **credit transaction processing** that describes the offline/online data transformations and aggregations. The feature store automatically adds an offline `parquet` target and an online `NoSQL` target by using `set_targets()`.

The data pipeline consists of:

- **Extracting** the data components (hour, day of week)
- **Mapping** the age values
- **One hot encoding** for the transaction category and the gender
- **Aggregating** the amount (avg, sum, count, max over 2/12/24 hour time windows)
- **Aggregating** the transactions per category (over 14 days time windows)
- **Writing** the results to **offline** (Parquet) and **online** (NoSQL) targets

```
# Import MLRun's Feature Store
import mlrun.feature_store as fstore
from mlrun.feature_store.steps import OneHotEncoder, MapValues, DateExtractor
```

```
# Define the transactions FeatureSet
transaction_set = fstore.FeatureSet("transactions",
                                    entities=[fstore.Entity("source")],
                                    timestamp_key='timestamp',
                                    description="transactions feature set")
```

```
# Define and add value mapping
main_categories = ["es_transportation", "es_health", "es_otherservices",
       "es_food", "es_hotelservices", "es_barsandrestaurants",
       "es_tech", "es_sportsandtoys", "es_wellnessandbeauty",
       "es_hyper", "es_fashion", "es_home", "es_contents",
       "es_travel", "es_leisure"]

# One Hot Encode the newly defined mappings
one_hot_encoder_mapping = {'category': main_categories,
```

(continues on next page)

(continued from previous page)

```
                            'gender': list(transactions_data.gender.unique())}

# Define the graph steps
transaction_set.graph\
    .to(DateExtractor(parts = ['hour', 'day_of_week'], timestamp_col = 'timestamp'))\
    .to(MapValues(mapping={'age': {'U': '0'}}, with_original_features=True))\
    .to(OneHotEncoder(mapping=one_hot_encoder_mapping))


# Add aggregations for 2, 12, and 24 hour time windows
transaction_set.add_aggregation(name='amount',
                                column='amount',
                                operations=['avg','sum', 'count','max'],
                                windows=['2h', '12h', '24h'],
                                period='1h')


# Add the category aggregations over a 14 day window
for category in main_categories:
    transaction_set.add_aggregation(name=category,column=f'category_{category}',
                                    operations=['count'], windows=['14d'], period='1d')

# Add default (offline-parquet & online-nosql) targets
transaction_set.set_targets()

# Plot the pipeline so we can see the different steps
transaction_set.plot(rankdir="LR", with_targets=True)
```

```
<graphviz.dot.Digraph at 0x7f88af0f9cd0>
```

Transactions - ingestion

```
# Ingest the transactions dataset through the defined pipeline
transactions_df = fstore.ingest(transaction_set, transactions_data,
                 infer_options=fstore.InferOptions.default())

transactions_df.head(3)
```

```
persist count = 0
persist count = 100
persist count = 200
persist count = 300
persist count = 400
persist count = 500
persist count = 600
persist count = 700
persist count = 800
persist count = 900
persist count = 1000
```

```
             amount_count_2h  amount_count_12h  amount_count_24h  \
source
C1093826151              1.0               1.0               1.0
C352968107               1.0               1.0               1.0
C2054744914              1.0               1.0               1.0

             amount_max_2h  amount_max_12h  amount_max_24h  amount_sum_2h  \
source
C1093826151           4.55            4.55            4.55           4.55
C352968107           39.68           39.68           39.68          39.68
C2054744914          26.89           26.89           26.89          26.89

             amount_sum_12h  amount_sum_24h  amount_avg_2h  ...  \
source                                                      ...
C1093826151            4.55            4.55           4.55  ...
C352968107            39.68           39.68          39.68  ...
C2054744914           26.89           26.89          26.89  ...

             category_es_contents  category_es_travel  category_es_leisure  \
source
C1093826151                     0                   0                    0
C352968107                      0                   0                    0
C2054744914                     0                   0                    0

             amount  fraud                     timestamp       target  \
source
C1093826151    4.55      0 2022-03-15 16:13:54.851486383   M348934600
C352968107    39.68      0 2022-03-14 09:21:09.710448366   M348934600
C2054744914   26.89      0 2022-03-15 22:41:20.666966912  M1823072687

                                  device  timestamp_hour  \
source
C1093826151  f802e61d76564b7a89a83adcdfa573da              16
C352968107   38ef7fc3eb7442c8ae64579a483f1d2b               9
C2054744914  7a851d0758894078b5846851ae32d5e3              22

             timestamp_day_of_week
source
C1093826151                      1
C352968107                       0
C2054744914                      1

[3 rows x 57 columns]
```

1.2 - User events

User events - fetching

```
# Fetch the user_events dataset from the server
user_events_data = pd.read_csv('https://s3.wasabisys.com/iguazio/data/fraud-demo-mlrun-
↪fs-docs/events.csv',
                               index_col=0, quotechar="\'", parse_dates=['timestamp'],␣
↪nrows=500)

# Adjust to the last 2 days to see the latest aggregations in our online feature vectors
user_events_data = adjust_data_timespan(user_events_data, new_period='2d')

# Preview
user_events_data.head(3)
```

```
        source           event                       timestamp
0  C1974668487  details_change 2022-03-15 15:03:17.518565985
1  C1973547259           login 2022-03-15 18:05:50.652706656
2   C515668508           login 2022-03-15 14:37:49.845093748
```

User events - create a feature set and preprocessing pipeline

Define the events feature set. This is a fairly straightforward pipeline in which you only "one hot encode" the event categories and save the data to the default targets.

```
user_events_set = fstore.FeatureSet("events",
                            entities=[fstore.Entity("source")],
                            timestamp_key='timestamp',
                            description="user events feature set")
```

```
# Define and add value mapping
events_mapping = {'event': list(user_events_data.event.unique())}

# One Hot Encode
user_events_set.graph.to(OneHotEncoder(mapping=events_mapping))

# Add default (offline-parquet & online-nosql) targets
user_events_set.set_targets()

# Plot the pipeline so we can see the different steps
user_events_set.plot(rankdir="LR", with_targets=True)
```

```
<graphviz.dot.Digraph at 0x7f88ad6f1fd0>
```

User events - ingestion

```
# Ingestion of the newly created events feature set
events_df = fstore.ingest(user_events_set, user_events_data)
events_df.head(3)
```

```
persist count = 0
persist count = 100
persist count = 200
persist count = 300
persist count = 400
persist count = 500
```

```
             event_details_change  event_login  event_password_change  \
source
C1974668487                     1            0                      0
C1973547259                     0            1                      0
C515668508                      0            1                      0

                               timestamp
source
C1974668487 2022-03-15 15:03:17.518565985
C1973547259 2022-03-15 18:05:50.652706656
C515668508  2022-03-15 14:37:49.845093748
```

Step 2 - Create a labels dataset for model training

Label set - create a feature set

This feature set contains the label for the fraud demo, it is ingested directly to the default targets without any changes

```
def create_labels(df):
    labels = df[['fraud','source','timestamp']].copy()
    labels = labels.rename(columns={"fraud": "label"})
    labels['timestamp'] = labels['timestamp'].astype("datetime64[ms]")
    labels['label'] = labels['label'].astype(int)
    labels.set_index('source', inplace=True)
    return labels
```

```
# Define the "labels" feature set
labels_set = fstore.FeatureSet("labels",
                           entities=[fstore.Entity("source")],
                           timestamp_key='timestamp',
                           description="training labels",
                           engine="pandas")

labels_set.graph.to(name="create_labels", handler=create_labels)

# specify only Parquet (offline) target since its not used for real-time
labels_set.set_targets(['parquet'], with_defaults=False)
labels_set.plot(with_targets=True)
```

```
<graphviz.dot.Digraph at 0x7f88a3561f90>
```

Label set - ingestion

```
# Ingest the labels feature set
labels_df = fstore.ingest(labels_set, transactions_data)
labels_df.head(3)
```

```
             label               timestamp
source
C1093826151      0 2022-03-15 16:13:54.851
C352968107       0 2022-03-14 09:21:09.710
C2054744914      0 2022-03-15 22:41:20.666
```

Step 3 - Deploy a real-time pipeline

When dealing with real-time aggregation, it's important to be able to update these aggregations in real-time. For this purpose, you'll create live serving functions that update the online feature store of the `transactions` FeatureSet and `Events` FeatureSet.

Using MLRun's `serving` runtime, create a nuclio function loaded with the feature set's computational graph definition and an `HttpSource` to define the HTTP trigger.

Notice that the implementation below does not require any rewrite of the pipeline logic.

3.1 - Transactions

Transactions - deploy the feature set live endpoint

```
# Create iguazio v3io stream and transactions push API endpoint
transaction_stream = f'v3io:///projects/{project.name}/streams/transaction'
transaction_pusher = mlrun.datastore.get_stream_pusher(transaction_stream)
```

```
# Define the source stream trigger (use v3io streams)
# Define the `key` and `time` fields (extracted from the Json message).
source = mlrun.datastore.sources.StreamSource(path=transaction_stream , key_field='source
↪', time_field='timestamp')

# Deploy the transactions feature set's ingestion service over a real-time (Nuclio)␣
↪serverless function
# you can use the run_config parameter to pass function/service specific configuration
transaction_set_endpoint = fstore.deploy_ingestion_service(featureset=transaction_set,␣
↪source=source)
```

```
> 2022-03-16 05:45:43,035 [info] Starting remote function deploy
2022-03-16 05:45:43  (info) Deploying function
2022-03-16 05:45:43  (info) Building
2022-03-16 05:45:43  (info) Staging files and preparing base images
```

(continues on next page)

(continued from previous page)

```
2022-03-16 05:45:43  (warn) Python 3.6 runtime is deprecated and will soon not be␣
↪supported. Please migrate your code and use Python 3.7 runtime (`python:3.7`) or higher
2022-03-16 05:45:43  (info) Building processor image
2022-03-16 05:47:03  (info) Build complete
2022-03-16 05:47:08  (info) Function deploy complete
> 2022-03-16 05:47:08,835 [info] successfully deployed function: {'internal_invocation_
↪urls': ['nuclio-fraud-demo-admin-transactions-ingest.default-tenant.svc.cluster.
↪local:8080'], 'external_invocation_urls': ['fraud-demo-admin-transactions-ingest-fraud-
↪demo-admin.default-tenant.app.xtvtjecfcssi.iguazio-cd1.com/']}
```

Transactions - test the feature set HTTP endpoint

By defining the `transactions` feature set you can now use MLRun and Storey to deploy it as a live endpoint, ready to ingest new data!

Using MLRun's `serving` runtime, create a nuclio function loaded with the feature set's computational graph definition and an `HttpSource` to define the HTTP trigger.

```
import requests
import json

# Select a sample from the dataset and serialize it to JSON
transaction_sample = json.loads(transactions_data.sample(1).to_json(orient='records'))[0]
transaction_sample['timestamp'] = str(pd.Timestamp.now())
transaction_sample
```

```
{'step': 0,
 'age': '5',
 'gender': 'M',
 'zipcodeOri': 28007,
 'zipMerchant': 28007,
 'category': 'es_transportation',
 'amount': 2.19,
 'fraud': 0,
 'timestamp': '2022-03-16 05:47:08.884971',
 'source': 'C546957379',
 'target': 'M348934600',
 'device': '8ee3b24f2eb143759e938b6148da547c'}
```

```
# Post the sample to the ingestion endpoint
requests.post(transaction_set_endpoint, json=transaction_sample).text
```

```
'{"id": "c3f8a087-3932-45c0-8dcb-1fd8efabe681"}'
```

3.2 - User events

User events - deploy the feature set live endpoint

Deploy the events feature set's ingestion service using the feature set and all the previously defined resources.

```
# Create iguazio v3io stream and transactions push API endpoint
events_stream = f'v3io:///projects/{project.name}/streams/events'
events_pusher = mlrun.datastore.get_stream_pusher(events_stream)
```

```
# Define the source stream trigger (use v3io streams)
# Define the `key` and `time` fields (extracted from the Json message).
source = mlrun.datastore.sources.StreamSource(path=events_stream , key_field='source',␣
↪time_field='timestamp')

# Deploy the transactions feature set's ingestion service over a real-time (Nuclio)␣
↪serverless function
# you can use the run_config parameter to pass function/service specific configuration
events_set_endpoint = fstore.deploy_ingestion_service(featureset=user_events_set,␣
↪source=source)
```

```
> 2022-03-16 05:47:09,035 [info] Starting remote function deploy
2022-03-16 05:47:09  (info) Deploying function
2022-03-16 05:47:09  (info) Building
2022-03-16 05:47:09  (info) Staging files and preparing base images
2022-03-16 05:47:09  (warn) Python 3.6 runtime is deprecated and will soon not be␣
↪supported. Please migrate your code and use Python 3.7 runtime (`python:3.7`) or higher
2022-03-16 05:47:09  (info) Building processor image
```

User events - test the feature set HTTP endpoint

```
# Select a sample from the events dataset and serialize it to JSON
user_events_sample = json.loads(user_events_data.sample(1).to_json(orient='records'))[0]
user_events_sample['timestamp'] = str(pd.Timestamp.now())
user_events_sample
```

```
# Post the sample to the ingestion endpoint
requests.post(events_set_endpoint, json=user_events_sample).text
```

Done!

You've completed Part 1 of the data-ingestion with the feature store. Proceed to Part 2 to learn how to train an ML model using the feature store data.

8.5.2 Part 2: Training

This part shows how to use MLRun's **Feature Store** to easily define a **Feature Vector** and create the dataset you need to run the training process.
By the end of this tutorial you'll learn how to:

- Combine multiple data sources to a single Feature Vector
- Create training dataset
- Create a model using an MLRun Hub function

```
project_name = 'fraud-demo'
```

```
import mlrun

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name, context="./", user_project=True)
```

```
> 2021-09-19 17:59:27,165 [info] loaded project fraud-demo from MLRun DB
```

Step 1 - Create a feature vector

In this section you create the Feature Vector.
The Feature vector has a `name` so you can reference to it later via the URI or the serving function, and a list of `features` from the available FeatureSets. You can add a feature from a feature set by adding `<FeatureSet>.<Feature>` to the list, or add `<FeatureSet>.*` to add all the FeatureSet's available features.

By default, the first FeatureSet in the feature list acts as the spine, meaning that all the other features will be joined to it.
For example, in this instance the spine is the early_sense sensor data, so for each early_sense event we will create produce a row in the resulted Feature Vector.

```
# Define the list of features you will be using
features = ['transactions.amount_max_2h',
            'transactions.amount_sum_2h',
            'transactions.amount_count_2h',
            'transactions.amount_avg_2h',
            'transactions.amount_max_12h',
            'transactions.amount_sum_12h',
            'transactions.amount_count_12h',
            'transactions.amount_avg_12h',
            'transactions.amount_max_24h',
            'transactions.amount_sum_24h',
            'transactions.amount_count_24h',
            'transactions.amount_avg_24h',
            'transactions.es_transportation_count_14d',
            'transactions.es_health_count_14d',
            'transactions.es_otherservices_count_14d',
            'transactions.es_food_count_14d',
            'transactions.es_hotelservices_count_14d',
            'transactions.es_barsandrestaurants_count_14d',
            'transactions.es_tech_count_14d',
            'transactions.es_sportsandtoys_count_14d',
```

(continues on next page)

(continued from previous page)

```
            'transactions.es_wellnessandbeauty_count_14d',
            'transactions.es_hyper_count_14d',
            'transactions.es_fashion_count_14d',
            'transactions.es_home_count_14d',
            'transactions.es_travel_count_14d',
            'transactions.es_leisure_count_14d',
            'transactions.gender_F',
            'transactions.gender_M',
            'transactions.step',
            'transactions.amount',
            'transactions.timestamp_hour',
            'transactions.timestamp_day_of_week',
            'events.*']
```

```
# Import MLRun's Feature Store
import mlrun.feature_store as fstore

# Define the feature vector name for future reference
fv_name = 'transactions-fraud'

# Define the feature vector using our Feature Store (fstore)
transactions_fv = fstore.FeatureVector(fv_name,
                          features,
                          label_feature="labels.label",
                          description='Predicting a fraudulent transaction')

# Save the feature vector in the Feature Store
transactions_fv.save()
```

Step 2 - Preview the feature vector data

Obtain the values of the features in the feature vector, to ensure the data appears as expected.

```
# Import the Parquet Target so you can directly save the dataset as a file
from mlrun.datastore.targets import ParquetTarget

# Get offline feature vector as dataframe and save the dataset to parquet
train_dataset = fstore.get_offline_features(fv_name, target=ParquetTarget())
```

```
> 2021-09-19 17:59:28,415 [info] wrote target: {'name': 'parquet', 'kind': 'parquet',
↪'path': 'v3io:///projects/fraud-demo-admin/FeatureStore/transactions-fraud/parquet/
↪vectors/transactions-fraud-latest.parquet', 'status': 'ready', 'updated': '2021-09-
↪19T17:59:28.415727+00:00', 'size': 1915182}
```

```
# Preview the dataset
train_dataset.to_dataframe().tail(5)
```

```
       amount_max_2h  amount_sum_2h  amount_count_2h  amount_avg_2h  \
49995           2.95           2.95              1.0          2.950
49996          37.40          37.40              1.0         37.400
```

(continues on next page)

(continued from previous page)

```
49997           7.75           7.75              1.0          7.750
49998          28.89          28.89              1.0         28.890
49999          78.18         105.43              2.0         52.715

       amount_max_12h  amount_sum_12h  amount_count_12h  amount_avg_12h  \
49995            2.95            2.95               1.0           2.950
49996           37.40           37.40               1.0          37.400
49997            7.75           12.99               2.0           6.495
49998           38.35          107.76               4.0          26.940
49999           78.18          153.78               3.0          51.260

       amount_max_24h  amount_sum_24h  ...  gender_F  gender_M  step  amount  \
49995            2.95            2.95  ...         1         0    41    2.95
49996           37.40           37.40  ...         0         1    40   37.40
49997           61.23          112.76  ...         1         0    91    7.75
49998           52.97          249.41  ...         0         1    56   28.89
49999           78.18          220.19  ...         1         0    76   78.18

       timestamp_hour  timestamp_day_of_week  event_details_change  \
49995              17                      6                   0.0
49996              17                      6                   0.0
49997              17                      6                   1.0
49998              17                      6                   1.0
49999              17                      6                   1.0

       event_login  event_password_change  label
49995          0.0                    1.0      0
49996          0.0                    1.0      0
49997          0.0                    0.0      0
49998          0.0                    0.0      0
49999          0.0                    0.0      0

[5 rows x 36 columns]
```

Step 3 - Train models and choose highest accuracy

With MLRun, one can easily train different models and compare the results. The code below trains three different models, and chooses the model with the highest accuracy. Each uses a different algorithm (random forest, XGBoost, adabost).

```
# Import the Sklearn classifier function from the function hub
classifier_fn = mlrun.import_function('hub://sklearn-classifier')
```

```
# Prepare the parameters list for the training function
# Use 3 different models
training_params = {"model_name": ['transaction_fraud_rf',
                                  'transaction_fraud_xgboost',
                                  'transaction_fraud_adaboost'],

                   "model_pkg_class": ['sklearn.ensemble.RandomForestClassifier',
                                       'sklearn.ensemble.GradientBoostingClassifier',
```

(continues on next page)

(continued from previous page)

```
                                    'sklearn.ensemble.AdaBoostClassifier']}

# Define the training task, including the feature vector, label and hyperparams␣
↪definitions
train_task = mlrun.new_task('training',
                      inputs={'dataset': transactions_fv.uri},
                      params={'label_column': 'label'}
                     )

train_task.with_hyper_params(training_params, strategy='list', selector='max.accuracy')

# Specify the cluster image
classifier_fn.spec.image = 'mlrun/mlrun'

# Run training
classifier_fn.run(train_task, local=False)
```

```
> 2021-09-19 17:59:28,799 [info] starting run training␣
↪uid=9349c60dd9f24a33b536c59e89978e7b DB=http://mlrun-api:8080
> 2021-09-19 17:59:29,042 [info] Job is running in the background, pod: training-2jntc
> 2021-09-19 17:59:47,926 [info] best iteration=1, used criteria max.accuracy
> 2021-09-19 17:59:48,990 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-09-19 17:59:51,574 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f464baf0c50>
```

Step 4 - Perform feature selection

As part of our data science process, try to reduce the training dataset's size to get rid of bad or unuseful features and save computation time.

Use the ready-made feature selection function from the hub hub://feature_selection to select the best features to keep on a sample from the dataset, and run the function on that.

```
feature_selection_fn = mlrun.import_function('hub://feature_selection')

feature_selection_run = feature_selection_fn.run(
            params={'sample_ratio':0.25,
                    'output_vector_name':fv_name + "-short",
                   'ignore_type_errors': True},

            inputs={'df_artifact': transactions_fv.uri},
```

(continues on next page)

(continued from previous page)

```
            name='feature_extraction',
            handler='feature_selection',
    local=False)
```

```
> 2021-09-19 17:59:51,768 [info] starting run feature_extraction␣
↪uid=3a50bd0e4175459fb53873d8f78a440a DB=http://mlrun-api:8080
> 2021-09-19 17:59:52,004 [info] Job is running in the background, pod: feature-
↪extraction-lf46d
> 2021-09-19 17:59:59,099 [info] Couldn't calculate chi2 because of: Input X must be non-
↪negative.
> 2021-09-19 18:00:04,008 [info] votes needed to be selected: 3
> 2021-09-19 18:00:05,329 [info] wrote target: {'name': 'parquet', 'kind': 'parquet',
↪'path': 'v3io:///projects/fraud-demo-admin/FeatureStore/transactions-fraud-short/
↪parquet/vectors/transactions-fraud-short-latest.parquet', 'status': 'ready', 'updated
↪': '2021-09-19T18:00:05.329695+00:00', 'size': 668722}
> 2021-09-19 18:00:05,677 [info] run executed, status=completed
Pass k=5 as keyword args. From version 0.25 passing these as positional arguments will␣
↪result in an error
Liblinear failed to converge, increase the number of iterations.
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-09-19 18:00:07,537 [info] run executed, status=completed
```

```
mlrun.get_dataitem(feature_selection_run.outputs['top_features_vector']).as_df().tail(5)
```

```
       amount_max_2h  amount_sum_2h  amount_count_2h  amount_avg_2h  \
49996          37.40          37.40              1.0      37.400000
49997           7.75           7.75              1.0       7.750000
49998          28.89          28.89              1.0      28.890000
49999          78.18         105.43              2.0      52.715000
50000          19.37          24.61              3.0       8.203333

       amount_max_12h  label
49996           37.40      0
49997            7.75      0
49998           38.35      0
49999           78.18      0
50000           19.37      0
```

Step 5 - Train the models with top features

Following the feature selection, you train new models using the resultant features. You can observe the accuracy and other results remain high, meaning you get a model that requires less features to be accurate and thus less error-prone.

```
# Defining our training task, including our feature vector, label and hyperparams␣
↪definitions
ensemble_train_task = mlrun.new_task('training',
                      inputs={'dataset': feature_selection_run.outputs['top_features_
↪vector']},
                      params={'label_column': 'label'}
                     )
ensemble_train_task.with_hyper_params(training_params, strategy='list', selector='max.
↪accuracy')

classifier_fn.run(ensemble_train_task)
```

```
> 2021-09-19 18:00:07,661 [info] starting run training␣
↪uid=a6d9ae72cfd3462cace205f8b363d214 DB=http://mlrun-api:8080
> 2021-09-19 18:00:08,077 [info] Job is running in the background, pod: training-v2bt4
> 2021-09-19 18:00:20,781 [info] best iteration=3, used criteria max.accuracy
> 2021-09-19 18:00:21,696 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-09-19 18:00:27,561 [info] run executed, status=completed
```

```
<mlrun.model.RunObject at 0x7f464baed490>
```

Done!

You've completed Part 2 of the model training with the feature store. Proceed to Part 3 to learn how to deploy and monitor the model.

8.5.3 Part 3: Serving

In this part you use MLRun's **serving runtime** to deploy the trained models from the previous stage a `Voting Ensemble` using **max vote** logic.
You will also use MLRun's **Feature store** to receive the latest tag of the online **Feature Vector** we defined in the previous stage.

By the end of this tutorial you'll learn how to:

- Define a model class to load the models, run preprocessing, and predict on the data
- Define Voting Ensemble function on top of our models

- Test the serving function locally using the `mock server`
- Deploy the function to the cluster and test it live

Environment setup

First, make sure SciKit-Learn is installed in the correct version:

```
!pip install -U scikit-learn==1.0.2
```

Restart your kernel post installing. Secondly, since the work is done in this project scope, define the project itself for all your MLRun work in this notebook.

```
project_name = 'fraud-demo'
```

```
import mlrun

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name, context="./", user_project=True)
```

```
> 2021-10-28 11:59:01,033 [info] loaded project fraud-demo from MLRun DB
```

Define model class

- Load models
- Predict from the FS Online service via the `source` key

```
# mlrun: start-code
```

```
import numpy as np
from cloudpickle import load
from mlrun.serving.v2_serving import V2ModelServer

class ClassifierModel(V2ModelServer):

    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

    def predict(self, body: dict) -> list:
        """Generate model predictions from sample"""
        print(f"Input -> {body['inputs']}")
        feats = np.asarray(body['inputs'])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()
```

```
# mlrun: end-code
```

Define a serving function

MLRun serving can produce managed real-time serverless pipelines from various tasks, including MLRun models or standard model files. The pipelines use the Nuclio real-time serverless engine, which can be deployed anywhere. Nuclio is a high-performance open-source serverless framework that's focused on data, I/O, and compute-intensive workloads.

The **EnrichmentVotingEnsemble** and the **EnrichmentModelRouter** router classes auto enrich the request with data from the feature store. The router input accepts lists of inference request (each request can be a dict or list of incoming features/keys). It enriches the request with data from the specified feature vector (`feature_vector_uri`).

In many cases the features can have null values (None, NaN, Inf, ...). The `Enrichment` routers can substitute the null value with fixed or statistical value per feature. This is done through the `impute_policy` parameter, which accepts the impute policy per feature (where * is used to specify the default). The value can be a fixed number for constants or `$mean`, `$max`, `$min`, `$std`, `$count` for statistical values. to substitute the value with the equivalent feature stats (taken from the feature store).

The code below performs the following steps:

- Gather ClassifierModel code from this notebook
- Define `EnrichmentVotingEnsemble` - Max-Vote based ensemble with feature enrichment and imputing
- Add the previously trained models to the ensemble

```
# Create the serving function from the code above
serving_fn = mlrun.code_to_function('transaction-fraud', kind='serving', image="mlrun/
↪mlrun")

serving_fn.set_topology('router', 'mlrun.serving.routers.EnrichmentVotingEnsemble', name=
↪'VotingEnsemble',
                        feature_vector_uri="transactions-fraud-short", impute_policy={"*
↪": "$mean"})

model_names = [
'RandomForestClassifier',
'GradientBoostingClassifier',
'AdaBoostClassifier'
]

for i, name in enumerate(model_names, start=1):
    serving_fn.add_model(name, class_name="ClassifierModel", model_path=project.get_
↪artifact_uri(f"training_model#{i}:latest"))

# Plot the ensemble configuration
serving_fn.spec.graph.plot()
```

```
<graphviz.dot.Digraph at 0x7f5af5d471d0>
```

Test the server locally

Before deploying the serving function, test it in the current notebook and check the model output.

```
# Create a mock server from the serving function
local_server = serving_fn.to_mock_server()
```

```
> 2021-10-28 11:59:11,260 [info] model RandomForestClassifier was loaded
> 2021-10-28 11:59:11,306 [info] model GradientBoostingClassifier was loaded
> 2021-10-28 11:59:11,350 [info] model AdaBoostClassifier was loaded
```

```
# Choose an id for the test
sample_id = 'C76780537'

model_inference_path = '/v2/models/infer'

# Send our sample ID for prediction
local_server.test(path=model_inference_path,
            body={'inputs': [[sample_id]]})

# Notice the input vector is printed 3 times (once per child model) and is enriched with␣
↪data from the feature store
```

```
Input -> [[14.68, 14.68, 1.0, 14.68, 70.81]]
Input -> [[14.68, 14.68, 1.0, 14.68, 70.81]]
Input -> [[14.68, 14.68, 1.0, 14.68, 70.81]]
```

```
{'id': '757c736c985a4c42b3ebd58f3c50f1b2',
 'model_name': 'VotingEnsemble',
 'outputs': [0],
 'model_version': 'v1'}
```

Accessing the real-time feature vector directly

You can also directly query the feature store values using the `get_online_feature_service` method. This method is used internally in the EnrichmentVotingEnsemble router class

```
import mlrun.feature_store as fstore

# Create the online feature service
svc = fstore.get_online_feature_service('transactions-fraud-short:latest', impute_policy=
↪{"*": "$mean"})

# Get sample feature vector
sample_fv = svc.get([{'source': sample_id}])
sample_fv
```

```
[{'amount_max_2h': 14.68,
  'amount_max_12h': 70.81,
  'amount_sum_2h': 14.68,
```

(continues on next page)

(continued from previous page)

```
  'amount_count_2h': 1.0,
  'amount_avg_2h': 14.68}]
```

Deploying the function on the kubernetes cluster

You can now deploy the function. Once it's deployed you get a function with an http trigger that can be called from other locations.

```
import os

# Enable model monitoring
serving_fn.set_tracking()
project.set_model_monitoring_credentials(os.getenv('V3IO_ACCESS_KEY'))

# Deploy the serving function
serving_fn.deploy()
```

```
> 2021-10-28 11:59:17,554 [info] Starting remote function deploy
2021-10-28 11:59:17  (info) Deploying function
2021-10-28 11:59:17  (info) Building
2021-10-28 11:59:17  (info) Staging files and preparing base images
2021-10-28 11:59:17  (info) Building processor image
2021-10-28 11:59:19  (info) Build complete
2021-10-28 11:59:25  (info) Function deploy complete
> 2021-10-28 11:59:25,657 [info] successfully deployed function: {'internal_invocation_
↪urls': ['nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.
↪local:8080'], 'external_invocation_urls': ['default-tenant.app.yh38.iguazio-cd2.
↪com:32287']}
```

```
'http://default-tenant.app.yh38.iguazio-cd2.com:32287'
```

Test the server

Test the serving function and examine the model output.

```
# Choose an id for the test
sample_id = 'C76780537'

model_inference_path = '/v2/models/infer'

# Send the sample ID for prediction
serving_fn.invoke(path=model_inference_path,
                  body={'inputs': [[sample_id]]})
```

```
> 2021-10-28 11:59:25,722 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.local:8080/v2/
↪models/infer'}
```

```
{'id': '4b9c4914-964f-4bd5-903d-c4885ed7c090',
 'model_name': 'VotingEnsemble',
 'outputs': [0],
 'model_version': 'v1'}
```

You can also directly query the feature store values, which are used in the enrichment.

Simulate incoming data

```
# Load the dataset
data = mlrun.get_dataitem('https://s3.wasabisys.com/iguazio/data/fraud-demo-mlrun-fs-
↪docs/data.csv').as_df()

# Sample 50k lines
data = data.sample(50000)

# keys
sample_ids = data['source'].to_list()
```

```
from random import choice, uniform
from time import sleep

# Sending random requests
for _ in range(4000):
    data_point = choice(sample_ids)
    try:
        resp = serving_fn.invoke(path=model_inference_path, body={'inputs': [[data_
↪point]]})
        print(resp)
        sleep(uniform(0.2, 1.7))
    except OSError:
        pass
```

```
> 2021-10-28 12:00:23,079 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.local:8080/v2/
↪models/infer'}
{'id': '6b813638-e9ef-4e92-85c8-cfbd0b74fe32', 'model_name': 'VotingEnsemble', 'outputs
↪': [0], 'model_version': 'v1'}
> 2021-10-28 12:00:23,857 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.local:8080/v2/
↪models/infer'}
{'id': 'f84bf2ec-a718-4e90-a7d5-fe08e254f3c8', 'model_name': 'VotingEnsemble', 'outputs
↪': [0], 'model_version': 'v1'}
> 2021-10-28 12:00:24,545 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.local:8080/v2/
↪models/infer'}
{'id': '7bb023f7-edbc-47a6-937b-4a15c8380b74', 'model_name': 'VotingEnsemble', 'outputs
↪': [0], 'model_version': 'v1'}
> 2021-10-28 12:00:24,921 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-fraud-demo-admin-transaction-fraud.default-tenant.svc.cluster.local:8080/v2/
↪models/infer'}
```

(continues on next page)

(continued from previous page)

```
{'id': '57882cca-537a-43e1-9986-1bbc72fb84b7', 'model_name': 'VotingEnsemble', 'outputs
↪': [0], 'model_version': 'v1'}
```

8.5.4 Part 4: Automated ML pipeline

MLRun Project is a container for all your work on a particular activity: sll the associated code, functions, jobs/workflows and artifacts. Projects can be mapped to `git` repositories which enables versioning, collaboration, and CI/CD. Users can create project definitions using the SDK or a yaml file and store those in MLRun DB, file, or archive. Once the project is loaded you can run jobs/workflows which refer to any project element by name, allowing separation between configuration and code.

Projects contain `workflows` that execute the registered functions in a sequence/graph (DAG). It can reference project parameters, secrets and artifacts by name. The following notebook demonstrate how to build an automated workflow with **feature selection**, **training**, **testing**, and **deployment**.

Step 1: Setting up your project

To run a pipeline, you first need to get or create a project object and define/import the required functions for its execution. See load projects for details.

The following code gets or creates a user project named "fraud-demo-<username>".

```
# Set the base project name
project_name = 'fraud-demo'
```

```
import mlrun

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name, context="./", user_project=True)
```

```
> 2021-10-28 13:54:45,892 [info] loaded project fraud-demo from MLRun DB
```

Step 2: Updating project and function definitions

You need to save the definitions for the function you use in the projects so it is possible to automatically convert code to functions or import external functions whenever you load new versions of the code or when you run automated CI/CD workflows. In addition you may want to set other project attributes such as global parameters, secrets, and data.

The code can be stored in Python files, notebooks, external repositories, packaged containers, etc. Use the `project.set_function()` method to register the code in the project. The definitions are saved to the project object, as well as in a YAML file in the root of our project. Functions can also be imported from MLRun Function Hub (using the `hub://` schema).

You used the following functions in this tutorial:

- `feature_selection` — the first function, which determines the top features to be used for training
- `train` — the model-training function
- `test-classifier` — the model-testing function

- `mlrun-model` — the model-serving function

Note

`set_function` uses the `code_to_function` and `import_function` methods under the hood (used in the previous notebooks), but in addition it saves the function configurations in the project spec for use in automated workflows and CI/CD.

Add the function definitions to the project along with parameters and data artifacts and save the project.

```
project.set_function('hub://feature_selection', 'feature_selection')
project.set_function('hub://sklearn-classifier','train')
project.set_function('hub://test_classifier', 'test')
project.set_function('hub://v2_model_server', 'serving')
```

```
<mlrun.runtimes.serving.ServingRuntime at 0x7f6229497190>
```

```
# set project level parameters and save
project.spec.params = {'label_column': 'label'}
project.save()
```

When you save the project it stores the project definitions in the `project.yaml`. This means that you can load the project from the source control (GIT) and run it with a single command or API call.

The project YAML for this project can be printed using:

```
print(project.to_yaml())
```

```
kind: project
metadata:
  name: fraud-demo-admin
  created: '2021-08-05T15:59:59.434655'
spec:
  params:
    label_column: label
  functions:
  - url: hub://feature_selection
    name: feature_selection
  - url: hub://sklearn-classifier
    name: train
  - url: hub://test_classifier
    name: test
  - url: hub://v2_model_server
    name: serving
  workflows:
  - name: main
    path: workflow.py
    engine: null
  artifacts: []
  desired_state: online
  disable_auto_mount: false
```

(continues on next page)

(continued from previous page)

```
status:
  state: online
```

Saving and loading projects from GIT

After you save the project and its elements (functions, workflows, artifacts, etc.) you can commit all the changes to a GIT repository. Do this using standard GIT tools or using MLRun `project` methods such as `pull`, `push`, `remote` that call the Git API for you.

Projects can then be loaded from Git using MLRun `load_project` method, for example:

```
project = mlrun.load_project("./myproj", "git://github.com/mlrun/project-demo.git",␣
↪name=project_name)
```

or using MLRun CLI:

```
mlrun project -n myproj -u "git://github.com/mlrun/project-demo.git" ./myproj
```

Read load projects for more details.

Using Kubeflow pipelines

You're now ready to create a full ML pipeline. This is done by using Kubeflow Pipelines — an open-source framework for building and deploying portable, scalable, machine-learning workflows based on Docker containers. MLRun leverages this framework to take your existing code and deploy it as steps in the pipeline.

Step 3: Defining and saving a pipeline workflow

A pipeline is created by running an MLRun **"workflow"**. The following code defines a workflow and writes it to a file in your local directory; (the file name is **workflow.py**). The workflow describes a directed acyclic graph (DAG) for execution using Kubeflow Pipelines, and depicts the connections between the functions and the data as part of an end-to-end pipeline. The workflow file has a definition of a pipeline DSL for connecting the function inputs and outputs.

The defined pipeline includes the following steps:

- Perform feature selection (`feature_selection`).
- Train and the model (`train`).
- Test the model with its test data set.
- Deploy the model as a real-time serverless function (`deploy`).

  Note: A pipeline can also include continuous build integration and deployment (CI/CD) steps, such as building container images and deploying models.

```
%%writefile workflow.py
import mlrun
from kfp import dsl
from mlrun.model import HyperParamOptions
```

(continues on next page)

(continued from previous page)

```
from mlrun import (
    build_function,
    deploy_function,
    import_function,
    run_function,
)


@dsl.pipeline(
    name="Fraud Detection Pipeline",
    description="Detecting fraud from a transactions dataset"
)

def kfpipeline(vector_name='transactions-fraud'):

    project = mlrun.get_current_project()

    # Feature selection
    feature_selection = run_function(
        "feature_selection",
        name="feature_selection",
        params={'sample_ratio':0.25,'output_vector_name': "short",
                'ignore_type_errors': True},
        inputs={'df_artifact': project.get_artifact_uri(vector_name, 'feature-vector')},
        outputs=['top_features_vector'])


    # train with hyper-paremeters
    train = run_function(
        "train",
        name="train",
        params={"sample": -1,
                "label_column": project.get_param('label_column', 'label'),
                "test_size": 0.10},
        hyperparams={"model_name": ['transaction_fraud_rf',
                                    'transaction_fraud_xgboost',
                                    'transaction_fraud_adaboost'],

                     'model_pkg_class': ["sklearn.ensemble.RandomForestClassifier",
                                         "sklearn.linear_model.LogisticRegression",
                                         "sklearn.ensemble.AdaBoostClassifier"]},
        hyper_param_options=HyperParamOptions(selector="max.accuracy"),
        inputs={"dataset": feature_selection.outputs['top_features_vector']},
        outputs=['model', 'test_set'])


    # test and visualize our model
    test = run_function(
        "test",
        name="test",
        params={"label_column": project.get_param('label_column', 'label')},
        inputs={
```

(continues on next page)

(continued from previous page)

```
            "models_path": train.outputs["model"],
            "test_set": train.outputs["test_set"]})

    # route the serving model to use enrichment
    funcs['serving'].set_topology('router',
                                  'mlrun.serving.routers.EnrichmentModelRouter',
                                  name='EnrichmentModelRouter',
                                  feature_vector_uri="transactions-fraud-short",
                                  impute_policy={"*": "$mean"},
                                  exist_ok=True)


    # deploy the model as a serverless function, you can pass a list of models to serve
    deploy = deploy_function("serving", models=[{"key": 'fraud', "model_path": train.
↪outputs["model"]}])
```

```
Overwriting workflow.py
```

Step 4: Registering the workflow

Use the `set_workflow` MLRun project method to register your workflow with MLRun. The following code sets the `name` parameter to the selected workflow name ("main") and the `code` parameter to the name of the workflow file that is found in your project directory (**workflow.py**).

```
# Register the workflow file as "main"
project.set_workflow('main', 'workflow.py')
```

Step 5: Running a pipeline

First run the following code to save your project:

```
project.save()
```

Use the `run` MLRun project method to execute your workflow pipeline with Kubeflow Pipelines.

You can pass **`arguments`** or set the **`artifact_path`** to specify a unique path for storing the workflow artifacts.

```
run_id = project.run(
    'main',
    arguments={},
    dirty=True, watch=True)
```

```
<graphviz.dot.Digraph at 0x7f6234c28b50>
```

```
<IPython.core.display.HTML object>
```

Step 6: Test the model end point

Now that your model is deployed using the pipeline, you can invoke it as usual:

```
# Define the serving function
serving_fn = project.func('serving')

# Choose an id for the test
sample_id = 'C76780537'
model_inference_path = '/v2/models/fraud/infer'

# Send the sample ID for predcition
serving_fn.invoke(path=model_inference_path,
                  body={'inputs': [[sample_id]]})
```

```
> 2021-10-28 13:56:56,170 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-fraud-demo-admin-v2-model-server.default-tenant.svc.cluster.local:8080/v2/
↪models/fraud/infer'}
```

```
{'id': '90f4b67c-c9e0-4e35-917f-979b71c5ad75',
 'model_name': 'fraud',
 'outputs': [0.0]}
```

Done!

CHAPTER

NINE

BATCH RUNS AND WORKFLOWS

In this section

9.1 MLRun execution context

After running a job, you need to be able to track it. To gain the maximum value, MLRun uses the job `context` object inside the code. This provides access to job metadata, parameters, inputs, secrets, and API for logging and monitoring the results, as well as log text, files, artifacts, and labels.

- If `context` is specified as the first parameter in the function signature, MLRun injects the current job context into it.
- Alternatively, if it does not run inside a function handler (e.g. in Python main or Notebook) you can obtain the `context` object from the environment using the *get_or_create_ctx()* function.

Common context methods:

- `get_secret(key: str)` — get the value of a secret
- `logger.info("started experiment..")` — textual logs
- `log_result(key: str, value)` — log simple values
- `set_label(key, value)` — set a label tag for that task
- `log_artifact(key, body=None, local_path=None, ...)` — log an artifact (body or local file)
- `log_dataset(key, df, ...)` — log a dataframe object
- `log_model(key, ...)` — log a model object

Example function and usage of the context object:

```
from mlrun.artifacts import ChartArtifact
import pandas as pd

def my_job(context, p1=1, p2="x"):
    # load MLRUN runtime context (will be set by the runtime framework)

    # get parameters from the runtime context (or use defaults)

    # access input metadata, values, files, and secrets (passwords)
    print(f"Run: {context.name} (uid={context.uid})")
    print(f"Params: p1={p1}, p2={p2}")
    print("accesskey = {}".format(context.get_secret("ACCESS_KEY")))
```

(continues on next page)

(continued from previous page)

```
    print("file\n{}\n".format(context.get_input("infile.txt", "infile.txt").get()))

    # Run some useful code e.g. ML training, data prep, etc.

    # log scalar result values (job result metrics)
    context.log_result("accuracy", p1 * 2)
    context.log_result("loss", p1 * 3)
    context.set_label("framework", "sklearn")

    # log various types of artifacts (file, web page, table), will be versioned and␣
↪visible in the UI
    context.log_artifact(
        "model",
        body=b"abc is 123",
        local_path="model.txt",
        labels={"framework": "xgboost"},
    )
    context.log_artifact(
        "html_result", body=b"<b> Some HTML <b>", local_path="result.html"
    )

    # create a chart output (will show in the pipelines UI)
    chart = ChartArtifact("chart")
    chart.labels = {"type": "roc"}
    chart.header = ["Epoch", "Accuracy", "Loss"]
    for i in range(1, 8):
        chart.add_row([i, i / 20 + 0.75, 0.30 - i / 20])
    context.log_artifact(chart)

    raw_data = {
        "first_name": ["Jason", "Molly", "Tina", "Jake", "Amy"],
        "last_name": ["Miller", "Jacobson", "Ali", "Milner", "Cooze"],
        "age": [42, 52, 36, 24, 73],
        "testScore": [25, 94, 57, 62, 70],
    }
    df = pd.DataFrame(raw_data, columns=["first_name", "last_name", "age", "testScore"])
    context.log_dataset("mydf", df=df, stats=True)
```

Example of creating the context objects from the environment:

```
if __name__ == "__main__":
    context = mlrun.get_or_create_ctx('train')
    p1 = context.get_param('p1', 1)
    p2 = context.get_param('p2', 'a-string')
    # do something
    context.log_result("accuracy", p1 * 2)
    # commit the tracking results to the DB (and mark as completed)
    context.commit(completed=True)
```

Note that MLRun context is also a python context and can be used in a `with` statement (eliminating the need for `commit`).

```
if __name__ == "__main__":
    with mlrun.get_or_create_ctx('train') as context:
        p1 = context.get_param('p1', 1)
        p2 = context.get_param('p2', 'a-string')
        # do something
        context.log_result("accuracy", p1 * 2)
```

9.2 Decorators and auto-logging

While it is possible to log results and artifacts using *the MLRun execution context*, it is often more convenient to use the *mlrun.handler()* decorator.

9.2.1 Basic example

Assume you have the following code in `train.py`

```
import pandas as pd
from sklearn.svm import SVC

def train_and_predict(train_data,
                      predict_input,
                      label_column='label'):

    x = train_data.drop(label_column, axis=1)
    y = train_data[label_column]

    clf = SVC()
    clf.fit(x, y)

    return list(clf.predict(predict_input))
```

With the `mlrun.handler` the python function itself would not change, and logging of the inputs and outputs would be automatic. The resultant code is as follows:

```
import pandas as pd
from sklearn.svm import SVC
import mlrun

@mlrun.handler(labels={'framework':'scikit-learn'},
               outputs=['prediction:dataset'],
               inputs={"train_data": pd.DataFrame,
                       "predict_input": pd.DataFrame})
def train_and_predict(train_data,
                      predict_input,
                      label_column='label'):

    x = train_data.drop(label_column, axis=1)
    y = train_data[label_column]

    clf = SVC()
```

(continues on next page)

(continued from previous page)

```
    clf.fit(x, y)

    return list(clf.predict(predict_input))
```

To run the code, use the following example:

```
import mlrun
project = mlrun.get_or_create_project("mlrun-example", context="./", user_project=True)

trainer = project.set_function("train.py", name="train_and_predict", kind="job", image=
↪"mlrun/mlrun", handler="train_and_predict")

trainer_run = project.run_function(
    "train_and_predict",
    inputs={"train_data": mlrun.get_sample_path('data/iris/iris_dataset.csv'),
            "predict_input": mlrun.get_sample_path('data/iris/iris_to_predict.csv')
           }
)
```

The outcome is a run with:

1. A label with key "framework" and value "scikit-learn".
2. Two inputs "train_data" and "predict_input" created from Pandas DataFrame.
3. An artifact called "prediction" of type "dataset". The contents of the dataset will be the return value (in this case the prediction result).

9.2.2 Labels

The decorator gives you the option to set labels for the run. The `labels` parameter is a dictionary with keys and values to set for the labels.

9.2.3 Input type parsing

The `mlrun.handler` decorator can also parse the input types, if they are specified. An equivalent definition is as follows:

```
@mlrun.handler(labels={'framework':'scikit-learn'},
               outputs=['prediction:dataset'])
def train_and_predict(train_data: pd.DataFrame,
                      predict_input: pd.DataFrame,
                      label_column='label'):

...
```

> **Note:** If the inputs does not have a type input, the decorator assumes the parameter type in *mlrun.datastore.DataItem*. If you specify `inputs=False`, all the run inputs are assumed to be of type `mlrun.datastore.DataItem`. You also have the option to specify a dictionary where each key is the name of the input and the value is the type.

9.2.4 Logging return values as artifacts

If you specify the `outputs` parameter, the return values will be logged as the run artifacts. `outputs` expects a list; the length of the list must match the number of returned values.

The simplest option is to specify a list of strings. Each string contains the name of the artifact. You can also specify the artifact type by adding a colon after the artifact name followed by the type (`'name:artifact_type'`). The following are valid artifact types:

- dataset
- directory
- file
- object
- plot
- result

If you use only the name without the type, the following mapping is used:

Python type	Artifact type
pandas.DataFrame	Dataset
pandas.Series	Dataset
numpy.ndarray	Dataset
dict	Result
list	Result
tuple	Result
str	Result
int	Result
float	Result
bytes	Object
bytearray	Object
matplotlib.pyplot.Figure	Plot
plotly.graph_objs.Figure	Plot
bokeh.plotting.Figure	Plot

Another option is to specify a tuple in the form of `(name, artifact_type)` or `(name, artifact_type, arguments)`. Refer to the *`mlrun.handler()`* for more details.

9.3 Running a task (job)

In this section

- *Submit tasks (jobs) using run_function*
- *Run result object and UI*

9.3.1 Submit tasks (jobs) using run_function

Use the *run_function()* method for invoking a job over MLRun batch functions. The `run_function` method accepts various parameters such as `name`, `handler`, `params`, `inputs`, `schedule`, etc. Alternatively, you can pass a **`Task`** object (see: *new_task()*) that holds all of the parameters plus the advanced options.

Functions can host multiple methods (handlers). You can set the default handler per function. You need to specify which handler you intend to call in the run command.

You can pass `parameters` (arguments) or data `inputs` (such as datasets, feature-vectors, models, or files) to the functions through the `run` method.

- Inside the function you can access the parameters/inputs by simply adding them as parameters to the function, or you can get them from the context object (using `get_param()` and `get_input()`).
- Various data objects (files, tables, models, etc.) are passed to the function as data item objects. You can pass data objects using the inputs dictionary argument, where the dictionary keys match the function's handler argument names and the MLRun data urls are provided as the values. The data is passed into the function as a *DataItem* object that handles data movement, tracking, and security in an optimal way. Read more about data objects in Data stores.

You can use `run_function` as a `project` methods, or as global (`mlrun.`) methods. For example:

```
# run the "train" function in myproject
run_results = myproject.run_function("train", inputs={"data": data_url})

# run the "train" function in the current/active project (or in a pipeline)
run_results = mlrun.run_function("train", inputs={"data": data_url})
```

The first parameter in `run_function` is the function name (in the project), or it can be a function object if you want to use functions that you imported/created ad hoc, or modify a function spec, for example:

```
run_results = project.run_function(fn, params={"label_column": "label"}, inputs={'data':␣
↪data_url})
```

Run/simulate functions locally:

Functions can also run and be debugged locally by using the `local` runtime or by setting the `local=True` parameter in the *run()* method (for batch functions).

MLRun also supports iterative jobs that can run and track multiple child jobs (for hyperparameter tasks, AutoML, etc.). See *Hyperparameter tuning optimization* for details and examples.

9.3.2 Run result object and UI

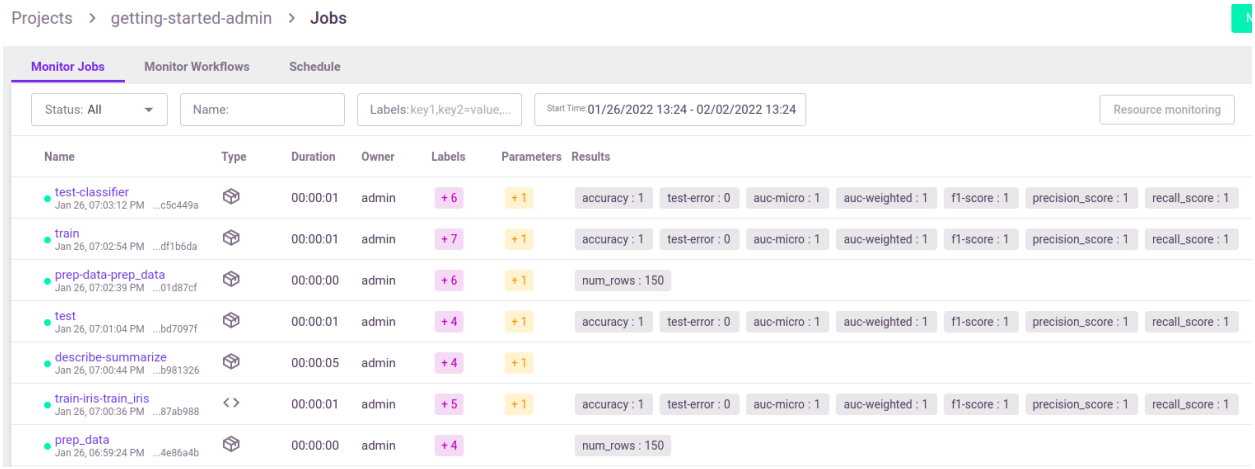
The *run_function()* command returns an MLRun *RunObject* object that you can use to track the job and its results. If you pass the parameter `watch=True` (default) the command blocks until the job completes.

Run object has the following methods/properties:

- `uid()` — returns the unique ID.
- `state()` — returns the last known state.
- `show()` — shows the latest job state and data in a visual widget (with hyperlinks and hints).

- `outputs` — returns a dictionary of the run results and artifact paths.
- `logs(watch=True)` — returns the latest logs. Use `Watch=False` to disable the interactive mode in running jobs.
- `artifact(key)` — returns an artifact for the provided key (as *DataItem* object).
- `output(key)` — returns a specific result or an artifact path for the provided key.
- `wait_for_completion()` — wait for async run to complete
- `refresh()` — refresh run state from the db/service
- `to_dict()`, `to_yaml()`, `to_json()` — converts the run object to a dictionary, YAML, or JSON format (respectively).

You can view the job details, logs, and artifacts in the UI. When you first open the **Monitor Jobs** tab it displays the last jobs that ran and their data. Click a job name to view its run history, and click a run to view more of the run's data.



See full details and examples in Functions.

9.4 Running a multi-stage workflow

A workflow is a definition of execution of functions. It defines the order of execution of multiple dependent steps in a directed acyclic graph (DAG). A workflow can reference the project's params, secrets, artifacts, etc. It can also use a function execution output as a function execution input (which, of course, defines the order of execution).

MLRun supports running workflows on a `local` or `kubeflow` pipeline engine. The `local` engine runs the workflow as a local process, which is simpler for debugging and running simple/sequential tasks. The `kubeflow` ("kfp") engine runs as a task over the cluster and supports more advanced operations (conditions, branches, etc.). You can select the engine at runtime. Kubeflow-specific directives like conditions and branches are not supported by the `local` engine.

Workflows are saved/registered in the project using the *set_workflow()*.
Workflows are executed using the *run()* method or using the CLI command `mlrun project`.

Refer to the ***Tutorials and Examples*** for complete examples.

In this section

- *Composing workflows*
- *Saving workflows*
- *Running workflows*

9.4.1 Composing workflows

Workflows are written as python functions that make use of function operations (run, build, deploy) and can access project parameters, secrets, and artifacts using *get_param()*, *get_secret()* and *get_artifact_uri()*.

For workflows to work in Kubeflow you need to add a decorator (`@dsl.pipeline(..)`) as shown below.

Example workflow:

```
from kfp import dsl
import mlrun
from mlrun.model import HyperParamOptions

funcs = {}
DATASET = "iris_dataset"

in_kfp = True


@dsl.pipeline(name="Demo training pipeline", description="Shows how to use mlrun.")
def newpipe():

    project = mlrun.get_current_project()

    # build our ingestion function (container image)
    builder = mlrun.build_function("gen-iris")

    # run the ingestion function with the new image and params
    ingest = mlrun.run_function(
        "gen-iris",
        name="get-data",
        params={"format": "pq"},
        outputs=[DATASET],
    ).after(builder)

    # train with hyper-parameters
    train = mlrun.run_function(
        "train",
        name="train",
        params={"sample": -1, "label_column": project.get_param("label", "label"), "test_
↪size": 0.10},
        hyperparams={
            "model_pkg_class": [
                "sklearn.ensemble.RandomForestClassifier",
                "sklearn.linear_model.LogisticRegression",
                "sklearn.ensemble.AdaBoostClassifier",
            ]
        },
        hyper_param_options=HyperParamOptions(selector="max.accuracy"),
        inputs={"dataset": ingest.outputs[DATASET]},
        outputs=["model", "test_set"],
    )
    print(train.outputs)
```

(continues on next page)

(continued from previous page)

```
    # test and visualize our model
    mlrun.run_function(
        "test",
        name="test",
        params={"label_column": project.get_param("label", "label")},
        inputs={
            "models_path": train.outputs["model"],
            "test_set": train.outputs["test_set"],
        },
    )

    # deploy our model as a serverless function, we can pass a list of models to serve
    serving = mlrun.import_function("hub://v2_model_server", new_name="serving")
    deploy = mlrun.deploy_function(
        serving,
        models=[{"key": f"{DATASET}:v1", "model_path": train.outputs["model"]}],
    )

    # test out new model server (via REST API calls), use imported function
    tester = mlrun.import_function("hub://v2_model_tester", new_name="live_tester")
    mlrun.run_function(
        tester,
        name="model-tester",
        params={"addr": deploy.outputs["endpoint"], "model": f"{DATASET}:v1"},
        inputs={"table": train.outputs["test_set"]},
    )
```

9.4.2 Saving workflows

If you want to use workflows as part of an automated flow, save them and register them in the project. Use the *set_workflow()* method to register workflows, to specify a workflow name, the path to the workflow file, and the function `handler` name (or it looks for a handler named "pipeline"), and can set the default `engine` (local or kfp).

When setting the `embed` flag to True, the workflow code is embedded in the project file (can be used if you want to describe the entire project using a single YAML file).

You can define the schema for workflow arguments (data type, default, doc, etc.) by setting the `args_schema` with a list of **`EntrypointParam`** objects.

Example:

```
    # define agrument for the workflow
    arg = mlrun.model.EntrypointParam(
        "model_pkg_class",
        type="str",
        default="sklearn.linear_model.LogisticRegression",
        doc="model package/algorithm",
    )

    # register the workflow in the project and save the project
    project.set_workflow("main", "./myflow.py", handler="newpipe", args_schema=[arg])
    project.save()
```

(continues on next page)

(continued from previous page)

```
    # run the workflow
    project.run("main", arguments={"model_pkg_class": "sklearn.ensemble.
↪RandomForestClassifier"})
```

9.4.3 Running workflows

Use the *run()* method to execute workflows. Specify the workflow using its `name` or `workflow_path` (path to the workflow file) or `workflow_handler` (the workflow function handler). You can specify the input `arguments` for the workflow and can override the system default `artifact_path`.

Workflows are asynchronous by default. You can set the `watch` flag to True and the run operation blocks until completion and prints out the workflow progress. Alternatively, you can use `.wait_for_completion()` on the run object.

The default workflow engine is `kfp`. You can override it by specifying the `engine` in the `run()` or `set_workflow()` methods. Using the `local` engine executes the workflow state machine locally (its functions still run as cluster jobs). If you set the `local` flag to True, the workflow uses the `local` engine AND the functions run as local process. This mode is used for local debugging of workflows.

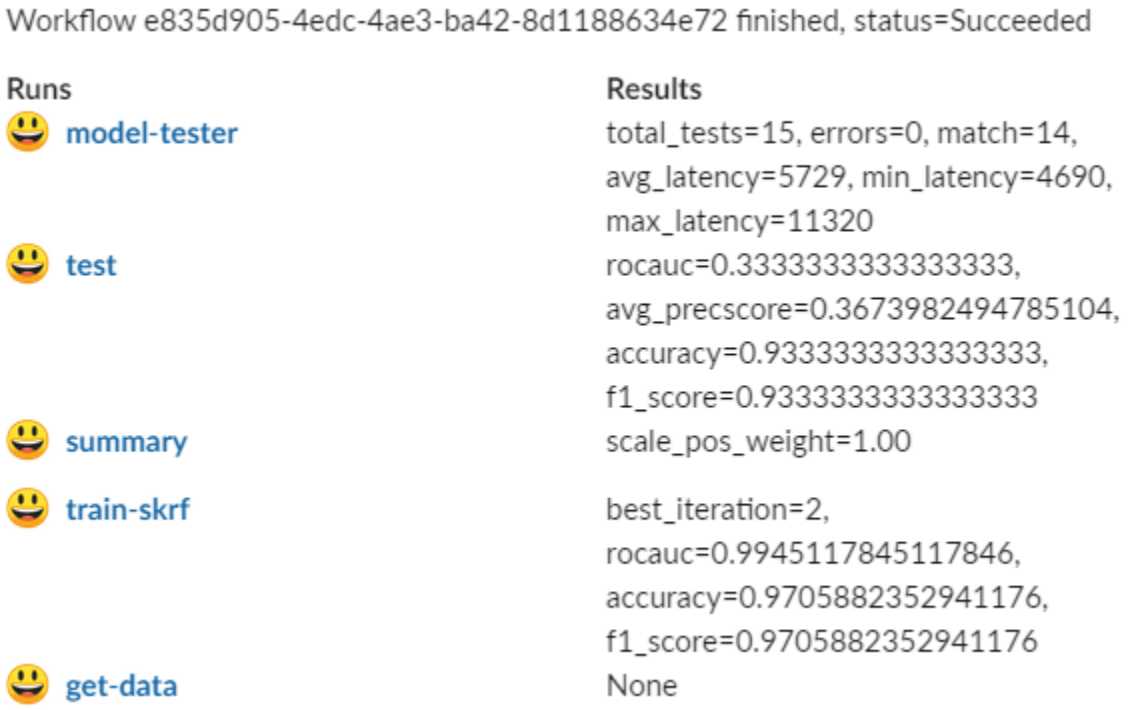
When running workflows from a git enabled context it first verifies that there are no uncommitted git changes (to guarantee that workflows that load from git do not use old code versions). You can suppress that check by setting the `dirty` flag to True.

Examples:

```
# simple run of workflow 'main' with arguments, block until it completes (watch=True)
run = project.run("main", arguments={"param1": 6}, watch=True)

# run workflow specified with a function handler (my_pipe)
run = project.run(workflow_handler=my_pipe)
# wait for pipeline completion
run.wait_for_completion()

# run workflow in local debug mode
run = project.run(workflow_handler=my_pipe, local=True, arguments={"param1": 6})
```

Notification

Instead of waiting for completion, you can set up a notification in Slack with a results summary, similar to:



Use one of:

```
# If you want to get slack notification after the run with the results summary, use
# project.notifiers.slack(webhook="https://<webhook>")
```

or in a Jupyter notebook with the `%env` magic command:

```
%env SLACK_WEBHOOK=<slack webhook url>
```

9.5 Scheduled jobs and workflows

Oftentimes you may want to run a `job` on a regular schedule. For example, fetching from a datasource every morning, compiling an analytics report every month, or detecting model drift every hour.

9.5.1 Creating a job and scheduling it

MLRun makes it very simple to add a schedule to a given `job`. To showcase this, the following job runs the code below, which resides in a file titled `schedule.py`:

```
def hello(context):
    print("You just ran a scheduled job!")
```

To create the job, use the `code_to_function` syntax and specify the `kind` like below:

```
import mlrun

job = mlrun.code_to_function(
    name="my-scheduled-job",      # Name of the job (displayed in console and UI)
    filename="schedule.py",       # Python file or Jupyter notebook to run
    kind="job",                   # Run as a job
    image="mlrun/mlrun",          # Use this Docker image
    handler="hello"               # Execute the function hello() within code.py
)
```

Running the job using a schedule

To add a schedule, run the job and specify the `schedule` parameter using Cron syntax like so:

```
job.run(schedule="0 * * * *")
```

This runs the job every hour. An excellent resource for generating Cron schedules is Crontab.guru.

9.5.2 Scheduling a workflow

After loading the project (`load_project`), run the project with the scheduled workflow:

```
project.run("main", schedule='0 * * * *')
```

CHAPTER

TEN

REAL-TIME SERVING PIPELINES (GRAPHS)

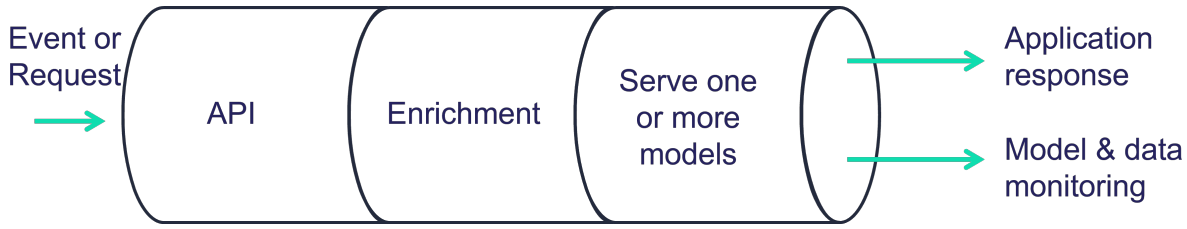
MLRun graphs enable building and running DAGs (directed acyclic graph).

MLRun graph capabilities include:

- Easy to build and deploy distributed real-time computation graphs
- Use the real-time serverless engine (Nuclio) for auto-scaling and optimized resource utilization
- Built-in operators to handle data manipulation, IO, machine learning, deep-learning, NLP, etc.
- Built-in monitoring for performance, resources, errors, data, model behaviour, and custom metrics
- Debug in the IDE/Notebook

Graphs are composed of individual steps. The first graph element accepts an `Event` object, transforms/processes the event and passes the result to the next steps in the graph. The final result can be written out to some destination (file, DB, stream, etc.) or returned back to the caller (one of the graph steps can be marked with `.respond()`).

The serving graphs can be composed of pre-defined graph steps, block-type elements (model servers, routers, ensembles, data readers and writers, data engineering tasks, validators, etc.), custom steps, or from native python classes/functions. A graph can have data processing steps, model ensembles, model servers, post-processing, etc. (see the Advanced Model Serving Graph Notebook Example). Graphs can auto-scale and span multiple function containers (connected through streaming protocols).



Different steps can run on the same local function, or run on a remote function. You can call existing functions from the graph and reuse them from other graphs, as well as scale up and down the different components individually.

Graphs can run inside your IDE or Notebook for test and simulation. Serving graphs are built on top of Nuclio (real-time serverless engine), MLRun jobs, MLRun Storey (native Python async and stream processing engine), and other MLRun facilities.

The serving graphs are used by MLRun's Feature Store to build real-time feature engineering pipelines.

In this section

10.1 Getting started

This example uses a custom class and custom function. See custom steps for more details.

In this section

10.1.1 Steps

The following code defines basic steps that illustrate building a graph. These steps are:

- **inc**: increments the value by 1.
- **mul**: multiplies the value by 2.
- **WithState**: class that increments an internal counter, prints an output, and adds the input value to the current counter.

```
# mlrun: start-code

def inc(x):
    return x + 1

def mul(x):
    return x * 2

class WithState:
    def __init__(self, name, context, init_val=0):
        self.name = name
        self.context = context
        self.counter = init_val

    def do(self, x):
        self.counter += 1
        print(f"Echo: {self.name}, x: {x}, counter: {self.counter}")
        return x + self.counter

# mlrun: end-code
```

10.1.2 Create a function

Now take the code above and create an MLRun function called `serving-graph`.

```
import mlrun
fn = mlrun.code_to_function("simple-graph", kind="serving", image="mlrun/mlrun")
graph = fn.set_topology("flow")
```

10.1.3 Build the graph

Use `graph.to()` to chain steps. Use `.respond()` to mark that the output of that step is returned to the caller (as an http response). By default the graph is async with no response.

```
graph.to(name="+1", handler='inc')\
     .to(name="*2", handler='mul')\
     .to(name="(X+counter)", class_name='WithState').respond()
```

```
<mlrun.serving.states.TaskStep at 0x7f821e504450>
```

10.1.4 Visualize the graph

Using the `plot` method, you can visualize the graph.

```
graph.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7f82294f2f90>
```

10.1.5 Test the function

Create a mock server and test the graph locally. Since this graph accepts a numeric value as the input, that value is provided in the `body` parameter.

```
server = fn.to_mock_server()
server.test(body=5)
```

```
Echo: (X+counter), x: 12, counter: 1
```

```
13
```

Run the function again. This time, the counter should be 2 and the output should be 14.

```
server.test(body=5)
```

```
Echo: (X+counter), x: 12, counter: 2
```

```
14
```

10.1.6 Deploy the function

Use the `deploy` method to deploy the function.

```
fn.deploy(project='basic-graph-demo')
```

```
> 2021-11-08 07:30:21,571 [info] Starting remote function deploy
2021-11-08 07:30:21  (info) Deploying function
2021-11-08 07:30:21  (info) Building
2021-11-08 07:30:21  (info) Staging files and preparing base images
2021-11-08 07:30:21  (info) Building processor image
2021-11-08 07:30:26  (info) Build complete
2021-11-08 07:30:31  (info) Function deploy complete
> 2021-11-08 07:30:31,785 [info] successfully deployed function: {'internal_invocation_
↪urls': ['nuclio-basic-graph-demo-simple-graph.default-tenant.svc.cluster.local:8080'],
↪'external_invocation_urls': ['basic-graph-demo-simple-graph-basic-graph-demo.default-
↪tenant.app.aganefaibuzg.iguazio-cd2.com/']}
```

```
'http://basic-graph-demo-simple-graph-basic-graph-demo.default-tenant.app.aganefaibuzg.
↪iguazio-cd2.com/'
```

10.1.7 Test the deployed function

Use the `invoke` method to call the function.

```
fn.invoke('', body=5)
```

```
> 2021-11-08 07:30:43,241 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-basic-graph-demo-simple-graph.default-tenant.svc.cluster.local:8080/'}
```

```
13
```

```
fn.invoke('', body=5)
```

```
> 2021-11-08 07:30:48,359 [info] invoking function: {'method': 'POST', 'path': 'http://
↪nuclio-basic-graph-demo-simple-graph.default-tenant.svc.cluster.local:8080/'}
```

```
14
```

10.2 Use cases

In this section

- *Data and feature engineering*
- *Example of Simple model serving router*
- *Example of Advanced data processing and serving ensemble*
- *Example of NLP processing pipeline with real-time streaming*

In addition to the examples in this section, see the:

- Distributed (multi-function) pipeline example that details how to run a pipeline that consists of multiple serverless functions (connected using streams).
- Advanced Model Serving Graph Notebook Example that illustrates the flow, task, model, and ensemble router states; building tasks from custom handlers; classes and storey components; using custom error handlers; testing graphs locally; deploying a graph as a real-time serverless function.
- MLRun demos repository for additional use cases and full end-to-end examples, including fraud prevention using the Iguazio feature store, a mask detection demo, and converting existing ML code to an MLRun project.

10.2.1 Data and feature engineering (using the feature store)

You can build a feature set transformation using serving graphs.

High-level transformation logic is automatically converted to real-time serverless processing engines that can read from any online or offline source, handle any type of structures or unstructured data, run complex computation graphs and native user code. Iguazio's solution uses a unique multi-model database, serving the computed features consistently through many different APIs and formats (like files, SQL queries, pandas, real-time REST APIs, time-series, streaming), resulting in better accuracy and simpler integration.

Read more in *Feature store*, and Feature set transformations.

10.2.2 Example of a simple model serving router

Graphs are used for serving models with different transformations.

To deploy a serving function, you need to import or create the serving function, add models to it, and then deploy it.

```
import mlrun
# load the sklearn model serving function and add models to it
fn = mlrun.import_function('hub://v2_model_server')
fn.add_model("model1", model_path={model1-url})
fn.add_model("model2", model_path={model2-url})

# deploy the function to the cluster
fn.deploy()

# test the live model endpoint
fn.invoke('/v2/models/model1/infer', body={"inputs": [5]})
```

The Serving function supports the same protocol used in KFServing V2 and Triton Serving framework. To invoke the model, to use following url: `<function-host>/v2/models/model1/infer`.

See the **serving protocol specification** for details.

Note: Model url is either an MLRun model store object (starts with `store://`) or URL of a model directory (in NFS, s3, v3io, azure, for example `s3://{bucket}/{model-dir}`). Note that credentials might need to be added to the serving function via environment variables or MLRun secrets.

See the **scikit-learn classifier example**, which explains how to create/log MLRun models.

Writing your own serving class

You can implement your own model serving or data processing classes. All you need to do is:

1. Inherit the base model serving class.
2. Add your implementation for model `load()` (download the model file(s) and load the model into memory).
3. `predict()` (accept the request payload and return the prediction/inference results).

You can override additional methods: `preprocess`, `validate`, `postprocess`, `explain`. You can add custom API endpoints by adding the method `op_xx(event)` (which can be invoked by calling the `<model-url>/xx`, where operation = xx). See model class API.

For an example of writing the minimal serving functions, see Minimal sklearn serving function example.

See the full V2 Model Server (SKLearn) example that tests one or more classifier models against a held-out dataset.

10.2.3 Example of advanced data processing and serving ensemble

MLRun Serving graphs can host advanced pipelines that handle event/data processing, ML functionality, or any custom task. The following example demonstrates an asynchronous pipeline that pre-processes data, passes the data into a model ensemble, and finishes off with post processing.

For a complete example, see the Advanced graph example notebook.

Create a new function of type serving from code and set the graph topology to `async flow`.

```
import mlrun
function = mlrun.code_to_function("advanced", filename="demo.py",
                                  kind="serving", image="mlrun/mlrun",
                                  requirements=['storey'])
graph = function.set_topology("flow", engine="async")
```

Build and connect the graph (DAG) using the custom function and classes and plot the result. Add steps using the `step.to()` method (adds a new step after the current one), or using the `graph.add_step()` method.

If you want the error from the graph or the step to be fed into a specific step (catcher), use the `graph.error_handler()` (apply to all steps) or `step.error_handler()` (apply to a specific step).

Specify which step is the responder (returns the HTTP response) using the `step.respond()` method. If the responder is not specified, the graph is non-blocking.

```
# use built-in storey class or our custom Echo class to create and link Task steps
graph.to("storey.Extend", name="enrich", _fn='({"tag": "something"})') \
     .to(class_name="Echo", name="pre-process", some_arg='abc').error_handler("catcher")

# add an Ensemble router with two child models (routes), the "*" prefix mark it is a␣
↪router class
router = graph.add_step("*mlrun.serving.VotingEnsemble", name="ensemble", after="pre-
↪process")
router.add_route("m1", class_name="ClassifierModel", model_path=path1)
router.add_route("m2", class_name="ClassifierModel", model_path=path2)

# add the final step (after the router) which handles post processing and respond to the␣
↪client
graph.add_step(class_name="Echo", name="final", after="ensemble").respond()
```

(continues on next page)

(continued from previous page)

```
# add error handling step, run only when/if the "pre-process" step fail (keep after="")
graph.add_step(handler="error_catcher", name="catcher", full_event=True, after="")

# plot the graph (using Graphviz) and run a test
graph.plot(rankdir='LR')
```

Create a mock (test) server, and run a test. Use `wait_for_completion()` to wait for the async event loop to complete.

```
server = function.to_mock_server()
resp = server.test("/v2/models/m2/infer", body={"inputs": data})
server.wait_for_completion()
```

And deploy the graph as a real-time Nuclio serverless function with one command:

```
function.deploy()
```

Note: If you test a Nuclio function that has a serving graph with the async engine via the Nuclio UI, the UI might not display the logs in the output.

10.2.4 Example of an NLP processing pipeline with real-time streaming

In some cases it's useful to split your processing to multiple functions and use streaming protocols to connect those functions. In this example the data processing is in the first function/container and the NLP processing is in the second function. In this example the GPU contained in the second function.

See the full notebook example.

```
# define a new real-time serving function (from code) with an async graph
fn = mlrun.code_to_function("multi-func", filename="./data_prep.py", kind="serving",␣
↪image='mlrun/mlrun')
graph = fn.set_topology("flow", engine="async")

# define the graph steps (DAG)
graph.to(name="load_url", handler="load_url")\
     .to(name="to_paragraphs", handler="to_paragraphs")\
     .to("storey.FlatMap", "flatten_paragraphs", _fn="(event)")\
     .to(">>", "q1", path=internal_stream)\
     .to(name="nlp", class_name="ApplyNLP", function="enrich")\
     .to(name="extract_entities", handler="extract_entities", function="enrich")\
     .to(name="enrich_entities", handler="enrich_entities", function="enrich")\
     .to("storey.FlatMap", "flatten_entities", _fn="(event)", function="enrich")\
     .to(name="printer", handler="myprint", function="enrich")\
     .to(">>", "output_stream", path=out_stream)

# specify the "enrich" child function, add extra package requirements
child = fn.add_child_function('enrich', './nlp.py', 'mlrun/mlrun')
child.spec.build.commands = ["python -m pip install spacy",
                             "python -m spacy download en_core_web_sm"]
graph.plot()
```

Currently queues support iguazio v3io and Kafka streams.

10.3 Graph concepts and state machine

A graph is composed of the following:

- Step: A Step runs a function or class handler or a REST API call. MLRun comes with a list of pre-built steps that include data manipulation, readers, writers and model serving. You can also write your own steps using standard Python functions or custom functions/classes, or can be a external REST API (the special `$remote` class).
- Router: A special type of step is a router with routing logic and multiple child routes/models. The basic routing logic is to route to the child routes based on the `event.path`. More advanced or custom routing can be used, for example, the ensemble router sends the event to all child routes in parallel, aggregates the result and responds.
- Queue: A queue or stream that accepts data from one or more source steps and publishes to one or more output steps. Queues are best used to connect independent functions/containers. Queues can run in-memory or be implemented using a stream, which allows it to span processes/containers.

The Graph server has two modes of operation (topologies):

- Router topology (default): A minimal configuration with a single router and child tasks/routes. This can be used for simple model serving or single hop configurations.
- Flow topology: A full graph/DAG. The flow topology is implemented using two engines: async (the default) is based on Storey and asynchronous event loop; and `sync`, which supports a simple sequence of steps.

In this section

10.3.1 The Event object

The Graph state machine accepts an Event object (similar to a Nuclio Event) and passes it along the pipeline. An Event object hosts the event `body` along with other attributes such as `path` (http request path), `method` (GET, POST, ...), and`id` (unique event ID).

In some cases the events represent a record with a unique `key`, which can be read/set through the `event.key`. Records have associated `event.time` that, by default, is the arrival time, but can also be set by a step.

The Task steps are called with the `event.body` by default. If a task step needs to read or set other event elements (key, path, time, ...) you should set the task `full_event` argument to `True`.

Task steps support optional `input_path` and `result_path` attributes that allow controlling which portion of the event is sent as input to the step, and where to update the returned result.

For example, for an event body `{"req": {"body": "x"}}`, `input_path="req.body"` and `result_path="resp"` the step gets `"x"` as the input. The output after the step is `{"req": {"body": "x"}: "resp": <step output>}`. Note that `input_path` and `result_path` do not work together with `full_event=True`.

10.3.2 The Context object

The step classes are initialized with a `context` object (when they have `context` in their `__init__` args). The context is used to pass data and for interfacing with system services. The context object has the following attributes and methods.

Attributes:

- **logger**: Central logger (Nuclio logger when running in Nuclio).
- **verbose**: True if in verbose/debug mode.
- **root**: The graph object.
- **current_function**: When running in a distributed graph, the current child function name.

Methods:

- **get_param(key, default=None)**: Get the graph parameter by key. Parameters are set at the serving function (e.g. `function.spec.parameters = {"param1": "x"}`).
- **get_secret(key)**: Get the value of a project/user secret.
- **get_store_resource(uri, use_cache=True)**: Get the mlrun store object (data item, artifact, model, feature set, feature vector).
- **get_remote_endpoint(name, external=False)**: Return the remote nuclio/serving function http(s) endpoint given its [project/]function-name[:tag].
- **Response(headers=None, body=None, content_type=None, status_code=200)**: Create a nuclio response object, for returning detailed http responses.

Example, using the context:

```
if self.context.verbose:
    self.context.logger.info('my message', some_arg='text')
    x = self.context.get_param('x', 0)
```

10.3.3 Topology

Router

Once you have a serving function, you need to choose the graph topology. The default is `router` topology. With the `router` topology you can specify different machine learning models. Each model has a logical name. This name is used to route to the correct model when calling the serving function.

```
from sklearn.datasets import load_iris

# set the topology/router
graph = fn.set_topology("router")

# Add the model
fn.add_model("model1", class_name="ClassifierModel", model_path="https://s3.wasabisys.
↪com/iguazio/models/iris/model.pkl")

# Add additional models
#fn.add_model("model2", class_name="ClassifierModel", model_path="<path2>")

# create and use the graph simulator
```

(continues on next page)

(continued from previous page)

```
server = fn.to_mock_server()
x = load_iris()['data'].tolist()
result = server.test("/v2/models/model1/infer", {"inputs": x})

print(result)
```

```
> 2021-11-02 04:18:36,925 [info] model model1 was loaded
> 2021-11-02 04:18:36,926 [info] Initializing endpoint records
> 2021-11-02 04:18:36,965 [info] Loaded ['model1']
{'id': '6bd11e864805484ea888f58e478d1f91', 'model_name': 'model1', 'outputs': [0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,␣
↪2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,␣
↪2, 2]}
```

Flow

Using the `flow` topology, you can specify tasks, which typically manipulate the data. The most common scenario is pre-processing of data prior to the model execution.

Note: Once the topology is set, you cannot change an existing function topology.

In this topology, you build and connect the graph (DAG) by adding steps using the `step.to()` method, or by using the `graph.add_step()` method.

> The `step.to()` is typically used to chain steps together. `graph.add_step` can add steps anywhere on the graph and has `before` and `after` parameters to specify the location of the step.

```
fn2 = mlrun.code_to_function("serving_example_flow",
                             kind="serving",
                             image="mlrun/mlrun")

graph2 = fn2.set_topology("flow")

graph2_enrich = graph2.to("storey.Extend", name="enrich", _fn='({"tag": "something"})')

# add an Ensemble router with two child models (routes)
router = graph2.add_step(mlrun.serving.ModelRouter(), name="router", after="enrich")
router.add_route("m1", class_name="ClassifierModel", model_path='https://s3.wasabisys.
↪com/iguazio/models/iris/model.pkl')
router.respond()

# Add additional models
#router.add_route("m2", class_name="ClassifierModel", model_path=path2)

# plot the graph (using Graphviz)
graph2.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7fd46e4dda50>
```

```
fn2_server = fn2.to_mock_server()

result = fn2_server.test("/v2/models/m1/infer", {"inputs": x})

print(result)
```

```
> 2021-11-02 04:18:42,142 [info] model m1 was loaded
> 2021-11-02 04:18:42,142 [info] Initializing endpoint records
> 2021-11-02 04:18:42,183 [info] Loaded ['m1']
{'id': 'f713fd7eedeb431eba101b13c53a15b5'}
```

10.3.4 Building distributed graphs

Graphs can be hosted by a single function (using zero to n containers), or span multiple functions where each function can have its own container image and resources (replicas, GPUs/CPUs, volumes, etc.). It has a `root` function, which is where you configure triggers (http, incoming stream, cron, ...), and optional downstream child functions.

You can specify the `function` attribute in `Task` or `Router` steps. This indicates where this step should run. When the `function` attribute is not specified it runs on the root function. `function="*"` means the step can run in any of the child functions.

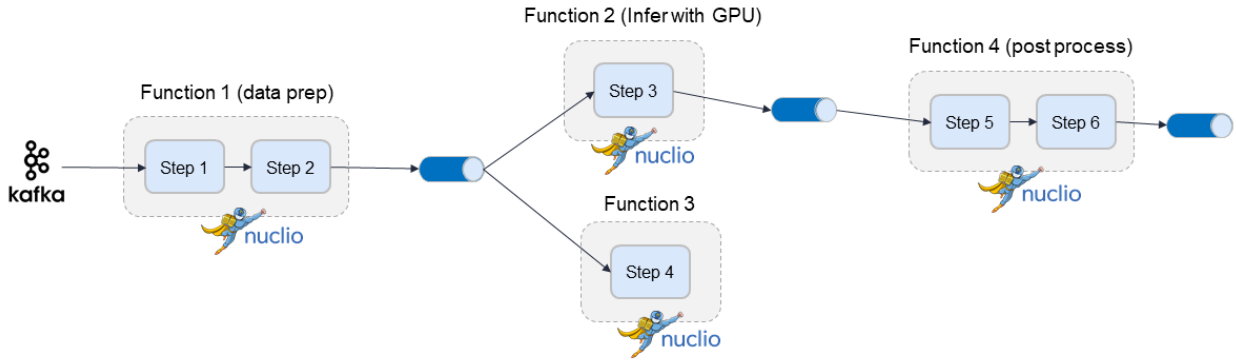
Steps on different functions should be connected using a `Queue` step (a stream).

Adding a child function:

```
fn.add_child_function('enrich',
                      './entity_extraction.ipynb',
                      image='mlrun/mlrun',
                      requirements=["storey", "sklearn"])
```

See a full example with child functions.

A distributed graph looks like this:



10.3.5 Error handling

Graph steps might raise an exception. If you want to have an error handling flow, you can specify an exception handling step/branch that is triggered on error. The error handler step receives the event that entered the failed step, with two extra attributes: `event.origin_state` indicates the name of the failed step; and `event.error` holds the error string.

Use the `graph.error_handler()` (apply to all steps) or `step.error_handler()` (apply to a specific step) if you want the error from the graph or the step to be fed into a specific step (catcher).

Example of setting an error catcher per step:

```
graph.add_step("MyClass", name="my-class", after="pre-process").error_handler("catcher")
graph.add_step("ErrHandler", name="catcher", full_event=True, after="")
```

Note: Additional steps can follow the `catcher` step.

Using the example in Model serving graph, you can add an error handler as follows:

```
graph2_enrich.error_handler("catcher")
graph2.add_step("ErrHandler", name="catcher", full_event=True, after="")
```

Now, display the graph again:

```
graph2.plot(rankdir='LR')
```

Exception stream

The graph errors/exceptions can be pushed into a special error stream. This is very convenient in the case of distributed and production graphs.

To set the exception stream address (using v3io streams uri):

```
fn_preprocess2.spec.error_stream = err_stream
```

10.4 Model serving graph

In this section

- *Serving Functions*
- *Topology*
- *Remote execution*
- *Examples*

10.4.1 Serving Functions

To start using a serving graph, you first need a serving function. A serving function contains the serving class code to run the model and all the code necessary to run the tasks. MLRun comes with a wide library of tasks. If you use just those, you don't have to add any special code to the serving function, you just have to provide the code that runs the model. For more information about serving classes see *Build your own model serving class*.

For example, the following code is a basic model serving class:

```
# mlrun: start-code
```

```
from cloudpickle import load
from typing import List
import numpy as np

import mlrun

class ClassifierModel(mlrun.serving.V2ModelServer):
    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model(".pkl")
        self.model = load(open(model_file, "rb"))

    def predict(self, body: dict) -> List:
        """Generate model predictions from sample."""
        feats = np.asarray(body["inputs"])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()
```

```
# mlrun: end-code
```

To obtain the serving function, use the `code_to_function` and specify `kind` to be `serving`.

```
fn = mlrun.code_to_function("serving_example",
                            kind="serving",
                            image="mlrun/mlrun")
```

10.4.2 Topology

Router

Once you have a serving function, you need to choose the graph topology. The default is `router` topology. With the `router` topology you can specify different machine learning models. Each model has a logical name. This name is used to route to the correct model when calling the serving function.

```
from sklearn.datasets import load_iris

# set the topology/router
graph = fn.set_topology("router")

# Add the model
fn.add_model("model1", class_name="ClassifierModel", model_path="https://s3.wasabisys.
↪com/iguazio/models/iris/model.pkl")
```

(continues on next page)

(continued from previous page)

```
# Add additional models
#fn.add_model("model2", class_name="ClassifierModel", model_path="<path2>")

# create and use the graph simulator
server = fn.to_mock_server()
x = load_iris()['data'].tolist()
result = server.test("/v2/models/model1/infer", {"inputs": x})

print(result)
```

```
> 2021-11-02 04:18:36,925 [info] model model1 was loaded
> 2021-11-02 04:18:36,926 [info] Initializing endpoint records
> 2021-11-02 04:18:36,965 [info] Loaded ['model1']
{'id': '6bd11e864805484ea888f58e478d1f91', 'model_name': 'model1', 'outputs': [0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,␣
↪2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,␣
↪2, 2]}
```

Flow

You can use the `flow` topology to specify tasks, which typically manipulates the data. The most common scenario is pre-processing of data prior to the model execution.

Note: Once the topology is set, you cannot change an existing function toplogy.

In this topology, you build and connect the graph (DAG) by adding steps using the `step.to()` method, or by using the `graph.add_step()` method.

> The `step.to()` is typically used to chain steps together. `graph.add_step` can add steps anywhere on the graph and has `before` and `after` parameters to specify the location of the step.

```
fn2 = mlrun.code_to_function("serving_example_flow",
                             kind="serving",
                             image="mlrun/mlrun")

graph2 = fn2.set_topology("flow")

graph2_enrich = graph2.to("storey.Extend", name="enrich", _fn='({"tag": "something"})')

# add an Ensemble router with two child models (routes)
router = graph2.add_step(mlrun.serving.ModelRouter(), name="router", after="enrich")
router.add_route("m1", class_name="ClassifierModel", model_path='https://s3.wasabisys.
↪com/iguazio/models/iris/model.pkl')
router.respond()

# Add additional models
```

(continues on next page)

(continued from previous page)

```
#router.add_route("m2", class_name="ClassifierModel", model_path=path2)

# plot the graph (using Graphviz)
graph2.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7fd46e4dda50>
```

```
fn2_server = fn2.to_mock_server()

result = fn2_server.test("/v2/models/m1/infer", {"inputs": x})

print(result)
```

```
> 2021-11-02 04:18:42,142 [info] model m1 was loaded
> 2021-11-02 04:18:42,142 [info] Initializing endpoint records
> 2021-11-02 04:18:42,183 [info] Loaded ['m1']
{'id': 'f713fd7eedeb431eba101b13c53a15b5'}
```

10.4.3 Remote execution

You can chain functions together with remote execution. This allows you to:

- Call existing functions from the graph and reuse them from other graphs.
- Scale up and down different components individually.

Calling a remote function can either use HTTP or via a queue (streaming).

HTTP

Calling a function using http uses the special `$remote` class. First deploy the remote function:

```
remote_func_name = "serving-example-flow"
project_name = "graph-basic-concepts"
fn_remote = mlrun.code_to_function(remote_func_name,
                                   project=project_name,
                                   kind="serving",
                                   image="mlrun/mlrun")

fn_remote.add_model("model1", class_name="ClassifierModel", model_path="https://s3.
↪wasabisys.com/iguazio/models/iris/model.pkl")

remote_addr = fn_remote.deploy()
```

```
> 2022-03-17 08:20:40,674 [info] Starting remote function deploy
2022-03-17 08:20:40  (info) Deploying function
2022-03-17 08:20:40  (info) Building
2022-03-17 08:20:40  (info) Staging files and preparing base images
2022-03-17 08:20:40  (info) Building processor image
2022-03-17 08:20:42  (info) Build complete
```

(continues on next page)

(continued from previous page)

```
2022-03-17 08:20:47  (info) Function deploy complete
> 2022-03-17 08:20:48,289 [info] successfully deployed function: {'internal_invocation_
↪urls': ['nuclio-graph-basic-concepts-serving-example-flow.default-tenant.svc.cluster.
↪local:8080'], 'external_invocation_urls': ['graph-basic-concepts-serving-example-flow-
↪graph-basic-concepts.default-tenant.app.maor-gcp2.iguazio-cd0.com/']}
```

Create a new function with a graph and call the remote function above:

```
fn_preprocess = mlrun.new_function("preprocess", kind="serving")
graph_preprocessing = fn_preprocess.set_topology("flow")

graph_preprocessing.to("storey.Extend", name="enrich", _fn='({"tag": "something"})').to(
                       "$remote", "remote_func", url=f'{remote_addr}v2/models/model1/
↪infer', method='put').respond()

graph_preprocessing.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7f57dc96a0d0>
```

```
fn3_server = fn_preprocess.to_mock_server()
my_data = '''{"inputs":[[5.1, 3.5, 1.4, 0.2],[7.7, 3.8, 6.7, 2.2]]}'''
result = fn3_server.test("/v2/models/my_model/infer", body=my_data)
print(result)
```

```
> 2022-03-17 08:20:48,374 [warning] run command, file or code were not specified
{'id': '3a1dd36c-e7de-45af-a0c4-72e3163ba92a', 'model_name': 'model1', 'outputs': [0, 2]}
```

Queue (streaming)

You can use queues to send events from one part of the graph to another and to decouple the processing of those parts. Queues are better suited to deal with bursts of events, since all the events are stored in the queue until they are processed.

V3IO stream example

The example below uses a V3IO stream, which is a fast real-time implementation of a stream that allows processing of events at very low latency.

```
%%writefile echo.py
def echo_handler(x):
    print(x)
    return x
```

```
Overwriting echo.py
```

Configure the streams

```
import os
streams_prefix = f"v3io:///users/{os.getenv('V3IO_USERNAME')}/examples/graph-basic-
↪concepts"
```

(continues on next page)

(continued from previous page)

```
input_stream = streams_prefix + "/in-stream"
out_stream = streams_prefix + "/out-stream"
err_stream = streams_prefix + "/err-stream"
```

Alternativey, use Kafka to configure the streams:

```
kafka_prefix = f"kafka://{broker}/"
internal_topic = kafka_prefix + "in-topic"
out_topic = kafka_prefix + "out-topic"
err_topic = kafka_prefix + "err-topic"
```

Create the graph. Note that in the `to` method the class name is specified to be >> or `$queue` to specify that this is a queue.

```
fn_preprocess2 = mlrun.new_function("preprocess", kind="serving")
fn_preprocess2.add_child_function('echo_func', './echo.py', 'mlrun/mlrun')

graph_preprocess2 = fn_preprocess2.set_topology("flow")

graph_preprocess2.to("storey.Extend", name="enrich", _fn='({"tag": "something"})')\
                 .to(">>", "input_stream", path=input_stream)\
                 .to(name="echo", handler="echo_handler", function="echo_func")\
                 .to(">>", "output_stream", path=out_stream, sharding_func="partition")

graph_preprocess2.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7f57c7907990>
```

```
from echo import *

fn4_server = fn_preprocess2.to_mock_server(current_function="*")

my_data = '''{"inputs": [[5.1, 3.5, 1.4, 0.2], [7.7, 3.8, 6.7, 2.2]], "partition": 0}'''

result = fn4_server.test("/v2/models/my_model/infer", body=my_data)

print(result)
```

```
> 2022-03-17 08:20:55,182 [warning] run command, file or code were not specified
{'id': 'a6efe8217b024ec7a7e02cf0b7850b91'}
{'inputs': [[5.1, 3.5, 1.4, 0.2], [7.7, 3.8, 6.7, 2.2]], 'tag': 'something'}
```

Kafka stream example

```
%%writefile echo.py
def echo_handler(x):
    print(x)
    return x
```

```
Overwriting echo.py
```

Configure the streams

```
import os

input_topic = "in-topic"
out_topic = "out-topic"
err_topic = "err-topic"

# replace this
brokers = "<broker IP>"
```

Create the graph. Note that in the `to` method the class name is specified to be >> or $queue to specify that this is a queue.

```
import mlrun

fn_preprocess2 = mlrun.new_function("preprocess", kind="serving")
fn_preprocess2.add_child_function('echo_func', './echo.py', 'mlrun/mlrun')

graph_preprocess2 = fn_preprocess2.set_topology("flow")

graph_preprocess2.to("storey.Extend", name="enrich", _fn='({"tag": "something"})')\
                 .to(">>", "input_stream", path=input_topic, kafka_bootstrap_
↪servers=brokers)\
                 .to(name="echo", handler="echo_handler", function="echo_func")\
                 .to(">>", "output_stream", path=out_topic, kafka_bootstrap_
↪servers=brokers)

graph_preprocess2.plot(rankdir='LR')

from echo import *

fn4_server = fn_preprocess2.to_mock_server(current_function="*")

fn4_server.set_error_stream(f"kafka://{brokers}/{err_topic}")

my_data = '''{"inputs":[[5.1, 3.5, 1.4, 0.2],[7.7, 3.8, 6.7, 2.2]]}'''

result = fn4_server.test("/v2/models/my_model/infer", body=my_data)

print(result)
```

10.4.4 Examples

NLP processing pipeline with real-time streaming

In some cases it's useful to split your processing to multiple functions and use streaming protocols to connect those functions.

See the full notebook example, where the data processing is in the first function/container and the NLP processing is in the second function. And the second function contains the GPU.

Currently queues support Iguazio v3io and Kafka streams.

10.5 Writing custom steps

The Graph executes built-in task classes, or task classes and functions that you implement. The task parameters include the following:

- `class_name` (str): the relative or absolute class name.
- `handler` (str): the function handler (if class_name is not specified it is the function handler).
- `**class_args`: a set of class `__init__` arguments.

For example, see the following simple `echo` class:

```
import mlrun
```

```
# mlrun: start
```

```
# echo class, custom class example
class Echo:
    def __init__(self, context, name=None, **kw):
        self.context = context
        self.name = name
        self.kw = kw

    def do(self, x):
        print("Echo:", self.name, x)
        return x
```

```
# mlrun: end
```

Test the graph: first convert the code to function, and then add the step to the graph:

```
fn_echo = mlrun.code_to_function("echo_function", kind="serving", image="mlrun/mlrun")

graph_echo = fn_echo.set_topology("flow")

graph_echo.to(class_name="Echo", name="pre-process", some_arg='abc')

graph_echo.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7f2d73584c90>
```

Create a mock server to test this locally:

```
echo_server = fn_echo.to_mock_server(current_function="*")

result = echo_server.test("", {"inputs": 123})

print(result)
```

```
{'id': '97397ea412334afdb5e4cb7d7c2e6dd3'}
Echo: pre-process {'inputs': 123}
```

For more information, see the Advanced model serving graph notebook example

You can use any Python function by specifying the handler name (e.g. `handler=json.dumps`). The function is triggered with the `event.body` as the first argument, and its result is passed to the next step.

Alternatively, you can use classes that can also store some step/configuration and separate the one time init logic from the per event logic. The classes are initialized with the `class_args`. If the class init args contain `context` or `name`, they are initialized with the graph context and the step name.

By default, the `class_name` and handler specify a class/function name in the `globals()` (i.e. this module). Alternatively, those can be full paths to the class (module.submodule.class), e.g. `storey.WriteToParquet`. You can also pass the module as an argument to functions such as `function.to_mock_server(namespace=module)`. In this case the class or handler names are also searched in the provided module.

When using classes the class event handler is invoked on every event with the `event.body`. If the Task step `full_event` parameter is set to `True` the handler is invoked and returns the full `event` object. If the class event handler is not specified, it invokes the class `do()` method.

If you need to implement async behavior, then subclass `storey.MapClass`.

10.6 Built-in steps

MlRun provides you with many built-in steps that you can use when building your graph.

Click on the step names in the following sections to see the full usage.

- *Base Operators*
- *Data Transformations*
- *External IO and data enrichment*
- *Sources*
- *Targets*
- *Models*
- *Routers*
- *Other*

10.6.1 Base Operators

Class name	Description
storey.transformations.Batch	Batches events. This step emits a batch every `max_events` events, or when `timeout` seconds have passed since the first event in the batch was received.
storey.transformations.Choice	Redirects each input element into one of the multiple downstreams.
storey.Extend	Adds fields to each incoming event.
storey.transformations.Filter	Filters events based on a user-provided function.
storey.transformations.FlatMap	Maps, or transforms, each incoming event into any number of events.
storey.steps.Flatten	Flatten is equivalent to FlatMap(lambda x: x).
storey.transformations.ForEach	Applies the given function on each event in the stream, and passes the original event downstream.
storey.transformations.MapClass	Similar to Map, but instead of a function argument, this class should be extended and its do() method overridden.
storey.transformations.MapWithState	Maps, or transforms, incoming events using a stateful user-provided function, and an initial state, which can be a database table.
storey.transformations.Partition	Partitions events by calling a predicate function on each event. Each processed event results in a Partitioned namedtuple of (left=Optional[Event], right=Optional[Event]).
storey.Reduce	Reduces incoming events into a single value that is returned upon the successful termination of the flow.
storey.transformations.SampleWindow	Emits a single event in a window of `window_size` events, in accordance with `emit_period` and `emit_before_termination`.

10.6.2 Data Transformations

Class name	Description
storey.AggregateByKey	Aggregates the data into the table object provided for later persistence, and outputs an event enriched with the requested aggregation features.
DateExtractor	Extract a date-time component.
mlrun.feature_store.Imputer	Replace None values with default values.
mlrun.feature_store.MapValues	Map column values to new values.
mlrun.feature_store.OneHotEncoder	Create new binary fields, one per category (one hot encoded).
mlrun.feature_store.SetEventMetadata	Set the event metadata (id, key, timestamp) from the event body.

10.6.3 External IO and data enrichment

Class name	Description
BatchHttpRequests	A class for calling remote endpoints in parallel.
mlrun.datastore.DataItem	Data input/output class abstracting access to various local/remote data sources.
storey.transformations.JoinWithTable	Joins each event with data from the given table.
JoinWithV3IOTable	Joins each event with a V3IO table. Used for event augmentation.
QueryByKey	Similar to to AggregateByKey, but this step is for serving only and does not aggregate the event.
RemoteStep	Class for calling remote endpoints.
storey.transformations.SendToHttp	Joins each event with data from any HTTP source. Used for event augmentation.

10.6.4 Sources

Class name	Description
mlrun.datastore.BigQuerySource	Reads Google BigQuery query results as input source for a flow.
mlrun.datastore.CSVSource	Reads a CSV file as input source for a flow.
DataframeSource	Reads data frame as input source for a flow.
mlrun.datastore.HttpSource	
mlrun.datastore.KafkaSource	Sets the kafka source for the flow.
mlrun.datastore.ParquetSource	Reads the Parquet file/dir as the input source for a flow.
mlrun.datastore.StreamSource	Sets the stream source for the flow. If the stream doesn't exist it creates it.

10.6.5 Targets

Class name	Description
mlrun.datastore.CSVTarget	Writes events to a CSV file.
mlrun.datastore.NoSqlTarget	Persists the data in *table* to its associated storage by key.
mlrun.datastore.ParquetTarget	The Parquet target storage driver, used to materialize feature set/vector data into parquet files.
mlrun.datastore.StreamTarget	Writes all incoming events into a V3IO stream.
storey.transformations.ToDataFrame	Create pandas data frame from events. Can appear in the middle of the flow, as opposed to ReduceToDataFrame.
mlrun.datastore.TSBDTarget	

10.6.6 Models

Class name	Description
mlrun.frameworks.onnx.ONNXModelServer	A model serving class for serving ONYX Models. A sub-class of the V2ModelServer class.
mlrun.frameworks.pytorch.PyTorchModelServer	A model serving class for serving PyTorch Models. A sub-class of the V2ModelServer class.
mlrun.frameworks.sklearn.SklearnModelServer	A model serving class for serving Sklearn Models. A sub-class of the V2ModelServer class.
mlrun.frameworks.tf_keras.TFKerasModelServer	A model serving class for serving TFKeras Models. A sub-class of the V2ModelServer class.
mlrun.frameworks.xgboost.XGBModelServer	A model serving class for serving XGB Models. A sub-class of the V2ModelServer class.

10.6.7 Routers

Class name	Description
mlrun.serving.EnrichmentModelRouter	Auto enrich the request with data from the feature store. The router input accepts a list of inference requests (each request can be a dict or a list of incoming features/keys). It enriches the request with data from the specified feature vector (`feature_vector_uri`).
mlrun.serving.EnrichmentVotingEnsemble	Auto enrich the request with data from the feature store. The router input accepts a list of inference requests (each request can be a dict or a list of incoming features/keys). It enriches the request with data from the specified feature vector (`feature_vector_uri`).
mlrun.serving.ModelRouter	Basic model router, for calling different models per each model path.
mlrun.serving.VotingEnsemble	An ensemble machine learning model that combines the prediction of several models.

10.6.8 Other

Class name	Description
mlrun.feature_store.FeaturesetValidator	Validate feature values according to the feature set validation policy.
ReduceToDataFrame	Builds a pandas DataFrame from events and returns that DataFrame on flow termination.

10.7 Demos and tutorials

Read these tutorials to get an even better understanding of serving graphs.

10.7.1 Distributed (multi-function) pipeline example

This example demonstrates how to run a pipeline that consists of multiple serverless functions (connected using streams).

In the pipeline example the request contains the a URL of a file. It loads the content of the file and breaks it into paragraphs (using the FlatMap class), and pushes the results to a queue/stream. The second function picks up the paragraphs and runs the NLP flow to extract the entities and push the results to the output stream.

Setting the stream URLs for the internal queue, the final output and error/exceptions stream:

```
streams_prefix = "v3io:///users/admin/"
internal_stream = streams_prefix + "in-stream"
out_stream = streams_prefix + "out-stream"
err_stream = streams_prefix + "err-stream"
```

Alternatively, using Kafka:

```
kafka_prefix = f"kafka://{broker}/"
internal_topic = kafka_prefix + "in-topic"
out_topic = kafka_prefix + "out-topic"
err_topic = kafka_prefix + "err-topic"
```

In either case, continue with:

```
# set up the environment
import mlrun
mlrun.set_environment(project="pipe")
```

```
> 2021-05-03 14:28:39,987 [warning] Failed resolving version info. Ignoring and using␣
↪defaults
> 2021-05-03 14:28:43,801 [warning] Unable to parse server or client version. Assuming␣
↪compatible: {'server_version': '0.6.3-rc4', 'client_version': 'unstable'}
```

```
('pipe', '/v3io/projects/{{run.project}}/artifacts')
```

```
# uncomment to install spacy requirements locally
# !pip install spacy
# !python -m spacy download en_core_web_sm
```

In this example

- *Create the pipeline*
- *Test the pipeline locally*
- *Deploy to the cluster*

Create the pipeline

The pipeline consists of two functions: data-prep and NLP. Each one has different package dependencies.

Create a file with data-prep graph steps:

```
%%writefile data_prep.py
import mlrun
import json

# load struct from a json file (event points to the url)
def load_url(event):
    url = event["url"]
    data = mlrun.get_object(url).decode("utf-8")
    return {"url": url, "doc": json.loads(data)}

def to_paragraphs(event):
    paragraphs = []
    url = event["url"]
    for i, paragraph in enumerate(event["doc"]):
        paragraphs.append(
            {"url": url, "paragraph_id": i, "paragraph": paragraph}
        )
    return paragraphs
```

```
Overwriting data_prep.py
```

Create a file with NLP graph steps (use spacy):

```
%%writefile nlp.py
import json
import spacy

def myprint(x):
    print(x)
    return x

class ApplyNLP:
    def __init__(self, context=None, spacy_dict="en_core_web_sm"):

        self.nlp = spacy.load(spacy_dict)

    def do(self, paragraph: dict):
        tokenized_paragraphs = []
        if isinstance(paragraph, (str, bytes)):
            paragraph = json.loads(paragraph)
        tokenized = {
            "url": paragraph["url"],
            "paragraph_id": paragraph["paragraph_id"],
            "tokens": self.nlp(paragraph["paragraph"]),
        }
        tokenized_paragraphs.append(tokenized)

        return tokenized_paragraphs

def extract_entities(tokens):
    paragraph_entities = []
    for token in tokens:
        entities = token["tokens"].ents
        for entity in entities:
            paragraph_entities.append(
                {
                    "url": token["url"],
                    "paragraph_id": token["paragraph_id"],
                    "entity": entity.ents,
                }
            )
    return paragraph_entities

def enrich_entities(entities):
    enriched_entities = []
    for entity in entities:
        enriched_entities.append(
            {
                "url": entity["url"],
                "paragraph_id": entity["paragraph_id"],
                "entity_text": entity["entity"][0].text,
                "entity_start_char": entity["entity"][0].start_char,
                "entity_end_char": entity["entity"][0].end_char,
                "entity_label": entity["entity"][0].label_,
            }
        )
```

(continues on next page)

(continued from previous page)

```
    return enriched_entities
```

```
Overwriting nlp.py
```

Build and show the graph:

Create the master function ("multi-func") with the `data_prep.py` source and an async graph topology. Add a pipeline of steps made of custom python handlers, classes and built-in classes (like `storey.FlatMap`).

The pipeline runs across two functions which are connected by a queue/stream (q1). Use the `function=` to specify which function runs the specified step. End the flow with writing to the output stream.

```
# define a new real-time serving function (from code) with an async graph
fn = mlrun.code_to_function("multi-func", filename="./data_prep.py", kind="serving",␣
↪image='mlrun/mlrun')
graph = fn.set_topology("flow", engine="async")

# define the graph steps (DAG)
graph.to(name="load_url", handler="load_url")\
     .to(name="to_paragraphs", handler="to_paragraphs")\
     .to("storey.FlatMap", "flatten_paragraphs", _fn="(event)")\
     .to(">>", "q1", path=internal_stream)\
     .to(name="nlp", class_name="ApplyNLP", function="enrich")\
     .to(name="extract_entities", handler="extract_entities", function="enrich")\
     .to(name="enrich_entities", handler="enrich_entities", function="enrich")\
     .to("storey.FlatMap", "flatten_entities", _fn="(event)", function="enrich")\
     .to(name="printer", handler="myprint", function="enrich")\
     .to(">>", "output_stream", path=out_stream)
```

```
<mlrun.serving.states.QueueState at 0x7f9e618f9910>
```

```
# specify the "enrich" child function, add extra package requirements
child = fn.add_child_function('enrich', './nlp.py', 'mlrun/mlrun')
child.spec.build.commands = ["python -m pip install spacy",
                             "python -m spacy download en_core_web_sm"]
graph.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7f9dd5dbed90>
```

Test the pipeline locally

Create an input file:

```
%%writefile in.json
["Born and raised in Queens, New York City, Trump attended Fordham University for two␣
↪years and received a bachelor's degree in economics from the Wharton School of the␣
↪University of Pennsylvania. He became president of his father Fred Trump's real estate␣
↪business in 1971, renamed it The Trump Organization, and expanded its operations to␣
↪building or renovating skyscrapers, hotels, casinos, and golf courses. Trump later␣
↪started various side ventures, mostly by licensing his name. Trump and his businesses␣
↪have been involved in more than 4,000 state and federal legal actions, including six␣
↪bankruptcies. He owned the Miss Universe brand of beauty pageants from 1996 to 2015,␣
↪and produced and hosted the reality television series The Apprentice from 2004 to 2015.
↪",
```

(continues on next page)

(continued from previous page)

```
 "Trump's political positions have been described as populist, protectionist,␣
↪isolationist, and nationalist. He entered the 2016 presidential race as a Republican␣
↪and was elected in a surprise electoral college victory over Democratic nominee␣
↪Hillary Clinton while losing the popular vote.[a] He became the oldest first-term U.S.␣
↪president[b] and the first without prior military or government service. His election␣
↪and policies have sparked numerous protests. Trump has made many false or misleading␣
↪statements during his campaign and presidency. The statements have been documented by␣
↪fact-checkers, and the media have widely described the phenomenon as unprecedented in␣
↪American politics. Many of his comments and actions have been characterized as␣
↪racially charged or racist."]
```

```
Overwriting in.json
```

Create a mock server (simulator) and test:

```
# tuggle verbosity if needed
fn.verbose = False
```

```
to
# create a mock server (simulator), specify to simulate all the functions in the␣
↪pipeline ("*")
server = fn.to_mock_server(current_function="*")
```

```
# push a sample request into the pipeline and see the results print out (by the printer␣
↪step)
resp = server.test(body={"url": "in.json"})
```

```
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'Queens', 'entity_start_char': 19,
↪'entity_end_char': 25, 'entity_label': 'GPE'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'New York City', 'entity_start_char
↪': 27, 'entity_end_char': 40, 'entity_label': 'GPE'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'Trump', 'entity_start_char': 42,
↪'entity_end_char': 47, 'entity_label': 'ORG'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'Fordham University', 'entity_start_
↪char': 57, 'entity_end_char': 75, 'entity_label': 'ORG'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'two years', 'entity_start_char':␣
↪80, 'entity_end_char': 89, 'entity_label': 'DATE'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'the Wharton School of the␣
↪University of Pennsylvania', 'entity_start_char': 141, 'entity_end_char': 193, 'entity_
↪label': 'ORG'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'Fred Trump', 'entity_start_char':␣
↪229, 'entity_end_char': 239, 'entity_label': 'PERSON'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': '1971', 'entity_start_char': 266,
↪'entity_end_char': 270, 'entity_label': 'DATE'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'The Trump Organization', 'entity_
↪start_char': 283, 'entity_end_char': 305, 'entity_label': 'ORG'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'more than 4,000', 'entity_start_
↪char': 529, 'entity_end_char': 544, 'entity_label': 'CARDINAL'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'six', 'entity_start_char': 588,
↪'entity_end_char': 591, 'entity_label': 'CARDINAL'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'Universe', 'entity_start_char':␣
↪624, 'entity_end_char': 632, 'entity_label': 'PERSON'}
```

(continues on next page)

(continued from previous page)

```
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': '1996 to 2015', 'entity_start_char
↪': 663, 'entity_end_char': 675, 'entity_label': 'DATE'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': 'The Apprentice', 'entity_start_char
↪': 731, 'entity_end_char': 745, 'entity_label': 'WORK_OF_ART'}
{'url': 'in.json', 'paragraph_id': 0, 'entity_text': '2004 to 2015', 'entity_start_char
↪': 751, 'entity_end_char': 763, 'entity_label': 'DATE'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'Trump', 'entity_start_char': 0,
↪'entity_end_char': 5, 'entity_label': 'ORG'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': '2016', 'entity_start_char': 122,
↪'entity_end_char': 126, 'entity_label': 'DATE'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'Republican', 'entity_start_char':␣
↪150, 'entity_end_char': 160, 'entity_label': 'NORP'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'Democratic', 'entity_start_char':␣
↪222, 'entity_end_char': 232, 'entity_label': 'NORP'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'Hillary Clinton', 'entity_start_
↪char': 241, 'entity_end_char': 256, 'entity_label': 'PERSON'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'first', 'entity_start_char': 312,
↪'entity_end_char': 317, 'entity_label': 'ORDINAL'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'U.S.', 'entity_start_char': 323,
↪'entity_end_char': 327, 'entity_label': 'GPE'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'first', 'entity_start_char': 349,
↪'entity_end_char': 354, 'entity_label': 'ORDINAL'}
{'url': 'in.json', 'paragraph_id': 1, 'entity_text': 'American', 'entity_start_char':␣
↪671, 'entity_end_char': 679, 'entity_label': 'NORP'}
```

```
server.wait_for_completion()
```

Deploy to the cluster

```
# add credentials to the data/streams
fn.apply(mlrun.platforms.v3io_cred())
child.apply(mlrun.platforms.v3io_cred())

# specify the error stream (to store exceptions from the functions)
fn.spec.error_stream = err_stream

# deploy as a set of serverless functions
fn.deploy()
```

```
> 2021-05-03 14:33:55,400 [info] deploy child function enrich ...
> 2021-05-03 14:33:55,427 [info] Starting remote function deploy
2021-05-03 14:33:55  (info) Deploying function
2021-05-03 14:33:55  (info) Building
2021-05-03 14:33:55  (info) Staging files and preparing base images
2021-05-03 14:33:55  (info) Building processor image
2021-05-03 14:34:02  (info) Build complete
2021-05-03 14:34:08  (info) Function deploy complete
> 2021-05-03 14:34:09,232 [info] function deployed, address=default-tenant.app.yh30.
↪iguazio-c0.com:32356
> 2021-05-03 14:34:09,233 [info] deploy root function multi-func ...
```

(continues on next page)

(continued from previous page)

```
> 2021-05-03 14:34:09,234 [info] Starting remote function deploy
2021-05-03 14:34:09  (info) Deploying function
2021-05-03 14:34:09  (info) Building
2021-05-03 14:34:09  (info) Staging files and preparing base images
2021-05-03 14:34:09  (info) Building processor image
2021-05-03 14:34:16  (info) Build complete
2021-05-03 14:34:22  (info) Function deploy complete
> 2021-05-03 14:34:22,891 [info] function deployed, address=default-tenant.app.yh30.
↪iguazio-c0.com:32046
```

```
'http://default-tenant.app.yh30.iguazio-c0.com:32046'
```

Listen on the output stream

You can use the SDK or CLI to listen on the output stream. Listening should be done in a separate console/notebook. Run:

```
mlrun watch-stream v3io:///users/admin/out-stream -j
```

or use the SDK:

```
from mlrun.platforms import watch_stream
watch_stream("v3io:///users/admin/out-stream", is_json=True)
```

Test the live function:

Note: The url must be a valid path to the input file.

```
fn.invoke('', body={"url": "v3io:///users/admin/pipe/in.json"})
```

```
{'id': '79354e45-a158-405f-811c-976e9cf4ab5e'}
```

10.7.2 Advanced model serving graph - notebook example

This example demonstrates how to use MLRun serving graphs and their advanced functionality including:

- Use of flow, task, model, and ensemble router states
- Build tasks from custom handlers, classes and storey components
- Use custom error handlers
- Test graphs locally
- Deploy the graph as a real-time serverless functions

In this example

- *Define functions and classes used in the graph*
- *Create a new serving function and graph*
- *Test the function locally*
- *Deploy the graph as a real-time serverless function*

Define functions and classes used in the graph

```
from cloudpickle import load
from typing import List
from sklearn.datasets import load_iris
import numpy as np

# model serving class example
class ClassifierModel(mlrun.serving.V2ModelServer):
    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

    def predict(self, body: dict) -> List:
        """Generate model predictions from sample."""
        feats = np.asarray(body['inputs'])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()

# echo class, custom class example
class Echo:
    def __init__(self, context, name=None, **kw):
        self.context = context
        self.name = name
        self.kw = kw

    def do(self, x):
        print("Echo:", self.name, x)
        return x

# error echo function, demo catching error and using custom function
def error_catcher(x):
        x.body = {"body": x.body, "origin_state": x.origin_state, "error": x.error}
        print("EchoError:", x)
        return None
```

```
# mark the end of the code section, DO NOT REMOVE !
# mlrun: end-code
```

Create a new serving function and graph

Use `code_to_function` to convert the above code into a serving function object and initialize a graph with async flow topology.

```
function = mlrun.code_to_function("advanced", kind="serving",
                                  image="mlrun/mlrun",
                                  requirements=['storey'])
graph = function.set_topology("flow", engine="async")
#function.verbose = True
```

Specify the sklearn models that are used in the ensemble.

```
models_path = 'https://s3.wasabisys.com/iguazio/models/iris/model.pkl'
path1 = models_path
path2 = models_path
```

Build and connect the graph (DAG) using the custom function and classes and plot the result. Add states using the `state.to()` method (adds a new state after the current one), or using the `graph.add_step()` method.

Use the `graph.error_handler()` (apply to all states) or `state.error_handler()` (apply to a specific state) if you want the error from the graph or the state to be fed into a specific state (catcher).

You can specify which state is the responder (returns the HTTP response) using the `state.respond()` method. If you don't specify the responder, the graph is non-blocking.

```
# use built-in storey class or our custom Echo class to create and link Task states
graph.to("storey.Extend", name="enrich", _fn='({"tag": "something"})') \
     .to(class_name="Echo", name="pre-process", some_arg='abc').error_handler("catcher")

# add an Ensemble router with two child models (routes). The "*" prefix mark it is a␣
↪router class
router = graph.add_step("*mlrun.serving.VotingEnsemble", name="ensemble", after="pre-
↪process")
router.add_route("m1", class_name="ClassifierModel", model_path=path1)
router.add_route("m2", class_name="ClassifierModel", model_path=path2)

# add the final step (after the router) that handles post processing and responds to the␣
↪client
graph.add_step(class_name="Echo", name="final", after="ensemble").respond()

# add error handling state, run only when/if the "pre-process" state fails (keep after="
↪")
graph.add_step(handler="error_catcher", name="catcher", full_event=True, after="")

# plot the graph (using Graphviz) and run a test
graph.plot(rankdir='LR')
```

```
<graphviz.dot.Digraph at 0x7fe03f6941d0>
```

Test the function locally

Create a test set.

```
import random
iris = load_iris()
x = random.sample(iris['data'].tolist(), 5)
```

Create a mock server (simulator) and test the graph with the test data.

> Note: The model and router objects support a common serving protocol API, see the protocol and API section.

```
server = function.to_mock_server()
resp = server.test("/v2/models/infer", body={"inputs": x})
server.wait_for_completion()
resp
```

```
> 2021-01-09 22:49:26,365 [info] model m1 was loaded
> 2021-01-09 22:49:26,493 [info] model m2 was loaded
> 2021-01-09 22:49:26,494 [info] Loaded ['m1', 'm2']
Echo: pre-process {'inputs': [[6.9, 3.2, 5.7, 2.3], [6.4, 2.7, 5.3, 1.9], [4.9, 3.1, 1.5,
↪ 0.1], [7.3, 2.9, 6.3, 1.8], [5.4, 3.7, 1.5, 0.2]], 'tag': 'something'}
Echo: final {'model_name': 'ensemble', 'outputs': [2, 2, 0, 2, 0], 'id':
↪'0ebcc5f6f4c24d4d83eb36391eaefb98'}
```

```
{'model_name': 'ensemble',
 'outputs': [2, 2, 0, 2, 0],
 'id': '0ebcc5f6f4c24d4d83eb36391eaefb98'}
```

Deploy the graph as a real-time serverless function

```
function.deploy()
```

```
> 2021-01-09 22:49:40,088 [info] Starting remote function deploy
2021-01-09 22:49:40  (info) Deploying function
2021-01-09 22:49:40  (info) Building
2021-01-09 22:49:40  (info) Staging files and preparing base images
2021-01-09 22:49:40  (info) Building processor image
2021-01-09 22:49:41  (info) Build complete
2021-01-09 22:49:47  (info) Function deploy complete
> 2021-01-09 22:49:48,422 [info] function deployed, address=default-tenant.app.yh55.
↪iguazio-cd0.com:32222
```

```
'http://default-tenant.app.yh55.iguazio-cd0.com:32222'
```

Invoke the remote function using the test data

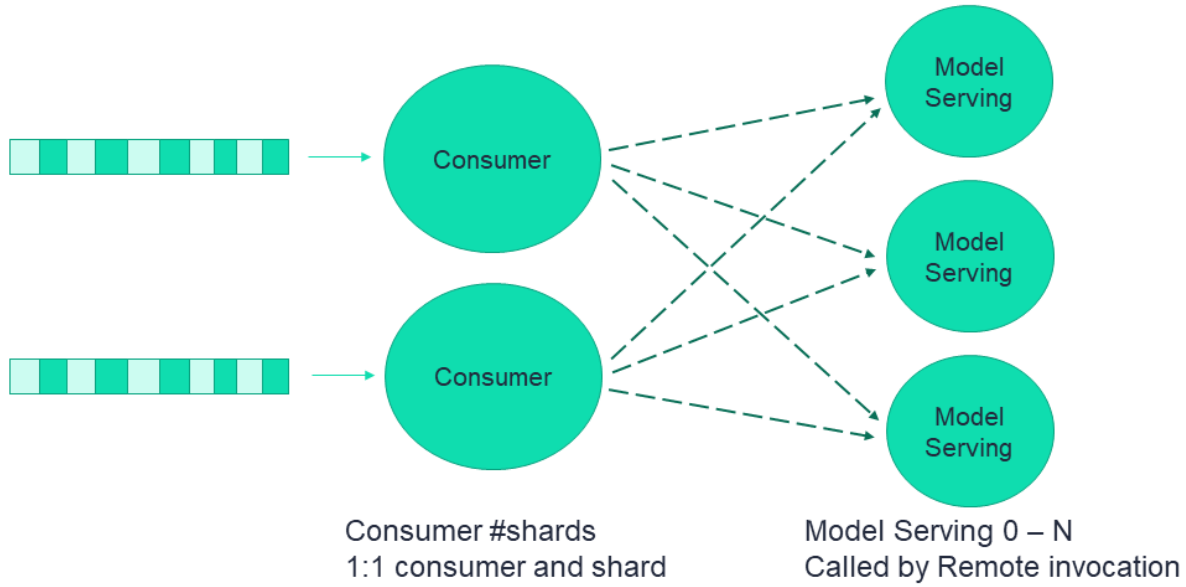
```
function.invoke("/v2/models/infer", body={"inputs": x})
```

```
{'model_name': 'ensemble',
 'outputs': [1, 2, 0, 0, 0],
 'id': '0ebcc5f6f4c24d4d83eb36391eaefb98'}
```

See the MLRun demos repository for additional use cases and full end-to-end examples, including Fraud Prevention using the Iguazio feature store, a mask detection demo, and converting existing ML code to an MLRun project.

10.8 Serving graph high availability configuration

This figure illustrates a simplistic flow of an MLRun serving graph with remote invocation:



As explained in *Real-time serving pipelines (graphs)*, the serving graph is based on Nuclio functions.

In this section

- *Using Nuclio with stream triggers*
- *Consumer function configuration*
- *Remote function retry mechanism*
- *Configuration considerations*

10.8.1 Using Nuclio with stream triggers

Nuclio can use different trigger types. When used with stream triggers, such as Kafka and V3IO, it uses a consumer group to continue reading from the last processed offset on function restart. This provides the "at least once" semantics for stateless functions. However, if the function does have state, such as persisting a batch of events to storage (e.g. parquet files, database) or if the function performs additional processing of events after the function handler returns, then the flow can get into situations where events seem to be lost. The mechanism of Window ACK provides a solution for such stateful event processing.

With Window ACK, the consumer group's committed offset is delayed by one window, committing the offset at (processed event num – window). When the function restarts (for any reason including scale-up or scale-down), it starts consuming from this last committed point.

The size of the required Window ACK is based on the number of events that could be in processing when the function terminates. You can define a window ACK per trigger (Kafka, V3IO stream, etc.). When used with a serving graph, the appropriate Window ACK size depends on the graph structure and should be calculated accordingly. The following sections explain the relevant considerations.

10.8.2 Consumer function configuration

A consumer function is essentially a Nuclio function with a stream trigger. As part of the trigger, you can set a consumer group.

When the consumer function is part of a graph then the consumer function's number of replicas is derived from the number of shards and is therefore nonconfigurable. The same applies to the number of workers in each replica, which is set to 1 and is not configurable.

The consumer function has one buffer per worker holding the incoming events that were received by the worker and are waiting to be processed. Once this buffer is full, events need to be processed so that the function is able to receive more events. The buffer size is configurable and is key to the overall configuration.

The buffer should be as small as possible. There is a trade-off between the buffer size and the latency. A larger buffer has lower latency but increases the recovery time after a failure, due to the high number of records that need to be reprocessed. To set the buffer size:

```
function.spec.parameters["source_args"] = {"buffer_size":  1}
```

The default `buffer_size` is 8.

10.8.3 Remote function retry mechanism

The required processing time of a remote function varies, depending on the function. The system assumes a processing time in the order of seconds, which affects the default configurations. However, some functions require a longer processing time. You can configure the timeout on both the caller and on the remote, as appropriate for your functions.

When an event is sent to the remote function, and no response is received by the configured (or default) timeout, or an error 500 (the remote function failed), or error 502, 503, or 504 (the remote function is too busy to handle the request at this time) is received, the caller retries the request, using the platform's exponential retry backoff mechanism. If the number of caller retries reaches the configured maximum number of retries, the event is pushed to the exception stream, indicating that this `event` did not complete successfully. You can look at the exception stream to see the functions that did not complete successfully.

Remote-function caller configuration

In a simplistic flow these are the consumer function defaults:

- Maximum retries: The default is 6, which is equivalent to about 3-4 minutes if all of the related parameters are at their default values. If you expect that some cases will require a higher number, for example, a new node needs to be scaled up depending on your cloud vendor, the instance type, and the zone you are running in, you might want to increase the number of retries.
- Remote step http timeout: The time interval the caller waits for a response from the remote before retrying the request. This value is affected by the remote function processing time.
- Max in flight: The maximum number of requests that each caller worker can send in parallel to the remote function. If the caller has more than one worker, each worker has its own Max in flight.

To set Max in flight, timeout, and retries:

```
RemoteStep(name="remote_scale", ...,  max_in_flight=2, timeout=100, retries=10)
```

Remote-function configuration

For the remote function, you can configure the following:

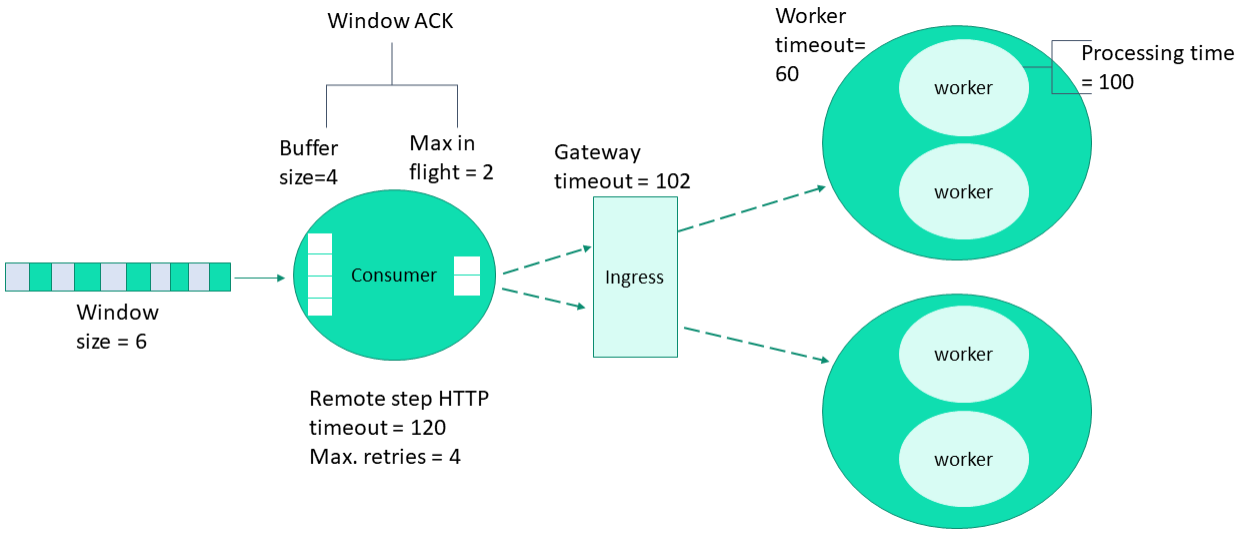
- Worker timeout: The maximum time interval, in seconds, an incoming request waits for an available worker. The worker timeout must be shorter than the gateway timeout. The default is 10.
- Gateway timeout: The maximum time interval, in seconds, the gateway waits for a response to a request. This determines when the ingress times out on a request. It must be slightly longer than the expected function processing time. The default is 60.

To set the buffer gateway timeout and worker timeout:

```
my_serving_func.with_http(gateway_timeout=125, worker_timeout=60)
```

10.8.4 Configuration considerations

The following figure zooms in on a single consumer and its workers and illustrates the various concepts and parameters that provide high availability, using a non-default configuration.



- Assume the processing time of the remote function is Pt, in seconds.
- `timeout`: Between <Pt+epsilon> and <Pt+`worker_timeout`>.
- Serving function
  - `gateway_timeout`: Pt+1 second (usually sufficient).
  - `worker_timeout`: The general rule is the greater of Pt/10 or 60 seconds. However, you should adjust the value according to your needs.
- `max_in_flight`: If the processing time is very high then `max_in_flight` should be low. Otherwise, there will be many retries.
- `ack_window_size`:
  - With 1 worker: The consumer `buffer_size`+`max_in_flight`, since it is per each shard and there is a single worker.
  - With >1 worker: The consumer (#workers x `buffer_size`)+`max_in_flight`

Make sure you thoroughly understand your serving graph and its functions before defining the `ack_window_size`. Its value depends on the entire graph flow. You need to understand which steps are parallel (branching) vs. sequential invocation. Another key aspect is that the number of workers affects the window size.

See the ack_window_size API.

For example:

- If a graph includes: consumer -> remote r1 -> remote r2
  - The window should be the sum of: consumer's buffer size + MIF to r1 + MIF to r2.
- If a graph includes: calling to remote r1 and r2 in parallel
  - The window should be set to: consumer's buffer size + max (MIF to r1, MIF to r2).

10.9 Error handling

Graph steps might raise an exception. If you want to have an error handling flow, you can specify an exception handling step/branch that is triggered on error. The error handler step receives the event that entered the failed step, with two extra attributes: `event.origin_state` indicates the name of the failed step; and `event.error` holds the error string.

Use the `graph.error_handler()` (apply to all steps) or `step.error_handler()` (apply to a specific step) if you want the error from the graph or the step to be fed into a specific step (catcher).

Example of setting an error catcher per step:

```
graph.add_step("MyClass", name="my-class", after="pre-process").error_handler("catcher")
graph.add_step("ErrHandler", name="catcher", full_event=True, after="")
```

Note

Additional steps can follow the catcher step.

Using the example in Model serving graph, you can add an error handler as follows:

```
graph2_enrich.error_handler("catcher")
graph2.add_step("ErrHandler", name="catcher", full_event=True, after="")
```

```
<mlrun.serving.states.TaskStep at 0x7fd46e557750>
```

Now, display the graph again:

```
graph2.plot(rankdir='LR')
```

```
<mlrun.serving.states.TaskStep at 0x7fd46e557750>
```

10.9.1 Exception stream

The graph errors/exceptions can be pushed into a special error stream. This is very convenient in the case of distributed and production graphs.

To set the exception stream address (using v3io streams uri):

```
fn_preprocess2.spec.error_stream = err_stream
```

CHAPTER

ELEVEN

MODEL MONITORING

By definition, ML models in production make inferences on constantly changing data. Even models that have been trained on massive data sets, with the most meticulously labelled data, start to degrade over time, due to concept drift. Changes in the live environment due to changing behavioral patterns, seasonal shifts, new regulatory environments, market volatility, etc., can have a big impact on a trained model's ability to make accurate predictions.

Model performance monitoring is a basic operational task that is implemented after an AI model has been deployed. Model monitoring includes:

- Built-in model monitoring: Machine learning model monitoring is natively built in to the Iguazio MLOps Platform, along with a wide range of model management features and ML monitoring reports. It monitors all of your models in a single, simple, dashboard.
- Automated drift detection: Automatically detects concept drift, anomalies, data skew, and model drift in real-time. Even if you are running hundreds of models simultaneously, you can be sure to spot and remediate the one that has drifted.
- Automated retraining: When drift is detected, Iguazio automatically starts the entire training pipeline to retrain the model, including all relevant steps in the pipeline. The output is a production-ready challenger model, ready to be deployed. This keeps your models up to date, automatically.
- Native feature store integration: Feature vectors and labels are stored and analyzed in the Iguazio feature store and are easily compared to the trained features and labels running as part of the model development phase, making it easier for data science teams to collaborate and maintain consistency between AI projects.

See full details and examples in Model monitoring.

CHAPTER

TWELVE

INGEST AND PROCESS DATA

MLRun provides a set of tools and capabilities to streamline the task of data ingestion and processing. For an end-to-end framework for data processing, management, and serving, MLRun has the feature-store capabilities, which are described in *Feature store*. However, in many cases the full feature-store capabilities are not needed, in which cases MLRun provides a set of utilities to facilitate data ingestion, collection and processing.

In this section

12.1 Using data sources and items

In this section

- *Connecting to data sources*
- *Data processing*

12.1.1 Connecting to data sources

Accessing data from multiple source types is possible through MLRun's `DataItem` object. This object plugs into the data-stores framework to connect to various types of data sources and download content. For example, to download data which is stored on S3 and load it into a `DataFrame`, use the following code:

```
# Access object in AWS S3, in the "input-data" bucket
import mlrun

# Access credentials
os.environ["AWS_ACCESS_KEY_ID"] = "<access key ID>"
os.environ["AWS_SECRET_ACCESS_KEY"] = "<access key>"

source_url = "s3://input-data/input_data.csv"

input_data = mlrun.get_dataitem(source_url).as_df()
```

This code runs locally (for example, in Jupyter) and relies on environment variables to supply credentials for data access. See *Data stores* for more info on the available data-stores, accessing them locally and remotely, and how to provide credentials for connecting.

Running the code locally is very useful for easy debugging and development of the code. When the code moves to a stable status, it is usually recommended to run it "remotely" on a pod running in the Kubernetes cluster. This allows setting up specific resources to the processing pod (such as memory, CPU and execution priority).

MLRun provides facilities to create `DataItem` objects as inputs to running code. For example, this is a basic data ingestion function:

```
def ingest_data(context, source_url: mlrun.DataItem):
    # Load the data from its source, and convert to a DataFrame
    df = source_url.as_df()

    # Perform data cleaning and processing
    # ...

    # Save the processed data to the artifact store
    context.log_dataset('cleaned_data', df=df, format='csv')
```

This code can be placed in a python file, or as a cell in the Python notebook. For example, if the code above was saved to a file, the following code creates an MLRun function from it and executes it remotely in a pod:

```
# create a function from py or notebook (ipynb) file, specify the default function␣
↪handler
ingest_func = mlrun.code_to_function(name='ingest_data', filename='./ingest_data.py',
                                     kind='job', image='mlrun/mlrun')

source_url = "s3://input-data/input_data.csv"

ingest_data_run = ingest_func.run(name='ingest_data',
                                  handler=ingest_data,
                                  inputs={'source_url': source_url},
                                  local=False)
```

As the `source_url` is part of the function's `inputs`, MLRun automatically wraps it up with a `DataItem`. The output is logged to the function's `artifact_path`, and can be obtained from the run result:

```
cleaned_data_frame = ingest_data_run.artifact('cleaned_data').as_df()
```

Note that running the function remotely may require attaching storage to the function, as well as passing storage credentials through project secrets. See the following pages for more details:

1. *Attach storage to functions*
2. *Working with secrets*

12.1.2 Data processing

Once the data is imported from its source, it can be processed using any framework. MLRun natively supports working with Pandas DataFrames and converting from and to its `DataItem` object.

For distributed processing of very large datasets, MLRun integrates with the Spark processing engine, and provides facilities for executing pySpark code using a Spark service (which can be deployed by the platform when running MLRun as part of an Iguazio system) or through submitting the processing task to Spark-operator. The following page provides additional details and code-samples:

1. Spark operator

In a similar manner, Dask can be used for parallel processing of the data. To read data as a Dask `DataFrame`, use the following code:

```
import dask.dataframe as dd

data_item = mlrun.get_dataitem(source_url)
dask_df: dd.DataFrame = data_item.as_df(df_module=dd)
```

12.2 Logging datasets

Storing datasets is important in order to have a record of the data that was used to train models, as well as storing any processed data. MLRun comes with built-in support for the DataFrame format. MLRun not only stores the DataFrame, but it also provides information about the data, such as statistics.

The simplest way to store a dataset is with the following code:

```
context.log_dataset(key='my_data', df=df)
```

Where `key` is the name of the artifact and `df` is the DataFrame. By default, MLRun stores a short preview of 20 lines. You can change the number of lines by changing the value of the `preview` parameter.

MLRun also calculates statistics on the DataFrame on all numeric fields. You can enable statistics regardless to the DataFrame size by setting the `stats` parameter to `True`.

12.2.1 Logging a dataset from a job

The following example shows how to work with datasets from a job:

```
from os import path
from mlrun.execution import MLClientCtx
from mlrun.datastore import DataItem

# Ingest a data set into the platform
def get_data(context: MLClientCtx, source_url: DataItem, format: str = 'csv'):

    iris_dataset = source_url.as_df()

    target_path = path.join(context.artifact_path, 'data')
    # Optionally print data to your logger
    context.logger.info('Saving Iris data set to {} ...'.format(target_path))

    # Store the data set in your artifacts database
    context.log_dataset('iris_dataset', df=iris_dataset, format=format,
                        index=False, artifact_path=target_path)
```

You can run this function locally or as a job. For example, to run it locally:

```
from os import path
from mlrun import new_project, run_local, mlconf

project_name = 'my-project'
project_path = path.abspath('conf')
project = new_project(project_name, project_path, init_git=True)
```

(continues on next page)

(continued from previous page)

```
# Target location for storing pipeline artifacts
artifact_path = path.abspath('jobs')
# MLRun DB path or API service URL
mlconf.dbpath = mlconf.dbpath or 'http://mlrun-api:8080'

source_url = 'https://s3.wasabisys.com/iguazio/data/iris/iris_dataset.csv'
# Run get-data function locally
get_data_run = run_local(name='get_data',
                         handler=get_data,
                         inputs={'source_url': source_url},
                         project=project_name,
                         artifact_path=artifact_path)
```

The dataset location is returned in the `outputs` field, therefore you can get the location by calling `get_data_run.artifact('iris_dataset')` to get the dataset itself.

```
# Read your data set
get_data_run.artifact('iris_dataset').as_df()

# Visualize an artifact in Jupyter (image, html, df, ..)
get_data_run.artifact('confusion-matrix').show()
```

The dataset returned from the run result is of the `DataItem` type. It allows access to the data itself as a Pandas Dataframe by calling the `dataset.as_df()`. It also contains the metadata of the artifact, accessed by the using `dataset.meta`. This artifact metadata object contains in it the statistics calculated, the schema of the dataset and other fields describing the dataset. For example, call `dataset.meta.stats` to obtain the data statistics.

12.3 Ingest data using the feature store

Define the source and material targets, and start the ingestion process (as *local process*, *using an MLRun job*, *real-time ingestion*, or *incremental ingestion*).

Data can be ingested as a batch process either by running the ingest command on demand or as a scheduled job. Batch ingestion can be done locally (i.e. running as a python process in the Jupyter pod) or as an MLRun job.

The data source can be a DataFrame or files (e.g. csv, parquet). Files can be either local files residing on a volume (e.g. v3io), or remote (e.g. S3, Azure blob). MLRun also supports Google BigQuery as a data source. If you define a transformation graph, then the ingestion process runs the graph transformations, infers metadata and stats, and writes the results to a target data store.

When targets are not specified, data is stored in the configured default targets (i.e. NoSQL for real-time and Parquet for offline).

Limitations

- Do not name columns starting with either `_` or `aggr_`. They are reserved for internal use. See also general limitations in Attribute name restrictions.
- When using the pandas engine, do not use spaces ( ) or periods (`.`) in the column names. These cause errors in the ingestion.

In this section

- *Inferring data*
- *Ingest data locally*
- *Ingest data using an MLRun job*
- *Real-time ingestion*
- *Incremental ingestion*
- *Data sources*
- *Target stores*

12.3.1 Inferring data

There are two types of inferring:

- Metadata/schema: This is responsible for describing the dataset and generating its meta-data, such as deducing the data-types of the features and listing the entities that are involved. Options belonging to this type are `Entities`, `Features` and `Index`. The `InferOptions` class has the `InferOptions.schema()` function which returns a value containing all the options of this type.
- Stats/preview: Ths related to calculating statistics and generating a preview of the actual data in the dataset. Options of this type are `Stats`, `Histogram` and `Preview`.

The `InferOptions class` has the following values: class InferOptions: Null = 0 Entities = 1 Features = 2 Index = 4 Stats = 8 Histogram = 16 Preview = 32

The `InferOptions class` basically translates to a value that can be a combination of the above values. For example, passing a value of 24 means `Stats` + `Histogram`.

When simultanesouly ingesting data and requesting infer options, part of the data might be ingested twice: once for inferring metadata/stats and once for the actual ingest. This is normal behavior.

12.3.2 Ingest data locally

Use a Feature Set to create the basic feature-set definition and then an ingest method to run a simple ingestion "locally" in the Jupyter Notebook pod.

```
# Simple feature set that reads a csv file as a dataframe and ingests it as is
stocks_set = FeatureSet("stocks", entities=[Entity("ticker")])
stocks = pd.read_csv("stocks.csv")
df = ingest(stocks_set, stocks)

# Specify a csv file as source, specify a custom CSV target
source = CSVSource("mycsv", path="stocks.csv")
targets = [CSVTarget("mycsv", path="./new_stocks.csv")]
ingest(measurements, source, targets)
```

To learn more about ingest go to *ingest*.

12.3.3 Ingest data using an MLRun job

Use the ingest method with the `run_config` parameter for running the ingestion process using a serverless MLRun job. By doing that, the ingestion process runs on its own pod or service on the kubernetes cluster. This option is more robust since it can leverage the cluster resources, as opposed to running within the Jupyter Notebook. It also enables you to schedule the job or use bigger/faster resources.

```
# Running as remote job
stocks_set = FeatureSet("stocks", entities=[Entity("ticker")])
config = RunConfig(image='mlrun/mlrun')
df = ingest(stocks_set, stocks, run_config=config)
```

12.3.4 Real-time ingestion

Real-time use cases (e.g. real time fraud detection) require feature engineering on live data (e.g. z-score calculation) while the data is coming from a streaming engine (e.g. kafka) or a live http endpoint. The feature store enables you to start real-time ingestion service. When running the *deploy_ingestion_service* the feature store creates an elastic real-time serverless function (the nuclio function) that runs the pipeline and stores the data results in the "offline" and "online" feature store by default. There are multiple data source options including http, kafka, kinesis, v3io stream, etc. Due to the asynchronous nature of feature store's execution engine, errors are not returned, but rather logged and pushed to the defined error stream.

```
# Create a real time function that receives http requests
# the "ingest" function runs the feature engineering logic on live events
source = HTTPSource()
func = mlrun.code_to_function("ingest", kind="serving").apply(mount_v3io())
config = RunConfig(function=func)
fstore.deploy_ingestion_service(my_set, source, run_config=config)
```

To learn more about deploy_ingestion_service go to *deploy_ingestion_service*.

12.3.5 Incremental ingestion

You can schedule an ingestion job for a feature set on an ongoing basis. The first scheduled job runs on all the data in the source and the subsequent jobs ingest only the deltas since the previous run (from the last timestamp of the previous run until `datetime.now`). Example:

```
cron_trigger = "* */1 * * *" #will run every hour
source = ParquetSource("myparquet", path=path, time_field="time", schedule=cron_trigger)
feature_set = fs.FeatureSet(name=name, entities=[fs.Entity("first_name")], timestamp_key=
↪"time",)
fs.ingest(feature_set, source, run_config=fs.RunConfig())
```

The default value for the `overwrite` parameter in the ingest function for scheduled ingest is `False`, meaning that the target from the previous ingest is not deleted. For the storey engine, the feature is currently implemented for ParquetSource only. (CsvSource will be supported in a future release). For Spark engine, other sources are also supported.

12.3.6 Data sources

For batch ingestion the feature store supports dataframes and files (i.e. csv & parquet). The files can reside on S3, NFS, Azure blob storage, or the Iguazio platform. MLRun also supports Google BigQuery as a data source. When working with S3/Azure, there are additional requirements. Use pip install mlrun[s3] or pip install mlrun[azure-blob-storage] to install them.

- Azure: define the environment variable `AZURE_STORAGE_CONNECTION_STRING`.
- S3: define `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` and `AWS_BUCKET`.

For real time ingestion the source can be http, kafka or v3io stream, etc. When defining a source, it maps to nuclio event triggers.

You can also create a custom `source` to access various databases or data sources.

12.3.7 Target stores

By default, the feature sets are saved in parquet and the Iguazio NoSQL DB (`NoSqlTarget`). The parquet file is ideal for fetching large set of data for training while the key value is ideal for an online application since it supports low latency data retrieval based on key access.

Note

When working with the Iguazio MLOps platform the default feature set storage location is under the "Projects" container: `<project name>/fs/..` folder. The default location can be modified in mlrun config or specified per ingest operation. The parquet/csv files can be stored in NFS, S3, Azure blob storage, Redis, and on Iguazio DB/FS.

Redis target store (Tech preview)

The Redis online target is called, in MLRun, `RedisNoSqlTarget`. The functionality of the `RedisNoSqlTarget` is identical to the `NoSqlTarget` except for:

- The `RedisNoSqlTarget` does not support the spark engine, (only supports the storey engine).
- The `RedisNoSqlTarget` accepts path parameter in the form `<redis|rediss>://[<username>]:[<password>]@<host>[:port]` For example: `rediss://:abcde@localhost:6379` creates a redis target, where:
  - The client/server protocol (rediss) is TLS protected (vs. "redis" if no TLS is established)
  - The server is password protected (password="abcde")
  - The server location is localhost port 6379.
- A default path can be configured in redis.url config (mlrun client has priority over mlrun server), and can be overwritten by `MLRUN_REDIS__URL` env var.
- Two types of Redis servers are supported: StandAlone and Cluster (no need to specify the server type in the config).
- A feature set supports one online target only. Therefore `RedisNoSqlTarget` and `NoSqlTarget` cannot be used as two targets of the same feature set.

To use the Redis online target store, you can either change the default to be parquet and Redis, or you can specify the Redis target explicitly each time with the path parameter, for example: `RedisNoSqlTarget(path ="redis://1.2.3.4:6379")`

12.4 Ingest features with Spark

The feature store supports using Spark for ingesting, transforming, and writing results to data targets. When using Spark, the internal execution graph is executed synchronously by utilizing a Spark session to perform read and write operations, as well as potential transformations on the data. Executing synchronously means that the source data is fully read into a data-frame that is processed, writing the output to the targets defined.

To use Spark as the transformation engine in ingestion, follow these steps:

When constructing the *FeatureSet* object, pass an `engine` parameter and set it to `spark`. For example:

```
feature_set = fstore.FeatureSet("stocks", entities=[fstore.Entity("ticker")], engine=
↪"spark")
```

To use a remote execution engine, pass a `RunConfig` object as the `run_config` parameter for the `ingest` API. The actual remote function to execute depends on the object passed:

- A default `RunConfig`, in which case the ingestion code either generates a new MLRun function runtime of type `remote-spark`, or utilizes the function specified in `feature_set.spec.function` (in which case, it has to be of runtime type `remote-spark` or `spark`).
- A `RunConfig` that has a function configured within it. As mentioned, the function runtime must be of type `remote-spark` or `spark`.

Spark execution can be done locally, utilizing a local Spark session provided to the ingestion call. To use a local Spark session, pass a Spark session context when calling the *ingest()* function, as the `spark_context` parameter. This session is used for data operations and transformations.

See code examples in:

- *Local Spark ingestion example*
- *Remote Spark ingestion example*
- *Spark operator ingestion example*
- *Spark dataframe ingestion example*
- *Spark over S3 full flow example*
- *Spark ingestion from Snowflake example*
- *Spark ingestion from Azure example*

12.4.1 Local Spark ingestion example

A local Spark session is a session running in the Jupyter service. The following code executes data ingestion using a local Spark session.

When using a local Spark session, the `ingest` API would wait for its completion.

```
import mlrun
from mlrun.datastore.sources import CSVSource
import mlrun.feature_store as fstore
from pyspark.sql import SparkSession

mlrun.get_or_create_project(name="stocks")
feature_set = fstore.FeatureSet("stocks", entities=[fstore.Entity("ticker")], engine=
↪"spark")
```

(continues on next page)

(continued from previous page)

```
# add_aggregation can be used in conjunction with Spark
feature_set.add_aggregation("price", ["min", "max"], ["1h"], "10m")

source = CSVSource("mycsv", path="v3io:///projects/stocks.csv")

# Execution using a local Spark session
spark = SparkSession.builder.appName("Spark function").getOrCreate()
fstore.ingest(feature_set, source, spark_context=spark)
```

12.4.2 Remote Spark ingestion example

Remote Spark refers to a session running from another service, for example, the Spark standalone service or the Spark operator service. When using remote execution the MLRun run execution details are returned, allowing tracking of its status and results.

The following code should be executed only once to build the remote spark image before running the first ingest. It may take a few minutes to prepare the image.

```
from mlrun.runtimes import RemoteSparkRuntime
RemoteSparkRuntime.deploy_default_image()
```

Remote ingestion:

```
# mlrun: start-code
```

```
from mlrun.feature_store.api import ingest
def ingest_handler(context):
    ingest(mlrun_context=context) # The handler function must call ingest with the mlrun_
↪context
```

You can run your PySpark code for ingesting data into the feature store by adding:

```
def my_spark_func(df, context=None):
    return df.filter("bid>55") # PySpark code
```

```
# mlrun: end-code
```

```
from mlrun.datastore.sources import CSVSource
from mlrun import code_to_function
import mlrun.feature_store as fstore

feature_set = fstore.FeatureSet("stock-quotes", entities=[fstore.Entity("ticker")],␣
↪engine="spark")

source = CSVSource("mycsv", path="v3io:///projects/quotes.csv")

spark_service_name = "iguazio-spark-service" # As configured & shown in the Iguazio␣
↪dashboard

feature_set.graph.to(name="s1", handler="my_spark_func")
```

(continues on next page)

(continued from previous page)

```
my_func = code_to_function("func", kind="remote-spark")
config = fstore.RunConfig(local=False, function=my_func, handler="ingest_handler")
fstore.ingest(feature_set, source, run_config=config, spark_context=spark_service_name)
```

12.4.3 Spark operator ingestion example

When running with a Spark operator, the MLRun execution details are returned, allowing tracking of the job's status and results. Spark operator ingestion is always executed remotely.

The following code should be executed only once to build the spark job image before running the first ingest. It may take a few minutes to prepare the image.

```
from mlrun.runtimes import Spark3Runtime
Spark3Runtime.deploy_default_image()
```

Spark operator ingestion:

```
# mlrun: start-code

from mlrun.feature_store.api import ingest

def ingest_handler(context):
    ingest(mlrun_context=context) # The handler function must call ingest with the mlrun_
↪context

# You can add your own PySpark code as a graph step:
def my_spark_func(df, context=None):
    return df.filter("bid>55") # PySpark code

# mlrun: end-code
```

```
from mlrun.datastore.sources import CSVSource
from mlrun import code_to_function
import mlrun.feature_store as fstore

feature_set = fstore.FeatureSet("stock-quotes", entities=[fstore.Entity("ticker")],␣
↪engine="spark")

source = CSVSource("mycsv", path="v3io:///projects/quotes.csv")

feature_set.graph.to(name="s1", handler="my_spark_func")

my_func = code_to_function("func", kind="spark")

my_func.with_driver_requests(cpu="200m", mem="1G")
my_func.with_executor_requests(cpu="200m", mem="1G")
my_func.with_igz_spark()

# Enables using the default image (can be replace with specifying a specific image with .
↪spec.image)
my_func.spec.use_default_image = True
```

(continues on next page)

(continued from previous page)

```
# Not a must - default: 1
my_func.spec.replicas = 2

# If needed, sparkConf can be modified like this:
# my_func.spec.spark_conf['spark.specific.config.key'] = 'value'

config = fstore.RunConfig(local=False, function=my_func, handler="ingest_handler")
fstore.ingest(feature_set, source, run_config=config)
```

12.4.4 Spark dataframe ingestion example

The following code executes local data ingestion from a spark dataframe (Spark dataframe Ingestion cannot be executed remotely.) The specified dataframe should be associated with `spark_context`.

```
from pyspark.sql import SparkSession
import mlrun.feature_store as fstore

columns = ["id", "count"]
data = [("a", "12"), ("b", "14"), ("c", "88")]

spark = SparkSession.builder.appName('example').getOrCreate()
df = spark.createDataFrame(data).toDF(*columns)

fset = fstore.FeatureSet("myset", entities=[fstore.Entity("id")], engine="spark")

fstore.ingest(fset, df, spark_context=spark)

spark.stop()
```

12.4.5 Spark over S3 - full flow example

For Spark to work with S3, it requires several properties to be set. Spark over S3 can be executed both remotely and locally, as long as access credentials to the S3 objects are available to it. The following example writes a feature set to S3 in the parquet format in a remote k8s job:

One-time setup:

1. Deploy the default image for your job (this takes several minutes but should be executed only once per cluster for any MLRun/Iguazio upgrade):

   ```
   from mlrun.runtimes import RemoteSparkRuntime
   RemoteSparkRuntime.deploy_default_image()
   ```

2. Store your S3 credentials in a k8s secret:

   ```
   import mlrun
   secrets = {'s3_access_key': AWS_ACCESS_KEY, 's3_secret_key': AWS_SECRET_KEY}
   mlrun.get_run_db().create_project_secrets(
       project = "uhuh-proj",
       provider=mlrun.api.schemas.SecretProviderName.kubernetes,
   ```

(continues on next page)

(continued from previous page)

```
    secrets=secrets
)
```

Ingestion job code (to be executed in the remote pod):

```
# mlrun: start-code

from pyspark import SparkConf
from pyspark.sql import SparkSession


from mlrun.feature_store.api import ingest
def ingest_handler(context):
    conf = (SparkConf()
            .set("spark.hadoop.fs.s3a.path.style.access", True)
            .set("spark.hadoop.fs.s3a.access.key", context.get_secret('s3_access_key'))
            .set("spark.hadoop.fs.s3a.secret.key", context.get_secret('s3_secret_key'))
            .set("spark.hadoop.fs.s3a.endpoint", context.get_param("s3_endpoint"))
            .set("spark.hadoop.fs.s3a.region", context.get_param("s3_region"))
            .set("spark.hadoop.fs.s3a.impl", "org.apache.hadoop.fs.s3a.S3AFileSystem")
            .set("com.amazonaws.services.s3.enableV4", True)
            .set("spark.driver.extraJavaOptions", "-Dcom.amazonaws.services.s3.
↪enableV4=true"))
    spark = (
        SparkSession.builder.config(conf=conf).appName("S3 app").getOrCreate()
    )

    ingest(mlrun_context=context, spark_context=spark)

# mlrun: end-code
```

Ingestion invocation:

```
from mlrun.datastore.sources import CSVSource
from mlrun.datastore.targets import ParquetTarget
from mlrun import code_to_function
import mlrun.feature_store as fstore

feature_set = fstore.FeatureSet("stock-quotes", entities=[fstore.Entity("ticker")],␣
↪engine="spark")

source = CSVSource("mycsv", path="v3io:///projects/quotes.csv")

spark_service_name = "spark" # As configured & shown in the Iguazio dashboard

fn = code_to_function(kind='remote-spark',  name='func')

run_config = fstore.RunConfig(local=False, function=fn, handler="ingest_handler")
run_config.with_secret('kubernetes', ['s3_access_key', 's3_secret_key'])
run_config.parameters = {
    "s3_endpoint" : "s3.us-east-2.amazonaws.com",
    "s3_region" : "us-east-2"
```

(continues on next page)

(continued from previous page)

```
}

target = ParquetTarget(
    path = "s3://my-s3-bucket/some/path",
    partitioned = False,
)

fstore.ingest(feature_set, source, targets=[target], run_config=run_config, spark_
↪context=spark_service_name)
```

12.4.6 Spark ingestion from Snowflake example

Spark ingestion from Snowflake can be executed both remotely and locally.

When running aggregations, they actually run on Spark and require Spark compute resources. The queries from the database are "regular" snowflake queries and they use Snowflake compute resources.

Note

`Entity` is case sensitive.

The following code executes local data ingestion from Snowflake.

```
from pyspark.sql import SparkSession

import mlrun
import mlrun.feature_store as fstore
from mlrun.datastore.sources import SnowflakeSource

spark = SparkSession.builder.appName("snowy").getOrCreate()

mlrun.get_or_create_project("feature_store")
feature_set = fstore.FeatureSet(
    name="customer", entities=[fstore.Entity("C_CUSTKEY")], engine="spark"
)

source = SnowflakeSource(
    "customer_sf",
    query="select * from customer limit 100000",
    url="<url>",
    user="<user>",
    password="<password>",
    database="SNOWFLAKE_SAMPLE_DATA",
    schema="TPCH_SF1",
    warehouse="compute_wh",
)

fstore.ingest(feature_set, source, spark_context=spark)
```

12.4.7 Spark ingestion from Azure example

Spark ingestion from Azure can be executed both remotely and locally. The following code executes remote data ingestion from Azure.

```
import mlrun

# Initialize the MLRun project object
project_name = "spark-azure-test"
project = mlrun.get_or_create_project(project_name, context="./")

from mlrun.runtimes import RemoteSparkRuntime
RemoteSparkRuntime.deploy_default_image()

from mlrun.datastore.sources import CSVSource
from mlrun.datastore.targets import ParquetTarget
from mlrun import code_to_function
import mlrun.feature_store as fstore

feature_set = fstore.FeatureSet("rides7", entities=[fstore.Entity("ride_id")], engine=
↪"spark", timestamp_key="key")

source = CSVSource("rides", path="wasbs://warroom@mlrunwarroom.blob.core.windows.net/ny_
↪taxi_train_subset_ride_id.csv")

spark_service_name = "spark-fs" # As configured & shown in the Iguazio dashboard

fn = code_to_function(kind='remote-spark',  name='func')

run_config = fstore.RunConfig(local=False, function=fn, handler="ingest_handler")

target = ParquetTarget(partitioned = True, time_partitioning_granularity="month")

feature_set.set_targets(targets=[target],with_defaults=False)

fstore.ingest(feature_set, source, run_config=run_config, spark_context=spark_service_
↪name)
```

CHAPTER

THIRTEEN

DEVELOP AND TRAIN MODELS

In this section

13.1 Model training and tracking

In this section

13.1.1 Create a basic training job

In this section, you create a simple job to train a model and log metrics, logs, and plots using MLRun's auto-logging:

- *Define the training code*
- *Create the job*
- *Run the job*
- *View job results*

Define the training code

The code you run is as follows. Notice, there is only a single line from MLRun to add all the MLOps capabilities:

```
%%writefile trainer.py

from sklearn import ensemble
from sklearn.model_selection import train_test_split

import mlrun
from mlrun.frameworks.sklearn import apply_mlrun


def train(
    dataset: mlrun.DataItem,  # data inputs are of type DataItem (abstract the data␣
↪source)
    label_column: str = "label",
    n_estimators: int = 100,
    learning_rate: float = 0.1,
    max_depth: int = 3,
    model_name: str = "cancer_classifier",
```

(continues on next page)

(continued from previous page)

```
):
    # Get the input dataframe (Use DataItem.as_df() to access any data source)
    df = dataset.as_df()

    # Initialize the x & y data
    X = df.drop(label_column, axis=1)
    y = df[label_column]

    # Train/Test split the dataset
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    # Pick an ideal ML model
    model = ensemble.GradientBoostingClassifier(
        n_estimators=n_estimators, learning_rate=learning_rate, max_depth=max_depth
    )

    # ------------------- The only line you need to add for MLOps ----------------------
    # Wraps the model with MLOps (test set is provided for analysis & accuracy measurements)
    apply_mlrun(model=model, model_name=model_name, x_test=X_test, y_test=y_test)
    # ----------------------------------------------------------------------------------

    # Train the model
    model.fit(X_train, y_train)
```

```
Writing trainer.py
```

Create the job

Next, use `code_to_function` to package up the `Job` to get ready to execute on the cluster:

```
import mlrun

training_job = mlrun.code_to_function(
    name="basic-training",
    filename="trainer.py",
    kind="job",
    image="mlrun/mlrun",
    handler="train"
)
```

Run the job

Finally, run the job. The dataset is from S3, but usually it is the output from a previous step in a pipeline.

```
run = training_job.run(
    inputs={"dataset": "https://igz-demo-datasets.s3.us-east-2.amazonaws.com/cancer-
↪dataset.csv"},
    params = {"n_estimators": 100, "learning_rate": 1e-1, "max_depth": 3}
)
```

```
> 2022-07-22 22:27:15,162 [info] starting run basic-training-train␣
↪uid=bc1c6ad491c340e1a3b9b91bb520454f DB=http://mlrun-api:8080
> 2022-07-22 22:27:15,349 [info] Job is running in the background, pod: basic-training-
↪train-kkntj
> 2022-07-22 22:27:20,927 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2022-07-22 22:27:21,640 [info] run executed, status=completed
```

View job results

Once the job is complete, you can view the output metrics and visualize the artifacts.

```
run.outputs
```

```
{'accuracy': 0.956140350877193,
 'f1_score': 0.965034965034965,
 'precision_score': 0.9583333333333334,
 'recall_score': 0.971830985915493,
 'feature-importance': 'v3io:///projects/default/artifacts/feature-importance.html',
 'test_set': 'store://artifacts/default/basic-training-train_test_
↪set:bc1c6ad491c340e1a3b9b91bb520454f',
 'confusion-matrix': 'v3io:///projects/default/artifacts/confusion-matrix.html',
 'roc-curves': 'v3io:///projects/default/artifacts/roc-curves.html',
 'calibration-curve': 'v3io:///projects/default/artifacts/calibration-curve.html',
 'model': 'store://artifacts/default/cancer_classifier:bc1c6ad491c340e1a3b9b91bb520454f'}
```

```
run.artifact("confusion-matrix").show()
```

```
<IPython.core.display.HTML object>
```

```
run.artifact("feature-importance").show()
```

```
<IPython.core.display.HTML object>
```

```
run.artifact("test_set").show()
```

```
     mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0          12.47         18.60           81.09      481.9          0.09965
1          18.94         21.31          123.60     1130.0          0.09009
2          15.46         19.48          101.70      748.9          0.10920
3          12.40         17.68           81.47      467.8          0.10540
4          11.54         14.44           74.65      402.9          0.09984
..           ...           ...             ...        ...              ...
109        14.64         16.85           94.21      666.0          0.08641
110        16.07         19.65          104.10      817.7          0.09168
111        11.52         14.93           73.87      406.3          0.10130
112        14.22         27.85           92.55      623.9          0.08223
113        20.73         31.12          135.70     1419.0          0.09469

     mean compactness  mean concavity  mean concave points  mean symmetry  \
0             0.10580         0.08005              0.03821         0.1925
1             0.10290         0.10800              0.07951         0.1582
2             0.12230         0.14660              0.08087         0.1931
3             0.13160         0.07741              0.02799         0.1811
4             0.11200         0.06737              0.02594         0.1818
..                ...             ...                  ...            ...
109           0.06698         0.05192              0.02791         0.1409
110           0.08424         0.09769              0.06638         0.1798
111           0.07808         0.04328              0.02929         0.1883
112           0.10390         0.11030              0.04408         0.1342
113           0.11430         0.13670              0.08646         0.1769

     mean fractal dimension  ...  worst texture  worst perimeter  worst area  \
0                   0.06373  ...          24.64            96.05       677.9
1                   0.05461  ...          26.58           165.90      1866.0
2                   0.05796  ...          26.00           124.90      1156.0
3                   0.07102  ...          22.91            89.61       515.8
4                   0.06782  ...          19.68            78.78       457.8
..                      ...  ...            ...              ...         ...
109                 0.05355  ...          25.44           106.00       831.0
110                 0.05391  ...          24.56           128.80      1223.0
111                 0.06168  ...          21.19            80.88       491.8
112                 0.06129  ...          40.54           102.50       764.0
113                 0.05674  ...          47.16           214.00      3432.0

     worst smoothness  worst compactness  worst concavity  \
0              0.1426             0.2378           0.2671
1              0.1193             0.2336           0.2687
2              0.1546             0.2394           0.3791
3              0.1450             0.2629           0.2403
4              0.1345             0.2118           0.1797
..                ...                ...              ...
109            0.1142             0.2070           0.2437
110            0.1500             0.2045           0.2829
```

(continues on next page)

(continued from previous page)

```
111            0.1389            0.1582           0.1804
112            0.1081            0.2426           0.3064
113            0.1401            0.2644           0.3442

     worst concave points  worst symmetry  worst fractal dimension  label
0                 0.10150          0.3014                  0.08750      1
1                 0.17890          0.2551                  0.06589      0
2                 0.15140          0.2837                  0.08019      0
3                 0.07370          0.2556                  0.09359      1
4                 0.06918          0.2329                  0.08134      1
..                    ...             ...                      ...    ...
109               0.07828          0.2455                  0.06596      1
110               0.15200          0.2650                  0.06387      0
111               0.09608          0.2664                  0.07809      1
112               0.08219          0.1890                  0.07796      1
113               0.16590          0.2868                  0.08218      0

[114 rows x 31 columns]
```

13.1.2 Working with data and model artifacts

When running a training job, you need to pass in the data used for training, and save the resulting model. Both the data and model can be considered *artifacts* in MLRun. In the context of an ML pipeline, the data is an `input` and the model is an `output`.

Consider the following snippet from a pipeline in the Build and run automated ML pipelines and CI/CD section of the docs:

```
# Ingest data
...

# Train a model using the auto_trainer hub function
train = mlrun.run_function(
    "hub://auto_trainer",
    inputs={"dataset": ingest.outputs["dataset"]},
    params = {
        "model_class": "sklearn.ensemble.RandomForestClassifier",
        "train_test_split_size": 0.2,
        "label_columns": "label",
        "model_name": 'cancer',
    },
    handler='train',
    outputs=["model"],
)

### Deploy model
...
```

This snippet trains a model using the data provided into `inputs` and passes the model to the rest of the pipeline using the `outputs`.

Input data

The `inputs` parameter is a dictionary of key-value mappings. In this case, the input is the `dataset` (which is actually an output from a previous step). Within the training job, you can access the `dataset` input as an MLRun *Data items* (essentially a smart data pointer that provides convenience methods).

For example, this Python training function is expecting a parameter called `dataset` that is of type `DataItem`. Within the function, you can get the training set as a Pandas dataframe via the following:

```
import mlrun

def train(context: mlrun.MLClientCtx, dataset: mlrun.DataItem, ...):
    df = dataset.as_df()
```

Notice how this maps to the parameter `datasets` that you passed into your `inputs`.

Output model

The `outputs` parameter is a list of artifacts that were logged during the job. In this case, it is your newly trained `model`, however it could also be a dataset or plot. These artifacts are logged using the experiment tracking hooks via the MLRun execution context.

One way to log models is via MLRun auto-logging with *apply_mlrun*. This saves the model, test sets, visualizations, and more as outputs. Additionally, you can use manual hooks to save datasets and models. For example, this Python training function uses both auto logging and manual logging:

```
import mlrun
from mlrun.frameworks.sklearn import apply_mlrun
from sklearn import ensemble
import cloudpickle

def train(context: mlrun.MLClientCtx, dataset: mlrun.DataItem, ...):
    # Prep data using df
    df = dataset.as_df()
    X_train, X_test, y_train, y_test = ...

    # Apply auto logging
    model = ensemble.GradientBoostingClassifier(...)
    apply_mlrun(model=model, model_name=model_name, x_test=X_test, y_test=y_test)

    # Train
    model.fit(X_train, y_train)

    # Manual logging
    context.log_dataset(key="X_test_dataset", df=X_test)
    context.log_model(key="my_model", body=cloudpickle.dumps(model), model_file="model.
↪pkl")
```

Once your artifact is logged, it can be accessed throughout the rest of the pipeline. For example, for the pipeline snippet from the Build and run automated ML pipelines and CI/CD section of the docs, you can access your model like the following:

```
# Train a model using the auto_trainer hub function
train = mlrun.run_function(
```

(continues on next page)

(continued from previous page)

```
    "hub://auto_trainer",
    inputs={"dataset": ingest.outputs["dataset"]},
    ...
    outputs=["model"],
)

# Get trained model
model = train.outputs["model"]
```

Notice how this maps to the parameter `model` that you passed into your `outputs`.

Model artifacts

By storing multiple models, you can experiment with them,
and compare their performance, without having to worry about losing the previous results.

The simplest way to store a model named `my_model` is with the following code:

```
from pickle import dumps
model_data = dumps(model)
context.log_model(key='my_model', body=model_data, model_file='my_model.pkl')
```

You can also store any related metrics by providing a dictionary in the `metrics` parameter, such as `metrics={'accuracy': 0.9}`. Furthermore, any additional data that you would like to store along with the model can be specified in the `extra_data` parameter. For example `extra_data={'confusion': confusion.target_path}`

A convenient utility method, `eval_model_v2`, which calculates mode metrics is available in `mlrun.utils`.

See example below for a simple model trained using scikit-learn (normally, you would send the data as input to the function). The last two lines evaluate the model and log the model.

```
from sklearn import linear_model
from sklearn import datasets
from sklearn.model_selection import train_test_split
from pickle import dumps

from mlrun.execution import MLClientCtx
from mlrun.mlutils import eval_model_v2

def train_iris(context: MLClientCtx):

    # Basic scikit-learn iris SVM model
    X, y = datasets.load_iris(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42)
    model = linear_model.LogisticRegression(max_iter=10000)
    model.fit(X_train, y_train)

    # Evaluate model results and get the evaluation metrics
    eval_metrics = eval_model_v2(context, X_test, y_test, model)

    # Log model
```

(continues on next page)

(continued from previous page)

```
    context.log_model("model",
                      body=dumps(model),
                      artifact_path=context.artifact_subpath("models"),
                      extra_data=eval_metrics,
                      model_file="model.pkl",
                      metrics=context.results,
                      labels={"class": "sklearn.linear_model.LogisticRegression"})
```

Save the code above to `train_iris.py`. The following code loads the function and runs it as a job. See the Quick start tutorial to learn how to create the project and set the artifact path.

```
from mlrun import code_to_function

gen_func = code_to_function(name='train_iris',
                            filename='train_iris.py',
                            handler='train_iris',
                            kind='job',
                            image='mlrun/ml-models')

train_iris_func = project.set_function(gen_func).apply(auto_mount())

train_iris = train_iris_func.run(name='train_iris',
                                 handler='train_iris',
                                 artifact_path=artifact_path)
```

You can now use `get_model` to read the model and run it. This function gets the model file, metadata, and extra data. The input can be either the path of the model, or the directory where the model resides. If you provide a directory, the function searches for the model file (by default it searches for .pkl files)

The following example gets the model from `models_path` and test data in `test_set` with the expected label provided as a column of the test data. The name of the column containing the expected label is provided in `label_column`. The example then retrieves the models, runs the model with the test data and updates the model with the metrics and results of the test data.

```
from pickle import load

from mlrun.execution import MLClientCtx
from mlrun.datastore import DataItem
from mlrun.artifacts import get_model, update_model
from mlrun.mlutils import eval_model_v2

def test_model(context: MLClientCtx,
               models_path: DataItem,
               test_set: DataItem,
               label_column: str):

    if models_path is None:
        models_path = context.artifact_subpath("models")
    xtest = test_set.as_df()
    ytest = xtest.pop(label_column)

    model_file, model_obj, _ = get_model(models_path)
    model = load(open(model_file, 'rb'))
```

(continues on next page)

(continued from previous page)

```
    extra_data = eval_model_v2(context, xtest, ytest.values, model)
    update_model(model_artifact=model_obj, extra_data=extra_data,
                 metrics=context.results, key_prefix='validation-')
```

To run the code, place the code above in `test_model.py` and use the following snippet. The model from the previous step is provided as the `models_path`:

```
from mlrun.platforms import auto_mount
gen_func = code_to_function(name='test_model',
                            filename='test_model.py',
                            handler='test_model',
                            kind='job',
                            image='mlrun/ml-models')

func = project.set_function(gen_func).apply(auto_mount())

run = func.run(name='test_model',
               handler='test_model',
               params={'label_column': 'label'},
               inputs={'models_path': train_iris.outputs['model'],
                       'test_set': 'https://s3.wasabisys.com/iguazio/data/iris/iris_
↪dataset.csv'}),
               artifact_path=artifact_path)
```

Plot artifacts

Storing plots is useful to visualize the data and to show any information regarding the model performance. For example, you can store scatter plots, histograms and cross-correlation of the data, and for the model store the ROC curve and confusion matrix.

The following code creates a confusion matrix plot using sklearn.metrics.plot_confusion_matrix and stores the plot in the artifact repository:

```
from mlrun.artifacts import PlotArtifact
from mlrun.mlutils import gcf_clear

gcf_clear(plt)
confusion_matrix = metrics.plot_confusion_matrix(model,
                                                 xtest,
                                                 ytest,
                                                 normalize='all',
                                                 values_format = '.2g',
                                                 cmap=plt.cm.Blues)
confusion_matrix = context.log_artifact(PlotArtifact('confusion-matrix', body=confusion_
↪matrix.figure_),
                                        local_path='plots/confusion_matrix.html')
```

You can use the `update_dataset_meta` function to associate the plot with the dataset by assigning the value of the `extra_data` parameter:

```
from mlrun.artifacts import update_dataset_meta

extra_data = {'confusion_matrix': confusion_matrix}
update_dataset_meta(dataset, extra_data=extra_data)
```

13.1.3 Automated experiment tracking

You can write custom training functions or use built-in hub functions for training models using common open-source frameworks and/or cloud services (such as AzureML, Sagemaker, etc.).

Inside the ML function you can use the `apply_mlrun()` method, which automates the tracking and MLOps functionality.

With `apply_mlrun()` the following outputs are generated automatically:

- Plots — loss convergence, ROC, confusion matrix, feature importance, etc.
- Metrics — accuracy, loss, etc.
- Dataset artifacts — like the dataset used for training and / or testing
- Custom code — like custom layers, metrics, and so on
- Model artifacts — enables versioning, monitoring and automated deployment

In addition it handles automation of various MLOps tasks like scaling runs over multiple containers (with Dask, Horovod, and Spark), run profiling, hyperparameter tuning, ML Pipeline, and CI/CD integration, etc.

`apply_mlrun()` accepts the model object and various optional parameters. For example:

```
apply_mlrun(model=model, model_name="my_model",
            x_test=x_test, y_test=y_test)
```

When specifying the `x_test` and `y_test` data it generates various plots and calculations to evaluate the model. Metadata and parameters are automatically recorded (from the MLRun `context` object) and don't need to be specified.

`apply_mlrun` is framework specific and can be imported from MLRun's **frameworks** package — a collection of commonly used machine and deep learning frameworks fully supported by MLRun.

`apply_mlrun` can be used with its default settings, but it is highly flexible and rich with different options and configurations. Reading the docs of your favorite framework to get the most out of MLRun:

- SciKit-Learn
- TensorFlow (and Keras)
- PyTorch
- XGBoost
- LightGBM
- ONNX

13.1.4 Using the built-in training function

The MLRun Function Hub includes, among other things, training functions. The most commonly used function for training is `auto_trainer`, which includes the following handlers:

- *Train*
- *Evaluate*

Train

The main and default handler of any training function is called `"train"`. In the Auto Trainer this handler performs an ML training function using SciKit-Learn's API, meaning the function follows the structure below:

1. **Get the data**: Get the dataset passed to a local path.
2. **Split the data into datasets**: Split the given data into a training set and a testing set.
3. **Get the model**: Initialize a model instance out of a given class or load a provided model The supported classes are anything based on `sklearn.Estimator`, `xgboost.XGBModel`, `lightgbm.LGBMModel`, including custom code as well.
4. **Train**: Call the model's `fit` method to train it on the training set.
5. **Test**: Test the model on the testing set.
6. **Log**: Calculate the metrics and produce the artifacts to log the results and plots.

MLRun orchestrates all of the above steps. The training is done with the shortcut function `apply_mlrun` that enables the automatic logging and additional features.

To start, run `import mlrun` and create a project:

```
import mlrun
# Set the base project name
project_name_base = 'training-test'

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name_base, context="./", user_project=True)
```

Next, import the Auto Trainer from the Function Hub using MLRun's `import_function` function:

```
auto_trainer = project.set_function(mlrun.import_function("hub://auto_trainer"))
```

The following example trains a Random Forest model:

```
dataset_url = "https://s3.wasabisys.com/iguazio/data/function-marketplace-data/xgb_
↪trainer/classifier-data.csv"

train_run = auto_trainer.run(
    handler="train",
    inputs={"dataset": dataset_url},
    params={
        # Model parameters:
        "model_class": "sklearn.ensemble.RandomForestClassifier",
        "model_kwargs": {"max_depth": 8},  # Could be also passed as "MODEL_max_depth": 8
        "model_name": "MyModel",
        # Dataset parameters:
```

(continues on next page)

(continued from previous page)

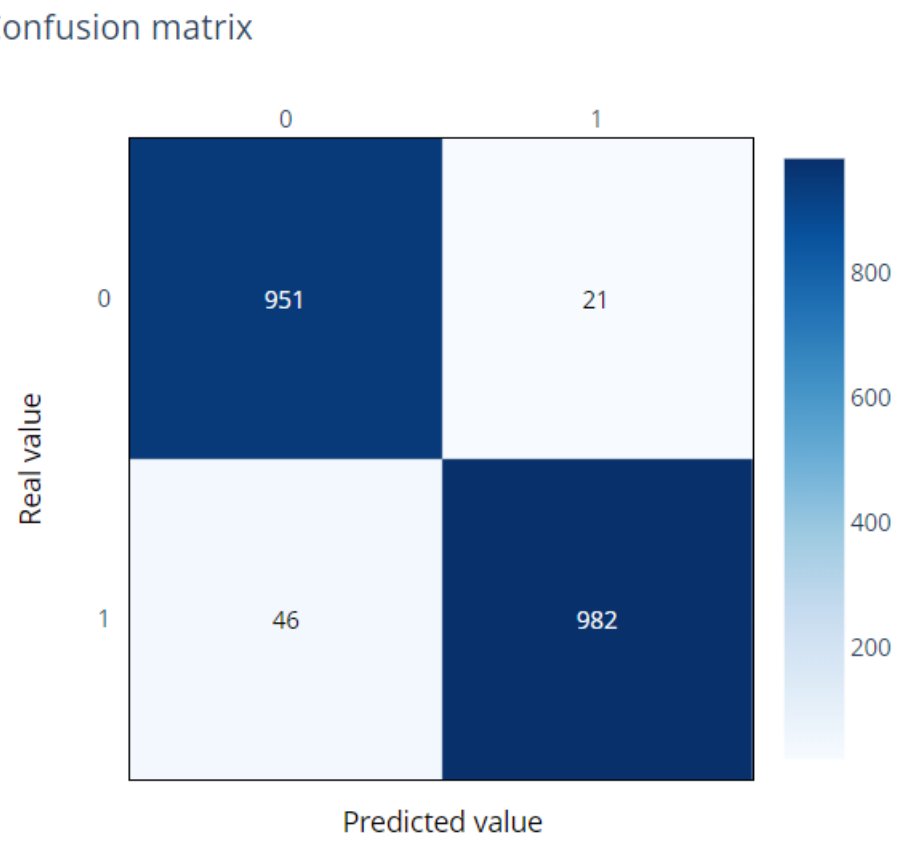
```
        "drop_columns": ["feat_0", "feat_2"],
        "train_test_split_size": 0.2,
        "random_state": 42,
        "label_columns": "labels",
    }
)
```

Outputs

`train_run.outputs` returns all the outputs. The outputs are:

- **Trained model**: The trained model is logged as a `ModelArtifact` with all the following artifacts registered to it.
- **Test dataset**: The test set used to test the model post training is logged as a `DatasetArtifact`.
- **Plots**: Informative plots regarding the model like confusion matrix and features importance are drawn and logged as `PlotArtifacts`.
- **Results**: List of all the calculations of metrics tested on the testing set.

For instance, calling `train_run.artifact('confusion-matrix').show()` shows the following confusion matrix:



Parameters

To view the parameters of `train`, expand the section below:

train handler parameters:

Model Parameters

Parameters to initialize a new model object or load a logged one for retraining.

- `model_class`: `str` — The class of the model to initialize. Can be a module path like `"sklearn.linear_model.LogisticRegression"` or a custom model passed through the custom objects parameters below. Only one of `model_class` and `model_path` can be given.
- `model_path`: `str` — A `ModelArtifact` URI to load and retrain. Only one of `model_class` and `model_path` can be given.
- `model_kwargs`: `dict` — Additional parameters to pass onto the initialization of the model object (the model's class `__init__` method).

Data parameters

Parameters to get a dataset and prepare it for training, splitting into training and testing if required.

- `dataset`: `Union[str, list, dict]` — The dataset to train the model on.
  - Can be passed as part of `inputs` to be parsed as `mlrun.DataItem`, meaning it supports either a URI or a FeatureVector.
  - Can be passed as part of `params`, meaning it can be a `list` or a `dict`.
- `drop_columns`: `Union[str, int, List[str], List[int]]` — Columns to drop from the dataset. Can be passed as strings representing the column names or integers representing the column numbers.
- `test_set`: `Union[str, list, dict]` — The test set to test the model with post training. Notice only one of `test_set` or `train_test_split_size` is expected.
  - Can be passed as part of `inputs` to be parsed as `mlrun.DataItem`, meaning it supports either a URI or a FeatureVector.
  - Can be passed as part of `params`, meaning it can be a `list` or a `dict`.
- `train_test_split_size`: `float` = `0.2` — The proportion of the dataset to include in the test split. The size of the Training set is set to the complement of this value. Must be between 0.0 and 1.0. Defaults to 0.2
- `label_columns`: `Union[str, int, List[str], List[int]]` — The target label(s) of the column(s) in the dataset. Can be passed as strings representing the column names or integers representing the column numbers.
- `random_state`: `int` - Random state (seed) for `train_test_split`.

Train parameters

Parameters to pass to the `fit` method of the model object.

- `train_kwargs`: `dict` — Additional parameters to pass onto the `fit` method.

Logging parameters

Parameters to control the automatic logging feature of MLRun. You can adjust the logging outputs as relevant and if not passed, a default list of artifacts and metrics is produced and calculated.

- `model_name`: `str` = "`model`" — The model's name to use for storing the model artifact, defaults to 'model'.
- `tag`: `str` — The model's tag to log with.

- sample_set: Union[str, list, dict] — A sample set of inputs for the model for logging its stats alongside the model in favor of model monitoring. If not given, the training set is used instead.
  - Can be passed as part of inputs to be parsed as mlrun.DataItem, meaning it supports either a URI or a FeatureVector.
  - Can be passed as part of params, meaning it can be a list or a dict.
- _artifacts: Dict[str, Union[list, dict]] — Additional artifacts to produce post training. See the ArtifactsLibrary of the desired framework to see the available list of artifacts.
- _metrics: Union[List[str], Dict[str, Union[list, dict]]] — Additional metrics to calculate post training. See how to pass metrics and custom metrics in the MetricsLibrary of the desired framework.
- apply_mlrun_kwargs: dict — Framework specific apply_mlrun key word arguments. Refer to the framework of choice to know more (SciKit-Learn, XGBoost or LightGBM)

Custom objects parameters

Parameters to include custom objects like custom model class, metric code and artifact plan. Keep in mind that the model artifact created is logged with the custom objects, so if model_path is used, the custom objects used to train it are not required for loading it, it happens automatically.

- custom_objects_map: Union[str, Dict[str, Union[str, List[str]]]] — A map of all the custom objects required for loading, training and testing the model. Can be passed as a dictionary or a json file path. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. For example:

```
{
    "/.../custom_model.py": "MyModel",
    "/.../custom_objects.py": ["object1", "object2"]
}
```

  All the paths are accessed from the given 'custom_objects_directory', meaning each py file is read from 'custom_objects_directory/'. If the model path given is of a store object, the custom objects map is read from the logged custom object map artifact of the model.

  Note

  The custom objects are imported in the order they came in this dictionary (or json). If a custom object is dependant on another, make sure to put it below the one it relies on.

- custom_objects_directory: Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (and are extracted during the start of the run).

  Note

  The parameters for additional arguments model_kwargs, train_kwargs and apply_mlrun_kwargs can be also passed in the global kwargs with the matching prefixes: "MODEL_", "TRAIN_", "MLRUN_".

Evaluate

The `"evaluate"` handler is used to test the model on a given testing set and log its results. This is a common phase in every model lifecycle and should be done periodically on updated testing sets to confirm that your model is still relevant. The function uses SciKit-Learn's API for evaluation, meaning the function follows the structure below:

1. **Get the data**: Get the testing dataset passed to a local path.
2. **Get the model**: Get the model object out of the `ModelArtifact` URI.
3. **Predict**: Call the model's `predict` (and `predict_proba` if needed) method to test it on the testing set.
4. **Log**: Test the model on the testing set and log the results and artifacts.

MLRun orchestrates all of the above steps. The evaluation is done with the shortcut function `apply_mlrun` that enables the automatic logging and further features.

To evaluate the test-set, use the following command:

```
evaluate_run = auto_trainer.run(
    handler="evaluate",
    inputs={"dataset": train_run.outputs['test_set']},
    params={
        "model": train_run.outputs['model'],
        "label_columns": "labels",
    },
)
```

Outputs

`evaluate_run.outputs` returns all the outputs. The outputs are:

- **Evaluated model**: The evaluated model's `ModelArtifact` is updated with all the following artifacts registered to it.
- **Test dataset**: The test set used to test the model post-training is logged as a `DatasetArtifact`.
- **Plots**: Informative plots regarding the model like confusion matrix and features importance are drawn and logged as `PlotArtifacts`.
- **Results**: List of all the calculations of metrics tested on the testing set.

Parameters

To view the parameters of `evaluate`, expand the section below:

evaluate handler parameters:

Model Parameters

Parameters to load a logged model.

- `model_path`: `str` — A `ModelArtifact` URI to load.

Data parameters

Parameters to get a dataset and prepare it for training, splitting into training and testing if required.

- `dataset`: `Union[str, list, dict]` — The dataset to train the model on.
  - Can be passed as part of `inputs` to be parsed as `mlrun.DataItem`, meaning it supports either a URI or a FeatureVector.
  - Can be passed as part of `params`, meaning it can be a `list` or a `dict`.
- `drop_columns`: `Union[str, int, List[str], List[int]]` — columns to drop from the dataset. Can be passed as strings representing the column names or integers representing the column numbers.
- `label_columns`: `Union[str, int, List[str], List[int]]` — The target label(s) of the column(s) in the dataset. Can be passed as strings representing the column names or integers representing the column numbers.

Predict parameters

Parameters to pass to the `predict` method of the model object.

- `predict_kwargs`: `dict` — Additional parameters to pass onto the `predict` method.

Logging parameters

Parameters to control the automatic logging feature of MLRun. You can adjust the logging outputs as relervant, and if not passed, a default list of artifacts and metrics is produced and calculated.

- `_artifacts`: `Dict[str, Union[list, dict]]` — Additional artifacts to produce post training. See the `ArtifactsLibrary` of the desired framework to see the available list of artifacts.
- `_metrics`: `Union[List[str], Dict[str, Union[list, dict]]]` — Additional metrics to calculate post training. See how to pass metrics and custom metrics in the `MetricsLibrary` of the desired framework.
- `apply_mlrun_kwargs`: `dict` — Framework specific `apply_mlrun` key word arguments. Refer to the framework of choice to know more (SciKit-Learn, XGBoost or LightGBM).

Custom objects parameters

Parameters to include custom objects for the evaluation like custom metric code and artifact plans. Keep in mind that the custom objects used to train the model are not required for loading it, it happens automatically.

- `custom_objects_map`: `Union[str, Dict[str, Union[str, List[str]]]]` — A map of all the custom objects required for loading, training and testing the model. Can be passed as a dictionary or a json file path. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. For example:

```
{
    "/.../custom_metric.py": "MyMetric",
    "/.../custom_plans.py": ["plan1", "plan2"]
}
```

  All the paths are accessed from the given 'custom_objects_directory', meaning each py file is read from the 'custom_objects_directory/'. If the model path given is of a store object, the custom objects map is read from the logged custom object map artifact of the model.

  Note

  The custom objects are imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- `custom_objects_directory` — Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (iti is extracted during the start of the run).

  Note

The parameters for additional arguments `predict_kwargs` and `apply_mlrun_kwargs` can be also passed in the global `kwargs` with the matching prefixes: `"PREDICT_"`, `"MLRUN_"`.

13.1.5 Hyperparameter tuning optimization

MLRun supports iterative tasks for automatic and distributed execution of many tasks with variable parameters (hyperparams). Iterative tasks can be distributed across multiple containers. They can be used for:

- Parallel loading and preparation of many data objects
- Model training with different parameter sets and/or algorithms
- Parallel testing with many test vector options
- AutoML

MLRun iterations can be viewed as child runs under the main task/run. Each child run gets a set of parameters that are computed/selected from the input hyperparameters based on the chosen strategy (*Grid*, *List*, *Random* or *Custom*).

The different iterations can run in parallel over multiple containers (using Dask or Nuclio runtimes, which manage the workers). Read more in *Parallel execution over containers*.

The hyperparameters and options are specified in the `task` or the *`run()`* command through the `hyperparams` (for hyperparam values) and `hyper_param_options` (for *`HyperParamOptions`*) properties. See the examples below. Hyperparameters can also be loaded directly from a CSV or Json file (by setting the `param_file` hyper option).

The hyperparams are specified as a struct of `key: list` values for example: `{"p1": [1,2,3], "p2": [10, 20]}`. The values can be of any type (int, string, float, ...). The lists are used to compute the parameter combinations using one of the following strategies:

- *Grid search* (`grid`) — running all the parameter combinations.
- *Random* (`random`) — running a sampled set from all the parameter combinations.
- *List* (`list`) — running the first parameter from each list followed by the seco2nd from each list and so on. **All the lists must be of equal size**.
- *Custom* (`custom`) — determine the parameter combination per run programmatically.

You can specify a selection criteria to select the best run among the different child runs by setting the `selector` option. This marks the selected result as the parent (iteration 0) result, and marks the best result in the user interface.

You can also specify the `stop_condition` to stop the execution of child runs when some criteria, based on the returned results, is met (for example `stop_condition="accuracy>=0.9"`).

In this section

- *Basic code*
- *Review the results*
- *Examples*
- *Parallel execution over containers*

Basic code

Here's a basic example of running multiple jobs in parallel for **hyperparameters tuning**, selecting the best run with respect to the `max accuracy`.

Run the hyperparameters tuning job by using the keywords arguments:

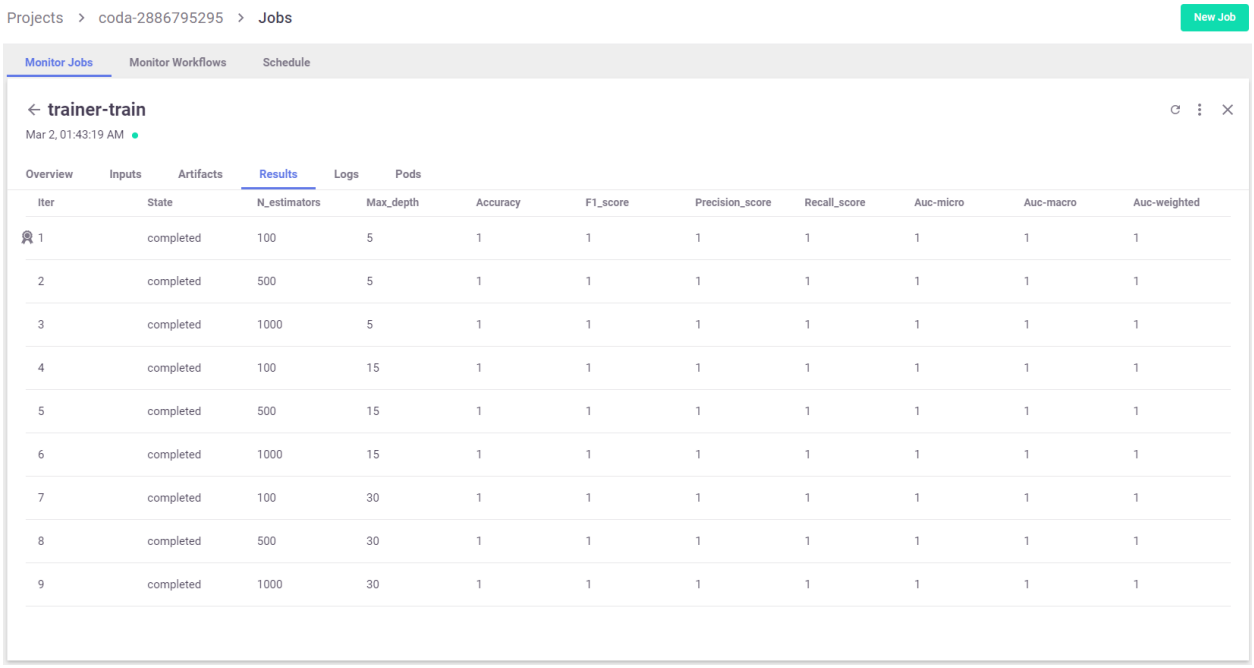
- `hyperparams` for the hyperparameters options and values of choice.
- `selector` for specifying how to select the best model.

```
hp_tuning_run = project.run_function(
    "trainer",
    inputs={"dataset": gen_data_run.outputs["dataset"]},
    hyperparams={
        "n_estimators": [100, 500, 1000],
        "max_depth": [5, 15, 30]
    },
    selector="max.accuracy",
    local=True
)
```

The returned run object in this case represents the `parent` (and the **best** result). You can also access the individual child runs (called iterations) in the MLRun UI.

Review the results

When running a hyperparam job, the job `results` tab shows the list and marks the best run:

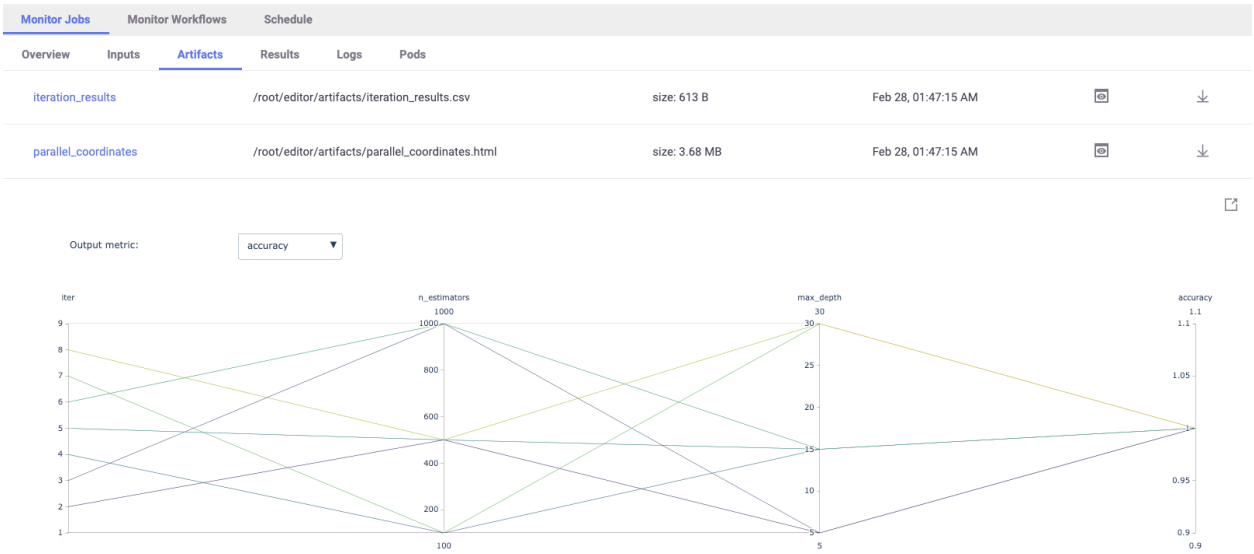


Iter	State	N_estimators	Max_depth	Accuracy	F1_score	Precision_score	Recall_score	Auc-micro	Auc-macro	Auc-weighted
1	completed	100	5	1	1	1	1	1	1	1
2	completed	500	5	1	1	1	1	1	1	1
3	completed	1000	5	1	1	1	1	1	1	1
4	completed	100	15	1	1	1	1	1	1	1
5	completed	500	15	1	1	1	1	1	1	1
6	completed	1000	15	1	1	1	1	1	1	1
7	completed	100	30	1	1	1	1	1	1	1
8	completed	500	30	1	1	1	1	1	1	1
9	completed	1000	30	1	1	1	1	1	1	1

You can also view results by printing the artifact `iteration_results`:

```
hp_tuning_run.artifact("iteration_results").as_df()
```

MLRun also generates a `parallel coordinates plot` for the run, you can view it in the MLRun UI.



Examples

Base dummy function:

```
import mlrun
```

```
> 2021-10-23 12:47:39,982 [warning] Failed resolving version info. Ignoring and using␣
↪defaults
> 2021-10-23 12:47:43,488 [warning] Unable to parse server or client version. Assuming␣
↪compatible: {'server_version': '0.8.0-rc7', 'client_version': 'unstable'}
```

```
def hyper_func(context, p1, p2):
    print(f"p1={p1}, p2={p2}, result={p1 * p2}")
    context.log_result("multiplier", p1 * p2)
```

Grid search (default)

```
grid_params = {"p1": [2,4,1], "p2": [10,20]}
task = mlrun.new_task("grid-demo").with_hyper_params(grid_params, selector="max.
↪multiplier")
run = mlrun.new_function().run(task, handler=hyper_func)
```

```
> 2021-10-23 12:47:43,505 [info] starting run grid-demo␣
↪uid=29c9083db6774e5096a97c9b6b6c8e93 DB=http://mlrun-api:8080
p1=2, p2=10, result=20
p1=4, p2=10, result=40
p1=1, p2=10, result=10
p1=2, p2=20, result=40
p1=4, p2=20, result=80
p1=1, p2=20, result=20
> 2021-10-23 12:47:44,851 [info] best iteration=5, used criteria max.multiplier
```

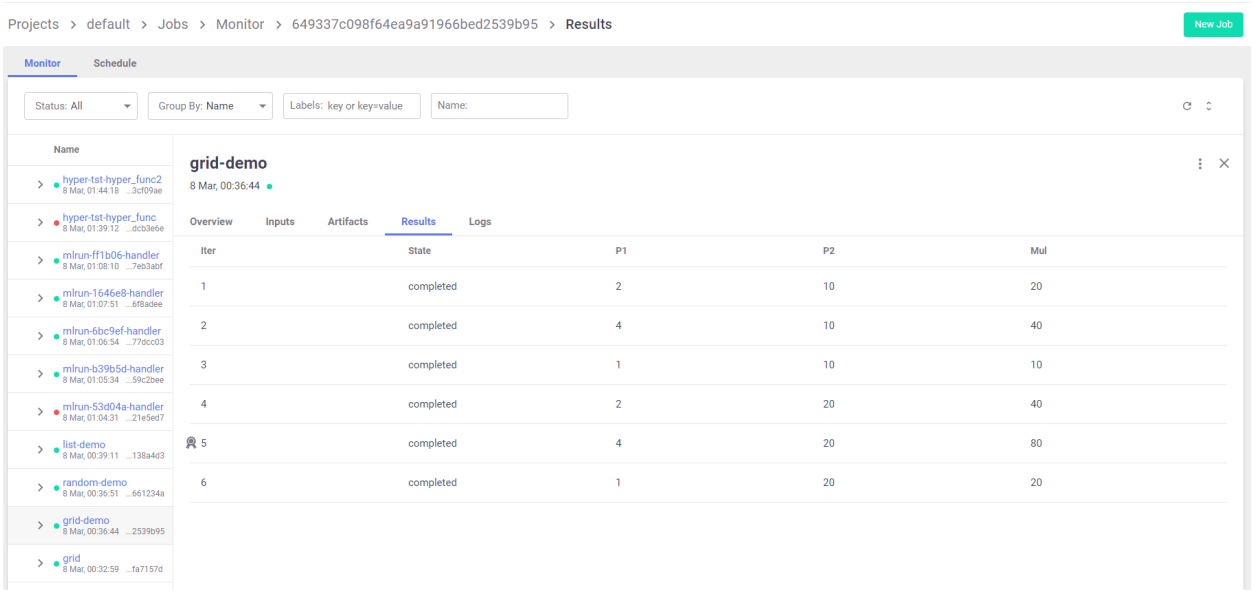
```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:47:45,071 [info] run executed, status=completed
```

UI Screenshot:



Random Search

MLRun chooses random parameter combinations. Limit the number of combinations using the `max_iterations` attribute.

```
grid_params = {"p1": [2,4,1,3], "p2": [10,20,30]}
task = mlrun.new_task("random-demo")
task.with_hyper_params(grid_params, selector="max.multiplier", strategy="random", max_
↪iterations=4)
run = mlrun.new_function().run(task, handler=hyper_func)
```

```
> 2021-10-23 12:47:45,077 [info] starting run random-demo␣
↪uid=cac368c7fc33455f97ca806e5c7abf2f DB=http://mlrun-api:8080
p1=2, p2=20, result=40
p1=4, p2=10, result=40
p1=3, p2=10, result=30
p1=3, p2=20, result=60
> 2021-10-23 12:47:45,966 [info] best iteration=4, used criteria max.multiplier
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:47:46,177 [info] run executed, status=completed
```

List search

This example also shows how to use the `stop_condition` option.

```
list_params = {"p1": [2,3,7,4,5], "p2": [15,10,10,20,30]}
task = mlrun.new_task("list-demo").with_hyper_params(
    list_params, selector="max.multiplier", strategy="list", stop_condition="multiplier>
↪=70")
run = mlrun.new_function().run(task, handler=hyper_func)
```

```
> 2021-10-23 12:47:46,184 [info] starting run list-demo␣
↪uid=136edfb9c9404a61933c73bbbd35b18b DB=http://mlrun-api:8080
p1=2, p2=15, result=30
p1=3, p2=10, result=30
p1=7, p2=10, result=70
> 2021-10-23 12:47:47,193 [info] reached early stop condition (multiplier>=70), stopping␣
↪iterations!
> 2021-10-23 12:47:47,195 [info] best iteration=3, used criteria max.multiplier
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:47:47,385 [info] run executed, status=completed
```

Custom iterator

You can define a child iteration context under the parent/main run. The child run is logged independently.

```
def handler(context: mlrun.MLClientCtx, param_list):
    best_multiplier = total = 0
    for param in param_list:
        with context.get_child_context(**param) as child:
            hyper_func(child, **child.parameters)
            multiplier = child.results['multiplier']
            total += multiplier
            if multiplier > best_multiplier:
                child.mark_as_best()
                best_multiplier = multiplier
```

(continues on next page)

(continued from previous page)

```
    # log result at the parent
    context.log_result('avg_multiplier', total / len(param_list))
```

```
param_list = [{"p1":2, "p2":10}, {"p1":3, "p2":30}, {"p1":4, "p2":7}]
run = mlrun.new_function().run(handler=handler, params={"param_list": param_list})
```

```
> 2021-10-23 12:47:47,403 [info] starting run mlrun-a79c5c-handler␣
↪uid=c3eb08ebae02464ca4025c77b12e3c39 DB=http://mlrun-api:8080
p1=2, p2=10, result=20
p1=3, p2=30, result=90
p1=4, p2=7, result=28
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:47:48,734 [info] run executed, status=completed
```

Parallel execution over containers

When working with compute intensive or long running tasks you'll want to run your iterations over a cluster of containers. At the same time, you don't want to bring up too many containers, and you want to limit the number of parallel tasks.

MLRun supports distribution of the child runs over Dask or Nuclio clusters. This is handled automatically by MLRun. You only need to deploy the Dask or Nuclio function used by the workers, and set the level of parallelism in the task. The execution can be controlled from the client/notebook, or can have a job (immediate or scheduled) that controls the execution.

Code example (single task)

```
# mark the start of a code section that will be sent to the job
# mlrun: start-code
```

```
import socket
import pandas as pd
def hyper_func2(context, data, p1, p2, p3):
    print(data.as_df().head())
    context.logger.info(f"p2={p2}, p3={p3}, r1={p2 * p3} at {socket.gethostname()}")
    context.log_result("r1", p2 * p3)
    raw_data = {
        "first_name": ["Jason", "Molly", "Tina", "Jake", "Amy"],
        "age": [42, 52, 36, 24, 73],
        "testScore": [25, 94, 57, 62, 70],
    }
    df = pd.DataFrame(raw_data, columns=["first_name", "age", "testScore"])
    context.log_dataset("mydf", df=df, stats=True)
```

```
# mlrun: end-code
```

Running the workers using Dask

This example creates a new function and executes the parent/controller as an MLRun `job` and the different child runs over a Dask cluster (MLRun Dask function).

Define a Dask cluster (using MLRun serverless Dask)

```
dask_cluster = mlrun.new_function("dask-cluster", kind='dask', image='mlrun/ml-models')
dask_cluster.apply(mlrun.mount_v3io())        # add volume mounts
dask_cluster.spec.service_type = "NodePort"   # open interface to the dask UI dashboard
dask_cluster.spec.replicas = 2                # define two containers
uri = dask_cluster.save()
uri
```

```
'db://default/dask-cluster'
```

```
# initialize the dask cluster and get its dashboard url
dask_cluster.client
```

```
> 2021-10-23 12:48:49,020 [info] trying dask client at: tcp://mlrun-dask-cluster-
↪eea516ff-5.default-tenant:8786
> 2021-10-23 12:48:49,049 [info] using remote dask scheduler (mlrun-dask-cluster-
↪eea516ff-5) at: tcp://mlrun-dask-cluster-eea516ff-5.default-tenant:8786
```

```
Mismatched versions found

+-------------+--------+-----------+---------+
| Package     | client | scheduler | workers |
+-------------+--------+-----------+---------+
| blosc       | 1.7.0  | 1.10.6    | None    |
| cloudpickle | 1.6.0  | 2.0.0     | None    |
| distributed | 2.30.0 | 2.30.1    | None    |
| lz4         | 3.1.0  | 3.1.3     | None    |
| msgpack     | 1.0.0  | 1.0.2     | None    |
| tornado     | 6.0.4  | 6.1       | None    |
+-------------+--------+-----------+---------+
Notes:
-  msgpack: Variation is ok, as long as everything is above 0.6
```

```
<IPython.core.display.HTML object>
```

```
<Client: 'tcp://10.200.0.72:8786' processes=0 threads=0, memory=0 B>
```

Define the parallel work

Set the `parallel_runs` attribute to indicate how many child tasks to run in parallel. Set the `dask_cluster_uri` to point to the dask cluster (if it's not set the cluster uri uses dask local). You can also set the `teardown_dask` flag to free up all the dask resources after completion.

```
grid_params = {"p2": [2,1,4,1], "p3": [10,20]}
task = mlrun.new_task(params={"p1": 8}, inputs={'data': 'https://s3.wasabisys.com/
↪iguazio/data/iris/iris_dataset.csv'})
task.with_hyper_params(
    grid_params, selector="r1", strategy="grid", parallel_runs=4, dask_cluster_uri=uri,␣
↪teardown_dask=True
)
```

```
<mlrun.model.RunTemplate at 0x7f673d7b1910>
```

Define a job that will take the code (using `code_to_function`) and run it over the cluster

```
fn = mlrun.code_to_function(name='hyper-tst', kind='job', image='mlrun/ml-models')
```

```
run = fn.run(task, handler=hyper_func2)
```

```
> 2021-10-23 12:49:56,388 [info] starting run hyper-tst-hyper_func2␣
↪uid=50eb72f5b0734954b8b1c57494f325bc DB=http://mlrun-api:8080
> 2021-10-23 12:49:56,565 [info] Job is running in the background, pod: hyper-tst-hyper-
↪func2-9g6z8
> 2021-10-23 12:49:59,813 [info] trying dask client at: tcp://mlrun-dask-cluster-
↪eea516ff-5.default-tenant:8786
> 2021-10-23 12:50:09,828 [warning] remote scheduler at tcp://mlrun-dask-cluster-
↪eea516ff-5.default-tenant:8786 not ready, will try to restart Timed out trying to␣
↪connect to tcp://mlrun-dask-cluster-eea516ff-5.default-tenant:8786 after 10 s
> 2021-10-23 12:50:15,733 [info] using remote dask scheduler (mlrun-dask-cluster-
↪04574796-5) at: tcp://mlrun-dask-cluster-04574796-5.default-tenant:8786
remote dashboard: default-tenant.app.yh38.iguazio-cd2.com:32577
> --------------- Iteration: (1) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:21,353 [info] p2=2, p3=10, r1=20 at mlrun-dask-cluster-04574796-5k5lhq

> --------------- Iteration: (3) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0
```

(continues on next page)

(continued from previous page)

```
[5 rows x 5 columns]
> 2021-10-23 12:50:21,459 [info] p2=4, p3=10, r1=40 at mlrun-dask-cluster-04574796-5k5lhq

> --------------- Iteration: (4) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:21,542 [info] p2=1, p3=10, r1=10 at mlrun-dask-cluster-04574796-5k5lhq

> --------------- Iteration: (6) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:21,629 [info] p2=1, p3=20, r1=20 at mlrun-dask-cluster-04574796-5k5lhq

> --------------- Iteration: (7) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:21,792 [info] p2=4, p3=20, r1=80 at mlrun-dask-cluster-04574796-5k5lhq

> --------------- Iteration: (8) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
2                4.7               3.2  ...               0.2      0
3                4.6               3.1  ...               0.2      0
4                5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:22,052 [info] p2=1, p3=20, r1=20 at mlrun-dask-cluster-04574796-5k5lhq

> --------------- Iteration: (2) ---------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                5.1               3.5  ...               0.2      0
1                4.9               3.0  ...               0.2      0
```

(continues on next page)

(continued from previous page)

```
2                 4.7               3.2  ...               0.2      0
3                 4.6               3.1  ...               0.2      0
4                 5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:23,134 [info] p2=1, p3=10, r1=10 at mlrun-dask-cluster-04574796-5j6v59

> ---------------- Iteration: (5) ----------------
   sepal length (cm)  sepal width (cm)  ...  petal width (cm)  label
0                 5.1               3.5  ...               0.2      0
1                 4.9               3.0  ...               0.2      0
2                 4.7               3.2  ...               0.2      0
3                 4.6               3.1  ...               0.2      0
4                 5.0               3.6  ...               0.2      0

[5 rows x 5 columns]
> 2021-10-23 12:50:23,219 [info] p2=2, p3=20, r1=40 at mlrun-dask-cluster-04574796-5k5lhq

> 2021-10-23 12:50:23,261 [info] tearing down the dask cluster..
> 2021-10-23 12:50:43,363 [info] best iteration=7, used criteria r1
> 2021-10-23 12:50:43,626 [info] run executed, status=completed
final state: completed
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:50:53,303 [info] run executed, status=completed
```

Running the workers using Nuclio

Nuclio is a high-performance serverless engine that can process many events in parallel. It can also separate initialization from execution. Certain parts of the code (imports, loading data, etc.) can be done once per worker vs. in any run.

Nuclio, by default, process events (http, stream, ...). There is a special Nuclio kind that runs MLRun jobs (nuclio:mlrun).

Notes

- Nuclio tasks are relatively short (preferably under 5 minutes), use it for running many iterations where each individual run is less than 5 min.
- Use `context.logger` to drive text outputs (vs `print()`).

Create a nuclio:mlrun function

```
fn = mlrun.code_to_function(name='hyper-tst2', kind='nuclio:mlrun', image='mlrun/mlrun')
# replicas * workers need to match or exceed parallel_runs
fn.spec.replicas = 2
fn.with_http(workers=2)
fn.deploy()
```

```
> 2021-10-23 12:51:10,152 [info] Starting remote function deploy
2021-10-23 12:51:10  (info) Deploying function
2021-10-23 12:51:10  (info) Building
2021-10-23 12:51:10  (info) Staging files and preparing base images
2021-10-23 12:51:10  (info) Building processor image
2021-10-23 12:51:11  (info) Build complete
2021-10-23 12:51:19  (info) Function deploy complete
> 2021-10-23 12:51:22,296 [info] successfully deployed function: {'internal_invocation_
↪urls': ['nuclio-default-hyper-tst2.default-tenant.svc.cluster.local:8080'], 'external_
↪invocation_urls': ['default-tenant.app.yh38.iguazio-cd2.com:32760']}
```

```
'http://default-tenant.app.yh38.iguazio-cd2.com:32760'
```

Run the parallel task over the function

```
# this is required to fix Jupyter issue with asyncio (not required outside of Jupyter)
# run it only once
import nest_asyncio
nest_asyncio.apply()
```

```
grid_params = {"p2": [2,1,4,1], "p3": [10,20]}
task = mlrun.new_task(params={"p1": 8}, inputs={'data': 'https://s3.wasabisys.com/
↪iguazio/data/iris/iris_dataset.csv'})
task.with_hyper_params(
    grid_params, selector="r1", strategy="grid", parallel_runs=4, max_errors=3
)
run = fn.run(task, handler=hyper_func2)
```

```
> 2021-10-23 12:51:31,618 [info] starting run hyper-tst2-hyper_func2␣
↪uid=97cc3e255f3c4c93822b0154d63f47f5 DB=http://mlrun-api:8080
> --------------- Iteration: (4) ---------------
2021-10-23 12:51:32.130812  info   logging run results to: http://mlrun-api:8080  worker_
↪id=1
2021-10-23 12:51:32.401258  info   p2=1, p3=10, r1=10 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=1

> --------------- Iteration: (2) ---------------
2021-10-23 12:51:32.130713  info   logging run results to: http://mlrun-api:8080  worker_
↪id=0
2021-10-23 12:51:32.409468  info   p2=1, p3=10, r1=10 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=0
```

(continues on next page)

(continued from previous page)

```
> --------------- Iteration: (1) ---------------
2021-10-23 12:51:32.130765  info   logging run results to: http://mlrun-api:8080  worker_
↪id=0
2021-10-23 12:51:32.432121  info   p2=2, p3=10, r1=20 at nuclio-default-hyper-tst2-
↪5d4976b685-2gdtc  worker_id=0

> --------------- Iteration: (5) ---------------
2021-10-23 12:51:32.568848  info   logging run results to: http://mlrun-api:8080  worker_
↪id=0
2021-10-23 12:51:32.716415  info   p2=2, p3=20, r1=40 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=0

> --------------- Iteration: (7) ---------------
2021-10-23 12:51:32.855399  info   logging run results to: http://mlrun-api:8080  worker_
↪id=1
2021-10-23 12:51:33.054417  info   p2=4, p3=20, r1=80 at nuclio-default-hyper-tst2-
↪5d4976b685-2gdtc  worker_id=1

> --------------- Iteration: (6) ---------------
2021-10-23 12:51:32.970002  info   logging run results to: http://mlrun-api:8080  worker_
↪id=0
2021-10-23 12:51:33.136621  info   p2=1, p3=20, r1=20 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=0

> --------------- Iteration: (3) ---------------
2021-10-23 12:51:32.541187  info   logging run results to: http://mlrun-api:8080  worker_
↪id=1
2021-10-23 12:51:33.301200  info   p2=4, p3=10, r1=40 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=1

> --------------- Iteration: (8) ---------------
2021-10-23 12:51:33.419442  info   logging run results to: http://mlrun-api:8080  worker_
↪id=0
2021-10-23 12:51:33.672165  info   p2=1, p3=20, r1=20 at nuclio-default-hyper-tst2-
↪5d4976b685-47dh6  worker_id=0

> 2021-10-23 12:51:34,153 [info] best iteration=7, used criteria r1
```

```
<IPython.core.display.HTML object>
```

```

```

```
<IPython.core.display.HTML object>
```

```
> 2021-10-23 12:51:34,420 [info] run executed, status=completed
```

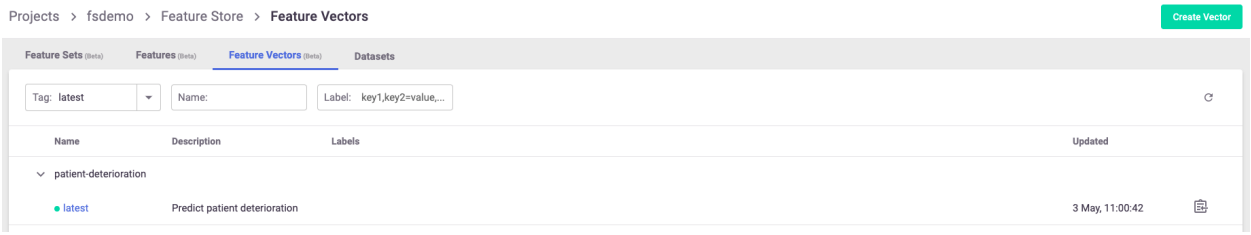
13.2 Training with the feature store

In this section

13.2.1 Creating an offline dataset

An offline dataset is a specific instance of the *feature vector definition*. To create this instance, use the feature store's `get_offline_features(<feature_vector>, <target>)` function on the feature vector using the `store://<project_name>/<feature_vector>` reference and an offline target (as in Parquet, CSV, etc.).

You can add a time-based filter condition when running `get_offline_feature` with a given vector. You can also filter with the query argument on all the other features as you like. See *`get_offline_features()`*.




```
import mlrun.feature_store as fstore

feature_vector = '<feature_vector_name>'
offline_fv = fstore.get_offline_features(feature_vector=feature_vector,␣
↪target=ParquetTarget())
```

Behind the scenes, `get_offline_features()` runs a local or Kubernetes job (can be specific by the `run_config` parameter) to retrieve all the relevant data from the feature sets, merge them and return it to the specified `target` which can be a local parquet, AZ Blob store or any other type of available storage.

Once instantiated with a target, the feature vector holds a reference to the instantiated dataset and references it as its current offline source.

You can also use MLRun's `log_dataset()` to log the specific dataset to the project as a specific dataset resource.

13.2.2 Training

Training your model using the feature store is a fairly simple task. (The offline dataset can also be used for your EDA.)

To retrieve a feature vector's offline dataset, use MLRun's data item mechanism, referencing the feature vector and specifying to receive it as a DataFrame.

```
df = mlrun.get_dataitem(f'store://feature-vectors/{project}/patient-deterioration').as_
↪df()
```

When trying to retrieve the dataset in your training function, you can put the feature vector reference as an input to the function and use the `as_df()` function to retrieve it automatically.

```
# A sample MLRun training function
def my_training_function(context, # MLRun context
                         dataset, # our feature vector reference
                         **kwargs):

    # retrieve the dataset
    df = dataset.as_df()

    # The rest of your training code...
```

And now you can create the MLRun function and run it locally or over the kubernetes cluster:

```
# Creating the training MLRun function with the code
fn = mlrun.code_to_function('training',
                            kind='job',
                            handler='my_training_function')

# Creating the task to run the function with its dataset
task = mlrun.new_task('training',
                      inputs={'dataset': f'store://feature-vectors/{project}/{feature_
↪vector_name}'}) # The feature vector is given as an input to the function

# Running the function over the kubernetes cluster
fn.run(task) # Set local=True to run locally
```

CHAPTER

FOURTEEN

DEPLOY MODELS AND APPLICATIONS

One of the advantages of using MLRun, is simplifying the deployment process. Deployment is more than just model deployment. Models usually run as part of a greater system which requires data processing before and after executing the model as well as being part of a business application.

Generally, there are two main modes of deployment:

1. **Real-time deployment**: this is the process of having a data and models pipeline respond for real-time events. The challenge here is usually ensuring that the data processing is performed in the same way that the batch training was done and sending the response in low latency. MLRun includes a specialized serving graph that eases that creation of a data transformation pipeline as part of the model serving. Feature store support is another way of ensuring feature calculation remain consistent between the training process and the serving process. For an end-to-end demo of model serving, refer to the **Serving pre-trained ML/DL models tutorial**.
2. **Batch inference**: this includes a process that runs on a large dataset. The data is usually read from an offline source, such as files or databases and the result is also written to offline targets. It is common to set-up a schedule when running batch inference. For an end-to-end demo of batch inference, refer to the **batch inference and drift detection tutorial**.

In this section

14.1 Real-time deployment

MLRun can produce managed real-time serverless pipelines from various tasks, including MLRun models or standard model files. The pipelines use a real-time serverless engine, called Nuclio, which can be deployed anywhere and is capable of delivering intensive data, I/O, and compute workloads.

Serving a model begins by creating a serving function. This function can run one or more models. To load and call a model, one needs to provide a serving class. MLRun has built-in support for commonly used frameworks and therefore it is often convenient to start with *built-in classes*. You can also *create your own custom model serving class*. You can also find an example notebook that shows how to build and run a serving class.

MLRun serving supports advanced real-time data processing and model serving pipelines. For more details and examples, see the *MLRun serving pipelines* documentation.

In this section

14.1.1 Using built-in model serving classes

MLRun includes built-in classes for commonly used frameworks. While you can *create your own class*, it is often not necessary to write one if you use these standard classes.

The following table specifies, for each framework, the relevant pre-integrated image and the corresponding MLRun `ModelServer` serving class:

framework	image	serving class
Scikit-learn	mlrun/mlrun	mlrun.frameworks.sklearn.SklearnModelServer
TensorFlow.Keras	mlrun/ml-models	mlrun.frameworks.tf_keras.TFKerasModelServer
ONNX	mlrun/ml-models	mlrun.frameworks.onnx.ONNXModelServer
XGBoost	mlrun/ml-models	mlrun.frameworks.xgboost.XGBoostModelServer
LightGBM	mlrun/ml-models	mlrun.frameworks.lgbm.LGBMModelServer
PyTorch	mlrun/ml-models	mlrun.frameworks.pytorch.PyTorchModelServer

For GPU support, use the `mlrun/ml-models-gpu` image (adding GPU drivers and support).

Example

The following code shows how to create a basic serving model using Scikit-learn.

```
import os
import urllib.request
import mlrun

model_path = os.path.abspath('sklearn.pkl')

# Download the model file locally
urllib.request.urlretrieve(mlrun.get_sample_path('models/serving/sklearn.pkl'), model_
↪path)

# Set the base project name
project_name_base = 'serving-test'

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name_base, context="./", user_project=True)

serving_function_image = "mlrun/mlrun"
serving_model_class_name = "mlrun.frameworks.sklearn.SklearnModelServer"

# Create a serving function
serving_fn = mlrun.new_function("serving", project=project.name, kind="serving",␣
↪image=serving_function_image)

# Add a model, the model key can be anything we choose. The class will be the built-in␣
↪scikit-learn model server class
model_key = "scikit-learn"
serving_fn.add_model(key=model_key,
                     model_path=model_path,
                     class_name=serving_model_class_name)
```

After the serving function is created, you can test it:

```
# Test data to send
my_data = {"inputs":[[5.1, 3.5, 1.4, 0.2],[7.7, 3.8, 6.7, 2.2]]}

# Create a mock server in order to test the model
mock_server = serving_fn.to_mock_server()

# Test the serving function
mock_server.test(f"/v2/models/{model_key}/infer", body=my_data)
```

Similarly, you can deploy the serving function and test it with some data:

```
serving_fn.with_code(body=" ") # Workaround, required only for mlrun <= 1.0.2

# Deploy the serving function
serving_fn.apply(mlrun.auto_mount()).deploy()

# Check the result using the deployed serving function
serving_fn.invoke(path=f'/v2/models/{model_key}/infer',body=my_data)
```

14.1.2 Build your own model serving class

Model serving classes implement the full model serving functionality, which includes loading models, pre- and post-processing, prediction, explainability, and model monitoring.

Model serving classes must inherit from `mlrun.serving.V2ModelServer`, and at the minimum implement the `load()` (download the model file(s) and load the model into memory) and `predict()` (accept request payload and return prediction/inference results) methods.

The class is initialized automatically by the model server and can run locally as part of a nuclio serverless function, or as part of a real-time pipeline.

You need to implement two mandatory methods:

- **load()** — download the model file(s) and load the model into memory, note this can be done synchronously or asynchronously.
- **predict()** — accept request payload and return prediction/inference results.

You can override additional methods : `preprocess`, `validate`, `postprocess`, `explain`. You can add a custom api endpoint by adding the method `op_xx(event)`. Invoke it by calling the /xx (operation = xx).

In this section

Minimal sklearn serving function example

```
from cloudpickle import load
import numpy as np
import mlrun

class ClassifierModel(mlrun.serving.V2ModelServer):
    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

    def predict(self, body: dict) -> list:
        """Generate model predictions from sample"""
        feats = np.asarray(body['inputs'])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()
```

Test the function locally using the mock server:

```
import mlrun
from sklearn.datasets import load_iris

fn = mlrun.new_function('my_server', kind='serving')

# set the topology/router and add models
graph = fn.set_topology("router")
fn.add_model("model1", class_name="ClassifierModel", model_path="<path1>")
fn.add_model("model2", class_name="ClassifierModel", model_path="<path2>")

# create and use the graph simulator
server = fn.to_mock_server()
x = load_iris()['data'].tolist()
result = server.test("/v2/models/model1/infer", {"inputs": x})
```

load() method

In the load method, download the model from external store, run the algorithm/framework `load()` call, and do any other initialization logic.

The load runs synchronously (the deploy is stalled until load completes). This can be an issue for large models and cause a readiness timeout. You can increase the function `spec.readiness_timeout`, or alternatively choose async loading (load () runs in the background) by setting the function `spec.load_mode = "async"`.

The function `self.get_model()` downloads the model metadata object and main file (into `model_file` path). Additional files can be accessed using the returned `extra_data` (dict of dataitem objects).

The model metadata object is stored in `self.model_spec` and provides model parameters, metrics, schema, etc. Parameters can be accessed using `self.get_param(key)`. The parameters can be specified in the model or during the function/model deployment.

predict() method

The predict method is called when you access the `/infer` or `/predict` url suffix (operation). The method accepts the request object (as dict), see Model server API. And it should return the specified response object.

explain() method

The explain method provides a hook for model explainability, and is accessed using the `/explain` operation.

pre/post and validate hooks

You can overwrite the `preprocess`, `validate`, and `postprocess` methods for additional control The call flow is:

```
pre-process -> validate -> predict/explain -> post-process
```

Models, routers and graphs

Every serving function can host multiple models and logical steps. Multiple functions can connect in a graph to form complex real-time pipelines.

The basic serving function has a logical `router` with routes to multiple child `models`. The url or the message determines which model is selected, e.g. using the url schema:

```
/v2/models/<model>[/versions/<ver>]/operation
```

Note

The `model`, `version` and `operation` can also be specified in the message body to support streaming protocols (e.g. Kafka).

More complex routers can be used to support ensembles (send the request to all child models and aggregate the result), multi-armed-bandit, etc.

You can use a pre-defined Router class, or write your own custom router. Routera can route to models on the same function or access models on a separate function.

To specify the topology, router class and class args use `.set_topology()` with your function.

Creating a model serving function (service)

To provision a serving function, you need to create an MLRun function of type `serving`. This can be done by using the `code_to_function()` call from a notebook. You can also import an existing serving function/template from the Function Hub.

Example (run inside a notebook): this code converts a notebook to a serving function and adding a model to it:

```
from mlrun import code_to_function
fn = code_to_function('my-function', kind='serving')
fn.add_model('m1', model_path=<model-artifact/dir>, class_name='MyClass', x=100)
```

See `.add_model()` docstring for help and parameters.

See the full Model Server example.

If you want to use multiple versions for the same model, use `:` to separate the name from the version. For example, if the name is `mymodel:v2` it means model name `mymodel` version `v2`.

You should specify the `model_path` (url of the model artifact/dir) and the `class_name` name (or class `module.submodule.class`). Alternatively, you can set the `model_url` for calling a model that is served by another function (can be used for ensembles).

The function object(fn) accepts many options. You can specify replicas range (auto-scaling), cpu/gpu/mem resources, add shared volume mounts, secrets, and any other Kubernetes resource through the `fn.spec` object or fn methods.

For example, `fn.gpu(1)` means each replica uses one GPU.

To deploy a model, simply call:

```
fn.deploy()
```

You can also deploy a model from within an ML pipeline (check the various demos for details).

Model monitoring

Model activities can be tracked into a real-time stream and time-series DB. The monitoring data is used to create real-time dashboards and track model accuracy and drift. To set the tracking stream options, specify the following function spec attributes:

```
    fn.set_tracking(stream_path, batch, sample)
```

- **stream_path** — the v3io stream path (e.g. `v3io:///users/..`)
- **sample** — optional, sample every N requests
- **batch** — optional, send micro-batches every N requests

14.1.3 Test and deploy a model server

In this section

- *Testing the model*
- *Deploying the model*

Testing the model

MLRun provides a mock server as part of the `serving` runtime. This gives you the ability to deploy your serving function in your local environment for testing purposes.

```
serving_fn = code_to_function(name='myService', kind='serving', image='mlrun/mlrun')
serving_fn.add_model('my_model', model_path=model_file_path)
server = serving_fn.to_mock_server()
```

You can use test data and programmatically invoke the `predict()` method of mock server. In this example, the model is expecting a python dictionary as input.

```
my_data = '''{"inputs":[[5.1, 3.5, 1.4, 0.2],[7.7, 3.8, 6.7, 2.2]]}'''
server.test("/v2/models/my_model/infer", body=my_data)
```

The data structure used in the body parameter depends on how the `predict()` method of the model server is defined. For examples of how to define your own model server class, see here.

To review the mock server api, see here.

Deploying the model

Deploying models in MLRun uses a special function type `serving`. You can create a `serving` function using the `code_to_function()` call from a notebook. You can also import an existing serving function/template from the Function Hub.

This example converts a notebook to a serving function and adds a model to it:

```
from mlrun import code_to_function
fn = code_to_function('my-function', kind='serving')
fn.add_model('m1', model_path=<model-artifact/dir>, class_name='MyClass', x=100)
```

See `.add_model()` docstring for help and parameters.

See the full Model Server example.

If you want to use multiple versions for the same model, use `:` to separate the name from the version. For example, if the name is `mymodel:v2` it means model name `mymodel` version `v2`.

You should specify the `model_path` (url of the model artifact/dir) and the `class_name` name (or class `module.submodule.class`). Alternatively, you can set the `model_url` for calling a model that is served by another function (can be used for ensembles).

The function object(fn) accepts many options. You can specify replicas range (auto-scaling), cpu/gpu/mem resources, add shared volume mounts, secrets, and any other Kubernetes resource through the `fn.spec` object or fn methods.

For example, `fn.gpu(1)` means each replica uses one GPU.

To deploy a model, simply call:

```
fn.deploy()
```

You can also deploy a model from within an ML pipeline (check the various demos for details).

14.1.4 Model serving API

MLRun Serving follows the same REST API defined by Triton and KFServing v2.

Nuclio also supports streaming protocols (Kafka, kinesis, MQTT, etc.). When streaming, the `model` name and `operation` can be encoded inside the message body.

The APIs are:

- *explain*
- *get model health / readiness*
- *get model metadata*
- *get server info*
- *infer / predict*

- *list models*

explain

POST /v2/models/[/versions/{VERSION}]/explain

Request body:

```
{
  "id" : $string #optional,
  "model" : $string #optional
  "parameters" : $parameters #optional,
  "inputs" : [ $request_input, ... ],
  "outputs" : [ $request_output, ... ] #optional
}
```

Response structure:

```
{
  "model_name" : $string,
  "model_version" : $string #optional,
  "id" : $string,
  "outputs" : [ $response_output, ... ]
}
```

get model health / readiness

```
GET v2/models/${MODEL_NAME}[/versions/${VERSION}]/ready
```

Returns 200 for Ok, 40X for not ready.

get model metadata

```
GET v2/models/${MODEL_NAME}[/versions/${VERSION}]
```

Response example: `{"name": "m3", "version": "v2", "inputs": [..], "outputs": [..]}`

get server info

```
GET /
GET /v2/health
```

Response example: `{'name': 'my-server', 'version': 'v2', 'extensions': []}`

infer / predict

```
POST /v2/models/<model>[/versions/{VERSION}]/infer
```

Request body:

```
{
  "id" : $string #optional,
  "model" : $string #optional
  "data_url" : $string #optional
  "parameters" : $parameters #optional,
  "inputs" : [ $request_input, ... ],
  "outputs" : [ $request_output, ... ] #optional
}
```

- **id**: Unique Id of the request, if not provided a random value is provided.
- **model**: Model to select (for streaming protocols without URLs).
- **data_url**: Option to load the `inputs` from an external file/s3/v3io/... object.
- **parameters**: Optional request parameters.
- **inputs**: List of input elements (numeric values, arrays, or dicts).
- **outputs:** Optional, requested output values.

Note: You can also send binary data to the function, for example, a JPEG image. The serving engine pre-processor detects it based on the HTTP content-type and converts it to the above request structure, placing the image bytes array in the `inputs` field.

Response structure:

```
{
  "model_name" : $string,
  "model_version" : $string #optional,
  "id" : $string,
  "outputs" : [ $response_output, ... ]
}
```

list models

```
GET /v2/models/
```

Response example: `{"models": ["m1", "m2", "m3:v1", "m3:v2"]}`

14.2 Serving with the feature store

In this section

14.2.1 Get online features

The online features are created ad-hoc using MLRun's feature store online feature service and are served from the **nosql** target for real-time performance needs.

To use it, first create an online feature service with the feature vector.

```
import mlrun.feature_store as fstore

svc = fstore.get_online_feature_service(<feature vector name>)
```

After creating the service you can use the feature vector's entity to get the latest feature vector for it. Pass a list of {<key name>: <key value>} pairs to receive a batch of feature vectors.

```
fv = svc.get([{<key name>: <key value>}])
```

14.2.2 Incorporating to the serving model

You can serve your models using the *Real-time serving pipelines (graphs)*. (See a V2 Model Server (SKLearn) example.) You define a serving model class and the computational graph required to run your entire prediction pipeline, and deploy it as a serverless function using nuclio.

To embed the online feature service in your model server, just create the feature vector service once when the model initializes, and then use it to retrieve the feature vectors of incoming keys.

You can import ready-made classes and functions from the MLRun Function Hub or write your own. As example of a scikit-learn based model server:

```
from cloudpickle import load
import numpy as np
import mlrun
import os

class ClassifierModel(mlrun.serving.V2ModelServer):

    def load(self):
        """load and initialize the model and/or other elements"""
        model_file, extra_data = self.get_model('.pkl')
        self.model = load(open(model_file, 'rb'))

        # Setup FS Online service
        self.feature_service = mlrun.feature_store.get_online_feature_service('patient-
↪deterioration')

        # Get feature vector statistics for imputing
```

(continues on next page)

(continued from previous page)

```
        self.feature_stats = self.feature_service.vector.get_stats_table()

    def preprocess(self, body: dict, op) -> list:
        # Get patient feature vector
        # from the patient_id given in the request
        vectors = self.feature_service.get([{'patient_id': patient_id} for patient_id in␣
↪body['inputs']])

        # Impute inf's in the data to the feature's mean value
        # using the collected statistics from the Feature store
        feature_vectors = []
        for fv in vectors:
            new_vec = []
            for f, v in fv.items():
                if np.isinf(v):
                    new_vec.append(self.feature_stats.loc[f, 'mean'])
                else:
                    new_vec.append(v)
            feature_vectors.append(new_vec)

        # Set the final feature vector as the inputs
        # to pass to the predict function
        body['inputs'] = feature_vectors
        return body

    def predict(self, body: dict) -> list:
        """Generate model predictions from sample"""
        feats = np.asarray(body['inputs'])
        result: np.ndarray = self.model.predict(feats)
        return result.tolist()
```

Which you can deploy with:

```
# Create the serving function from the code above
fn = mlrun.code_to_function(<function_name>,
                            kind='serving')

# Add a specific model to the serving function
fn.add_model(<model_name>,
             class_name='ClassifierModel',
             model_path=<store_model_file_reference>)

# Enable MLRun's model monitoring
fn.set_tracking()

# Add the system mount to the function so
# it will have access to the model files
fn.apply(mlrun.mount_v3io())

# Deploy the function to the cluster
fn.deploy()
```

And test using:

```
fn.invoke('/v2/models/infer', body={<key name>: <key value>})
```

14.3 Batch inference

Batch inference or offline inference addresses the need to run machine learning model on large datasets. It is the process of generating outputs on a batch of observations.

With batch inference, the batch runs are typically generated during some recurring schedule (e.g., hourly, or daily). These inferences are then stored in a database or a file and can be made available to developers or end users. With batch inference, the goal is usually tied to time constraints and the service-level agreement (SLA) of the job. Conversely, in real time serving, the goal is usually to optimize the number of transactions per second that the model can process. An online application displays a result to the user.

Batch inference can sometimes take advantage of big data technologies such as Spark to generate predictions. Big data technologies allows data scientists and machine learning engineers to take advantage of scalable compute resources to generate many predictions at once.

14.3.1 Test your model

To evaluate batch model prior to deployment, you should use the `evaluate` handler of the `auto_trainer` function.

This is typically done during model development. For more information refer to the *Evaluate* handler documentation. For example:

```
import mlrun

# Set the base project name
project_name_base = 'batch-inference'

# Initialize the MLRun project object
project = mlrun.get_or_create_project(project_name_base, context="./", user_project=True)

auto_trainer = mlrun.import_function(mlrun.import_function("hub://auto_trainer"))

evaluate_run = project.run_function(
    auto_trainer,
    handler="evaluate",
    inputs={"dataset": train_run.outputs['test_set']},
    params={
        "model": train_run.outputs['model'],
        "label_columns": "labels",
    },
)
```

14.3.2 Deploy your model

Batch inference is implemented in MLRun by running the function with an input dataset. With MLRun you can easily create any custom logic in a function, including loading a model and calling it.

The Function Hub batch inference function is used for running the models in batch as well as performing drift analysis. The function supports the following frameworks:

- Scikit-learn
- XGBoost
- LightGBM
- Tensorflow/Keras
- PyTorch
- ONNX

Internally the function uses MLRun's out-of-the-box capability to load run a model via the *mlrun.frameworks.auto_mlrun.auto_mlrun.AutoMLRun* class.

Basic example

The simplest example to run the function is as follows:

Create project

Import MLRun and create a project:

```
import mlrun
project = mlrun.get_or_create_project('batch-inference', context="./", user_project=True)
batch_inference = mlrun.import_function("hub://batch_inference")
```

Get the model

Get the model. The model is a decision tree classifier from scikit-learn. Note that if you previously trained your model using MLRun, you can reference the model artifact produced during that training process.

```
model_path = mlrun.get_sample_path('models/batch-predict/model.pkl')

model_artifact = project.log_model(
    key="model",
    model_file=model_path,
    framework="sklearn"
)
```

Get the data

Get the dataset to perform the inference. The dataset is in `parquet` format.

```
prediction_set_path = mlrun.get_sample_path('data/batch-predict/prediction_set.parquet')
```

Run the batch inference function

Run the inference. In the first example we will not perform any drift analysis

```
batch_run = project.run_function(
    batch_inference,
    inputs={"dataset": prediction_set_path},
    params={"model": model_artifact.uri},
)
```

Function output

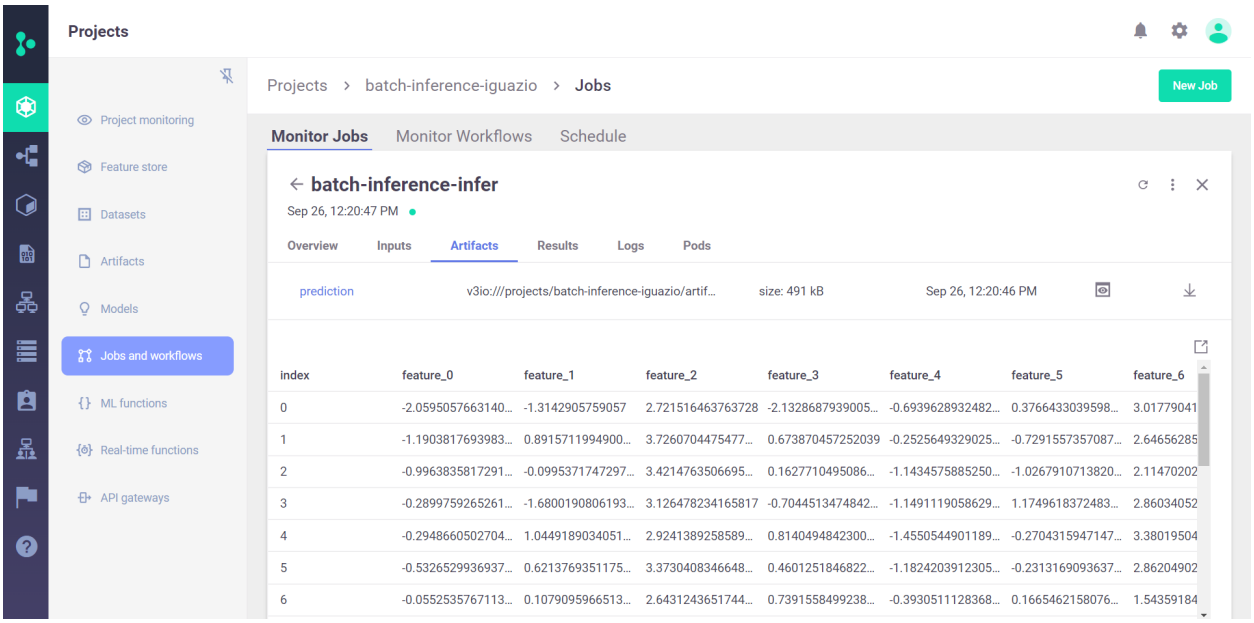
The output of the function is an artifact called `prediction`:

```
batch_run.artifact("prediction").as_df().head()
```

```
   feature_0  feature_1  feature_2  feature_3  feature_4  feature_5  \
0  -2.059506  -1.314291   2.721516  -2.132869  -0.693963   0.376643
1  -1.190382   0.891571   3.726070   0.673870  -0.252565  -0.729156
2  -0.996384  -0.099537   3.421476   0.162771  -1.143458  -1.026791
3  -0.289976  -1.680019   3.126478  -0.704451  -1.149112   1.174962
4  -0.294866   1.044919   2.924139   0.814049  -1.455054  -0.270432

   feature_6  feature_7  feature_8  feature_9  ...  feature_11  feature_12  \
0   3.017790   3.876329  -1.294736   0.030773  ...    2.775699    2.361580
1   2.646563   4.782729   0.318952  -0.781567  ...    1.101721    3.723400
2   2.114702   2.517553  -0.154620  -0.465423  ...    1.729386    2.820340
3   2.860341   3.753661  -0.326119   2.128411  ...    2.328688    3.397321
4   3.380195   2.339669   1.029101  -1.171018  ...    1.283565    0.677006

   feature_13  feature_14  feature_15  feature_16  feature_17  feature_18  \
0    0.173441    0.879510    1.141007    4.608280   -0.518388    0.129690
1   -0.466867   -0.056224    3.344701    0.194332    0.463992    0.292268
2   -1.041428   -0.331871    2.909172    2.138613   -0.046252   -0.732631
3   -0.932060   -1.442370    2.058517    3.881936    2.090635   -0.045832
4   -2.147444   -0.494150    3.222041    6.219348   -1.914110    0.317786

   feature_19  predicted_label
0    2.794967                0
1    4.665876                0
2    4.716266                0
3    4.197315                1
4    4.143443                1

[5 rows x 21 columns]
```

View the results in the UI

The output is saved as a parquet file under the project artifact path. In the UI you can go to the `batch-inference-infer` job –> artifact tab to view the details.



Scheduling a batch run

To schedule a run, you can set the schedule parameter of the run method. The scheduling is done by using a cron format.

You can also schedule runs from the dashboard. On the Projects > Jobs and Workflows page, you can create a new job using the New Job wizard. At the end of the wizard you can set the job scheduling. In the following example, the job is set to run every 30 minutes.

```
batch_run = project.run_function(
    batch_inference,
    inputs={"dataset": prediction_set_path},
    params={"model": model_artifact.uri},
    schedule='*/30 * * * *'
)
```

Drift analysis

By default, if a model has a sample set statistics, `batch_inference` performs drift analysis and will produce a data drift table artifact, as well as numerical drift metrics.

To provide sample set statistics for the model you can either:

1. Train the model using MLRun. This allows you to create the sample set during training.
2. Log an external model using `project.log_model` method and provide the training set in the `training_set` parameter.
3. Provide the set explicitly when calling the `batch_inference` function via the `sample_set` input.

In the example below, we will provide the training set as the sample set

```
training_set_path = mlrun.get_sample_path('data/batch-predict/training_set.parquet')

batch_run = project.run_function(
    batch_inference,
    inputs={
        "dataset": prediction_set_path,
        "sample_set": training_set_path
    },
    params={"model": model_artifact.uri,
            "label_columns": "label",
            "perform_drift_analysis" : True}
)
```

In this case, instead of just prediction, you will get drift analysis. The drift table plot that compares the drift between the training data and prediction data per feature:

```
batch_run.artifact("drift_table_plot").show()
```

	Count		Mean		Std		Min		Max		Tvd	Hellinger	Kld	Histograms
	Sample	Input	Sample	Input	Sample	Input	Sample	Input	Sample	Input				sample input
feature_10	2.5k	2.5k	0.0099619	-0.028823	1.0024	1.015	-3.468	-3.252	3.6221	3.4528	0.0404	0.044734	0.016108	
feature_0	2.5k	2.5k	-0.0040279	0.021953	0.99581	0.98954	-3.549	-3.267	3.4002	3.4347	0.0226	0.033574	0.0074546	
feature_17	2.5k	2.5k	-0.0034057	-0.018602	1.0027	0.98876	-3.481	-3.457	3.2288	3.3332	0.0328	0.038956	0.0090139	
feature_3	2.5k	2.5k	0.015564	-0.0099402	0.98579	0.99494	-3.986	-3.464	3.5643	3.217	0.0384	0.04914	0.017332	
feature_15	2.5k	2.5k	0.019173	2.9823	1.2	1.3556	-3.654	-1.256	3.2802	6.4363	0.519	0.74682	3.7154	
feature_12	2.5k	2.5k	0.020975	3.0041	0.99692	1.1486	-3.07	-0.86306	3.0677	6.7735	0.599	0.80796	4.5742	
feature_4	2.5k	2.5k	0.0043483	0.0038642	1.2859	1.2864	-3.757	-3.742	4.375	4.7429	0.0386	0.046911	0.01621	
feature_7	2.5k	2.5k	-0.022209	3.0088	1.0213	1.1624	-3.3	-0.62968	3.4935	6.7219	0.6646	0.79498	4.9492	
feature_6	2.5k	2.5k	-0.011094	3.0164	1.0216	1.1284	-3.258	-1.085	3.4455	7.1375	0.6598	0.79261	4.6801	
feature_16	2.5k	2.5k	0.017349	3.0134	0.97533	1.1673	-3.592	-0.82201	3.2731	7.0476	0.636	0.80032	4.6827	
feature_19	2.5k	2.5k	0.009379	2.9819	0.99404	1.1578	-3.745	-0.42378	4.5189	7.1048	0.7812	0.79934	6.9668	
feature_18	2.5k	2.5k	0.00756	0.01097	0.98851	1.0066	-3.753	-3.349	3.8614	3.4863	0.0424	0.046475	0.015894	
feature_8	2.5k	2.5k	0.014922	-0.023379	0.82918	0.84283	-2.13	-2.085	1.4455	1.6216	0.0384	0.03972	0.0094354	
label	2.5k	2.5k	0.4952	0.5156	0.50008	0.49986	0	0	1	1	0.0204	0.014427	0.0016651	
feature_14	2.5k	2.5k	0.013093	-0.0087792	1.0037	1.0024	-3.563	-3.216	3.3913	3.1733	0.038	0.054729	0.022306	
feature_1	2.5k	2.5k	0.018511	0.00019738	1.0159	0.99085	-3.159	-3.086	3.8723	4.213	0.0434	0.046301	0.014582	
feature_5	2.5k	2.5k	-0.029516	-0.0017621	0.98998	0.99237	-3.288	-3.235	3.0054	3.3326	0.0496	0.054084	0.017044	
feature_13	2.5k	2.5k	-0.0056543	0.019551	1.3625	1.3556	-4.484	-4.512	4.283	4.629	0.0396	0.045194	0.013241	
feature_11	2.5k	2.5k	0.0060043	3.0076	0.97702	1.1349	-3.991	-0.61732	3.2947	6.5215	0.6384	0.80589	4.8005	
feature_2	2.5k	2.5k	-0.009449	2.992	0.98855	1.1657	-3.242	-0.58678	3.697	7.0218	0.6882	0.79006	5.2144	
feature_9	2.5k	2.5k	0.023047	0.045638	0.98049	0.98343	-3.287	-4.242	3.6956	3.0063	0.0428	0.046567	0.012618	

You also get a numerical drift metric and boolean flag denoting whether or not data drift is detected:

```
print(batch_run.status.results)
```

```
{'drift_status': False, 'drift_metric': 0.29934242566253266}
```

```
# Data/concept drift per feature (use batch_run.artifact("features_drift_results").get()␣
↪to obtain the raw data)
batch_run.artifact("features_drift_results").show()
```


```
{'feature_0': 0.028086840976606773,
 'feature_1': 0.04485072701663093,
 'feature_2': 0.7391279921664593,
 'feature_3': 0.043769819014849734,
 'feature_4': 0.042755641152500176,
 'feature_5': 0.05184219833790496,
 'feature_6': 0.7262042202197605,
 'feature_7': 0.7297906294873706,
 'feature_8': 0.039060131873550404,
 'feature_9': 0.04468363504674985,
 'feature_10': 0.042567035578799796,
 'feature_11': 0.7221431701127441,
 'feature_12': 0.7034787615778625,
 'feature_13': 0.04239724655474124,
 'feature_14': 0.046364723781764774,
 'feature_15': 0.6329075683793959,
 'feature_16': 0.7181622588902428,
 'feature_17': 0.03587785749574268,
 'feature_18': 0.04443732609382538,
 'feature_19': 0.7902698698155215,
 'label': 0.017413285340161608}
```


14.3.3 `batch_inference` Parameters

Model Parameters

- `model`: `str` — The model store path.

Inference parameters

Parameters to specify the dataset for inference.

- `dataset`: `DatasetType` — The dataset to infer through the model. Can be passed in `inputs` as either a Dataset artifact / Feature vector URI or in `parameters` as a list, dictionary or numpy array.
- `drop_columns`: `Union[str, int, List[str], List[int]]` — A string / integer or a list of strings / integers that represent the column names / indices to drop. When the dataset is a list or a numpy array this parameter must be represented by integers.
- `label_columns`: `Union[str, List[str]]` — The target label(s) of the column(s) in the dataset for Regression or classification tasks. The label column can be accessed from the model object, or the feature vector provided if available.
- `predict_kwargs`: `Dict[str, Any]` — Additional parameters to pass to the prediction of the model.

Drift parameters

Parameters that affect the drift calculation.

- `perform_drift_analysis`: `bool` = `None` — Whether to perform drift analysis between the sample set of the model object to the dataset given. By default, None, which means it will perform drift analysis if the model has a sample set statistics. Perform drift analysis will produce a data drift table artifact.
- `sample_set`: `DatasetType` — A sample dataset to give to compare the inputs in the drift analysis. The default chosen sample set will always be the one who is set in the model artifact itself.
- `drift_threshold`: `float` = `0.7` — The threshold of which to mark drifts. Default is 0.7.
- `possible_drift_threshold`: `float` = `0.5` — The threshold of which to mark possible drifts. Default is 0.5.
- `inf_capping`: `float` = `10.0` — The value to set for when it reached infinity. Default is 10.0.

Logging parameters

Parameters to control the automatic logging feature of MLRun. You can adjust the logging outputs as relevant and if not passed, a default list of artifacts and metrics is produced and calculated.

- `log_result_set`: `bool` = `True` — Whether to log the result set - a DataFrame of the given inputs concatenated with the predictions. Default is True.
- `result_set_name`: `str` = `"prediction"` — The db key to set name of the prediction result and the filename Default is 'prediction'.
- `artifacts_tag`: `str` — Tag to use for all the artifacts resulted from the function.

14.4 Canary and rolling upgrades

Note

Relevant when MLRun is executed in the Iguazio platform (**"the platform"**).

Canary rollout is a known practice to first test a software update on a small number of users before rolling it out to all users. In machine learning, the main usage is to test a new model on a small subset of users before rolling it out to all users.

Canary functions are defined using an API gateway. The API gateway is a service that exposes your function as a web service. Essentially, it is a proxy that forwards requests to your functions and returns the response. You can configure authentication on the gateway.

The API traffic is randomly directed to the two functions at the percentages you specify. Start with a low percentage for the canary function. Verify that the canary function works as expected (or modify it until it does work as desired). Then gradually increase its percentage until you turn it into a production function.

In this section

- *Create an API gateway*
- *Create and use a canary function*

14.4.1 Create an API gateway

To create an API gateway in the UI:

1. In your project page, press **API Gateways** tab, then press **NEW API GATEWAY**.
2. Select an **Authentication Mode**:
   - None (default)
   - Basic
   - Access key
   - OAuth2

   and fill in any required values.
3. Type in the API Gateway parameters:
   - **Name**: The name of the API Gateway. Required
   - **Description**: A description of the API Gateway.
   - **Host**: The host of the API Gateway. (Relevant for open-source only.)
   - **Path**: The path of the API Gateway.
4. In **Primary**, type in the function that is triggered via the API Gateway.

14.4.2 Create and use a canary function

1. Press **Create a canary function** and type in the function name.
2. Leave the percentages at 5% and 95% to get started, and verify that the canary function works as expected.
3. Gradually increase the percentage, each time verifying its results.
4. When the percentage is high and you are fully satisfied, turn it into a production function by pressing > **Promote**.

CHAPTER

FIFTEEN

MONITOR AND ALERT

Note: Monitoring required at the moment Iguazio's streaming technology. Open-source integration with Kafka is under development.

Note: This is currently a beta feature.

The MLRun's model monitoring service includes built-in model monitoring and reporting capability. With monitoring you get out-of-the-box analysis of:

- **Model performance**: machine learning models train on data. It is important you know how well they perform in production. When you analyze the model performance, it is important you monitor not just the overall model performance, but also the feature-level performance. This gives you better insights for the reasons behind a particular result
- **Data drift**: the change in model input data that potentially leads to model performance degradation. There are various statistical metrics and drift metrics that you can use in order to identify data drift.
- **Concept drift**: applies to the target. Sometimes the statistical properties of the target variable, which the model is trying to predict, change over time in unforeseen ways.
- **Operational performance**: applies to the overall health of the system. This applies to data (e.g., whether all the expected data arrives to the model) as well as the model (e.g., response time, and throughput).

You have the option to set up notifications on various channels once an issue is detection. For example, you can set-up notification to your IT via email and slack when operational performance metrics pass a threshold. You can also set-up automated actions, for example, call a CI/CD pipeline when data drift is detected and allow a data scientist to review the model with the revised data.

Refer to the **model monitoring & drift detection tutorial** for an end-to-end example.

In this section

15.1 Model monitoring overview

Note: This is currently a beta feature.

In this section

15.1.1 Architecture

The model monitoring process flow starts with collecting operational data. The operational data are converted to vectors, which are posted to the Model Server. The model server is then wrapped around a machine learning model that uses a function to calculate predictions based on the available vectors. Next, the model server creates a log for the input and output of the vectors, and the entries are written to the production data stream (a v3io stream). While the model server is processing the vectors, a Nuclio operation monitors the log of the data stream and is triggered when a new log entry is detected. The Nuclio function examines the log entry, processes it into statistics which are then written to the statistics databases (parquet file, time series database and key value database). The parquet files are written as a feature set under the model monitoring project. The parquet files can be read either using `pandas.read_parquet` or `feature_set.get_offline_features`, like any other feature set. In parallel, a scheduled MLRun job runs reading the parquet files, performing drift analysis. The drift analysis data is stored so that the user can retrieve it in the Iguazio UI or in a Grafana dashboard.

Drift analysis

The model monitoring feature provides drift analysis monitoring. Model drift in machine learning is a situation where the statistical properties of the target variable (what the model is trying to predict) change over time. In other words, the production data has changed significantly over the course of time and no longer matches the input data used to train the model. So, for this new data, accuracy of the model predictions is low. Drift analysis statistics are computed once an hour. For more information see Concept Drift.

Common terminology

The following terms are used in all the model monitoring pages:

- **Total Variation Distance** (TVD) — The statistical difference between the actual predictions and the model's trained predictions.
- **Hellinger Distance** — A type of f-divergence that quantifies the similarity between the actual predictions, and the model's trained predictions.
- **Kullback–Leibler Divergence** (KLD) — The measure of how the probability distribution of actual predictions is different from the second model's trained reference probability distribution.
- **Model Endpoint** — A combination of a deployed Nuclio function and the models themselves. One function can run multiple endpoints; however, statistics are saved per endpoint.

15.1.2 Model monitoring using the Iguazio platform interface

Iguazio's model monitoring data is available for viewing through the regular platform interface. The platform provides four information pages with model monitoring data.

- *Model endpoint summary list*
- *Model endpoint overview*
- *Model drift analysis*
- *Model features analysis*

1. Select a project from the project tiles screen.
2. From the project dashboard, press the **Models** tile to view the models currently deployed .
3. Press **Model Endpoints** from the menu to display a list of monitored endpoints. If the Model Monitoring feature is not enabled, the endpoints list is empty.

Model endpoint summary list

The Model Endpoints summary list provides a quick view of the model monitoring data.

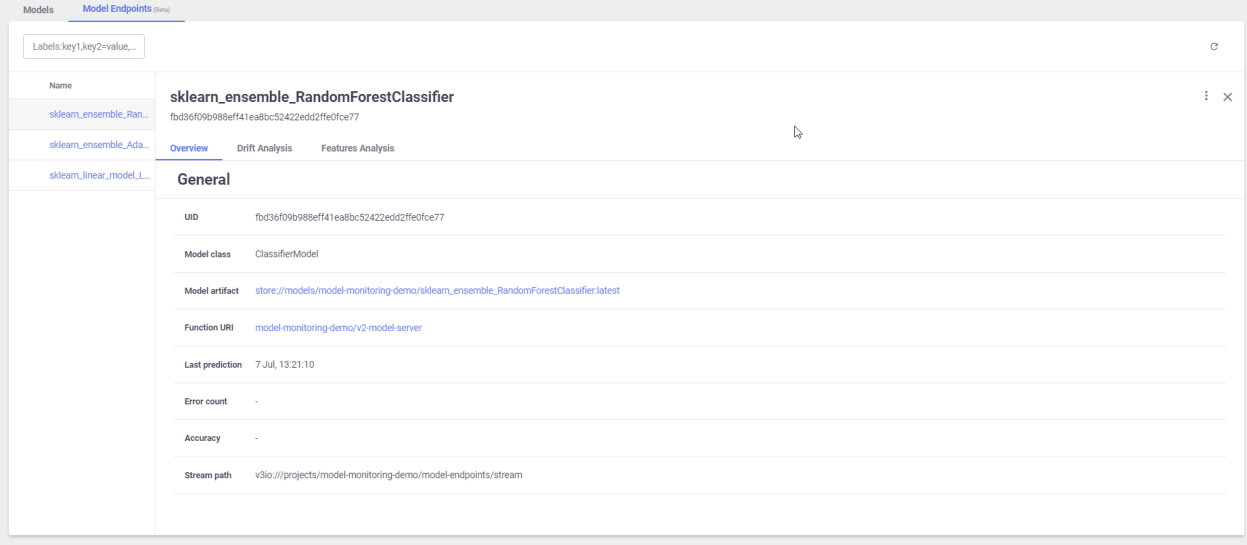
The summary page contains the following fields:

- **Name** — the name of the model endpoint
- **Version** — user configured version taken from model deployment
- **Class** — the implementation class that is used by the endpoint
- **Model** — user defined name for the model
- **Labels** — user configurable tags that are searchable
- **Uptime** — first request for production data
- **Last Prediction **— most recent request for production data
- **Error Count **— includes prediction process errors such as operational issues (For example, a function in a failed state), as well as data processing errors (For example, invalid timestamps, request ids, type mismatches etc.)
- **Drift** — indication of drift status (no drift (green), possible drift (yellow), drift detected (red))
- **Accuracy** — a numeric value representing the accuracy of model predictions (N/A)

Note: Model Accuracy is currently under development.

Model endpoint overview

The Model Endpoints overview pane displays general information about the selected model.



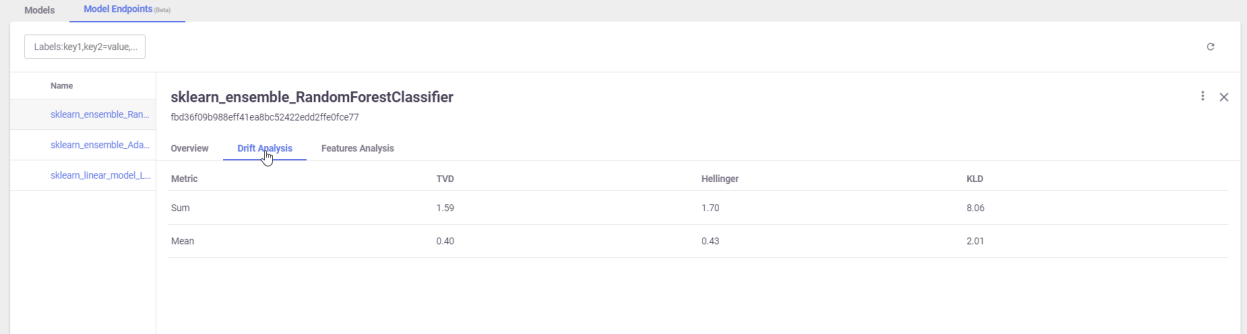
The Overview page contains the following fields:

- **UUID** — the ID of the deployed model
- **Model Class** — the implementation class that is used by the endpoint
- **Model Artifact** — reference to the model's file location
- **Function URI** — the MLRun function to access the model
- **Last Prediction** — most recent request for production data
- **Error Count** — includes prediction process errors such as operational issues (For example, a function in a failed state), as well as data processing errors (For example, invalid timestamps, request ids, type mismatches etc.)
- **Accuracy** — a numeric value representing the accuracy of model predictions (N/A)
- **Stream path** — the input and output stream of the selected model

Use the ellipsis to view the YAML resource file for details about the monitored resource.

Model drift analysis

The Drift Analysis pane provides performance statistics for the currently selected model.



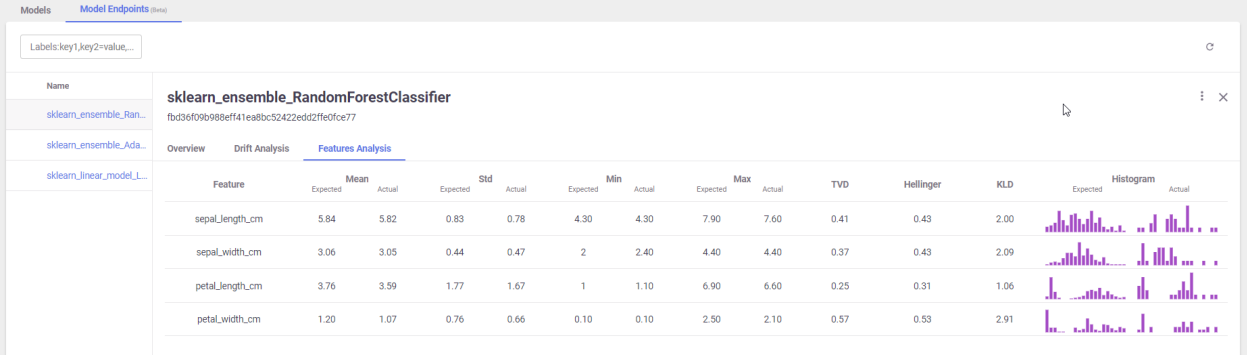
Each of the following fields has both sum and mean numbers displayed. For definitions of the terms see *Common Terminology*.

- **TVD**
- **Hellinger**
- **KLD**

Use the ellipsis to view the YAML resource file for details about the monitored resource.

Model features analysis

The Features Analysis pane provides details of the drift analysis in a table format with each feature in the selected model on its own line.



Each field has a pair of columns. The **Expected** column displays the results from the model training phase, and the **Actual** column displays the results from the live production data. The following fields are available:

- **Mean**
- **STD** (Standard deviation)
- **Min**
- **Max**
- **TVD**
- **Hellinger**
- **KLD**
- **Histograms**—the approximate representation of the distribution of the data. Hover over the bars in the graph for the details.

Use the ellipsis to view the YAML resource file for details about the monitored resource.

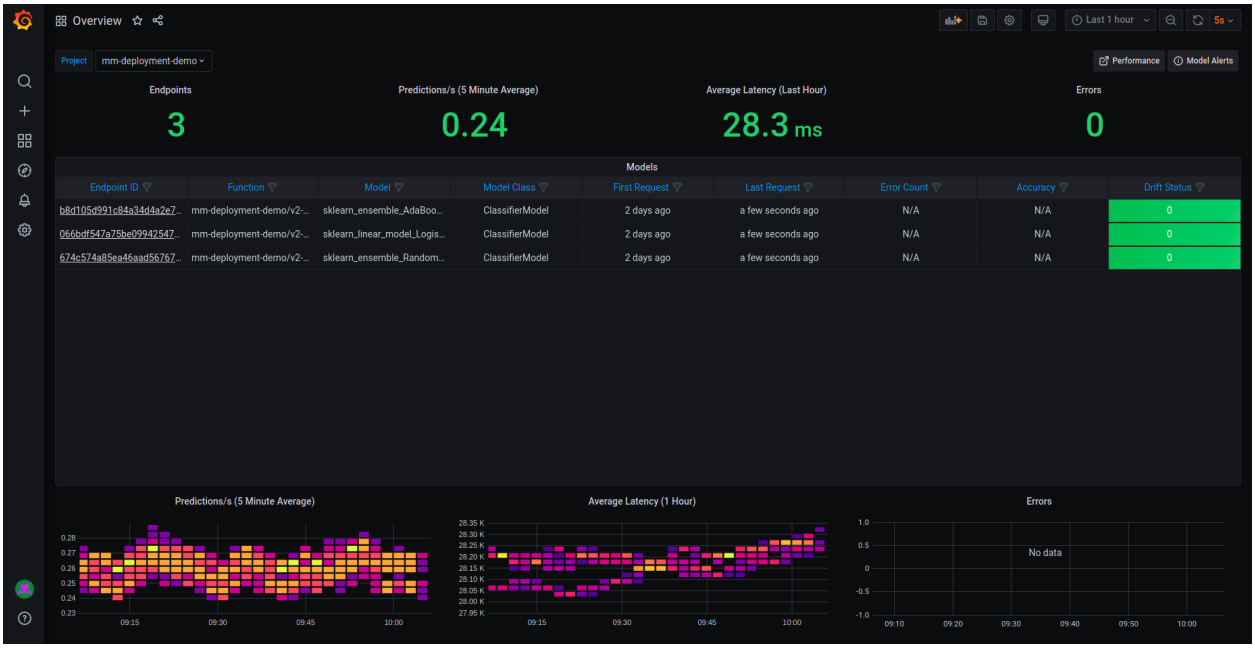
15.1.3 Model monitoring using Grafana dashboards

You can deploy a Grafana service in your Iguazio instance and use Grafana Dashboards to view model monitoring details. There are three dashboards available:

- *Overview Dashboard*
- *Details Dashboard*
- *Performance Dashboard*

Model endpoints overview dashboard

The Overview dashboard displays the model endpoint IDs of a specific project. Only deployed models with Model Monitoring enabled are displayed. Endpoint IDs are URIs used to provide access to performance data and drift detection statistics of a deployed model.



The Overview pane provides details about the performance of all the deployed and monitored models within a project. You can change projects by choosing a new project from the **Project** dropdown. The Overview dashboard displays the number of endpoints in the project, the average predictions per second (using a 5-minute rolling average), the average latency (using a 1-hour rolling average), and the total error count in the project.

Additional details include:

- **Endpoint ID** — the ID of the deployed model. Use this link to drill down to the model performance and details panes.
- **Function** — the MLRun function to access the model
- **Model** — user defined name for the model
- **Model Class** — the implementation class that is used by the endpoint
- **First Request** — first request for production data
- **Last Request** — most recent request for production data

- **Error Count** — includes prediction process errors such as operational issues (for example, a function in a failed state), as well as data processing errors (for example, invalid timestamps, request ids, type mismatches etc.)
- **Accuracy** — a numeric value representing the accuracy of model predictions (N/A)
- **Drift Status** — no drift (green), possible drift (yellow), drift detected (red)

At the bottom of the dashboard are heat maps for the Predictions per second, Average Latency and Errors. The heat maps display data based on 15 minute intervals. See *How to Read a Heat Map* for more details.

Click an endpoint ID to drill down the performance details of that model.

How to read a heat map

Heat maps are used to analyze trends and to instantly transform and enhance data through visualizations. This helps to quickly identify areas of interest, and empower users to explore the data in order to pinpoint where there may be potential issues. A heat map uses a matrix layout with colour and shading to show the relationship between two categories of values (x and y axes), so the darker the cell, the higher the value. The values presented along each axis correspond to a cell which is color-coded to represent the relationship between the two categories. The Predictions per second heatmap shows the relationship between time, and the predictions per second, and the Average Latency per hour shows the relationship between time and the latency.

To properly read the heap maps, follow the hierarchy of shades from the darkest (the highest values) to the lightest shades (the lowest values).

Note: The exact quantitative values represented by the colors may be difficult to determine. Use the *Performance Dashboard* to see detailed results.

Model endpoint details dashboard

The model endpoint details dashboard displays the real time performance data of the selected model in detail. Model performance data provided is rich and is used to fine tune or diagnose potential performance issues that may affect business goals. The data in this dashboard changes based on the selection of the project and model.

This dashboard has three panes:

1. *Project and model summary*
2. *Analysis panes*
    1. Overall drift analysis
    2. Features analysis
3. *Incoming features graph*

Project and model summary

Use the dropdown to change the project and model. The dashboard presents the following information about the project:

- **Endpoint ID** — the ID of the deployed model
- **Model** — user defined name for the model
- **Function URI** — the MLRun function to access the model
- **Model Class** — the implementation class that is used by the endpoint
- **Prediction/s** — the average number of predictions per second over a rolling 5-minute period
- **Average Latency** — the average latency over a rolling 1-hour period
- **First Request** — first request for production data
- **Last Request** — most recent request for production data

Use the *Performance* and *Overview* buttons view those dashboards.

Analysis panes

This pane has two sections: Overall Drift Analysis and Features Analysis. The Overall Drift Analysis pane provides performance statistics for the currently selected model.

- **TVD** (sum and mean)
- **Hellinger** (sum and mean)
- **KLD** (sum and mean)

The Features Analysis pane provides details of the drift analysis for each feature in the selected model. This pane includes five types of statistics:

- **Actual** (min, mean and max) — results based on actual live data stream

- **Expected** (min, mean and max) — results based on training data
- **TVD**
- **Hellinger**
- **KLD**

Incoming features graph

This graph displays the performance of the features that are in the selected model based on sampled data points from actual feature production data. The graph displays the values of the features in the model over time.

Model endpoint performance dashboard

Model endpoint performance displays performance details in graphical format.

This dashboard has five graphs:

- **Drift Measures** — the overall drift over time for each of the endpoints in the selected model
- **Average Latency** — the average latency of the model in 5 minute intervals, for 5 minutes and 1 hour rolling windows
- **Predictions/s** — the model predictions per second displayed in 5 second intervals for 5 minutes (rolling)
- **Predictions Count** — the number of predictions the model makes for 5 minutes and 1 hour rolling windows

Configuring Grafana dashboards

Verify that you have a Grafana service running in your Iguazio MLOps Platform. If you do not have a Grafana service running, see Adding Grafana Dashboards to create and configure it. When you create the service: In the **Custom Parameters** tab, **Platform data-access user** parameter, select a user with access to the `/user/pipelines` directory.

For working with Iguazio 3.0.x:

1. Make sure you have the `model-monitoring` as a Grafana data source configured in your Grafana service. If not, add it by:
    1. Open your grafana service.
    2. Navigate to **Configuration | Data Sources**.
    3. Press **Add data source**.
    4. Select the **SimpleJson** datasource and configure the following parameters.

```
Name: model-monitoring
URL: http://mlrun-api:8080/api/grafana-proxy/model-endpoints
Access: Server (default)

## Add a custom header of:
X-V3io-Session-Key: <YOUR ACCESS KEY>
```

    5. Press **Save & Test** for verification. You'll receive a confirmation with either a success or a failure message.
2. Download the following monitoring dashboards:
    - `Model Monitoring - Overview`
    - `Model Monitoring - Details`
    - `Model Monitoring - Performance`
3. Import the downloaded dashboards to your Grafana service:
    1. Navigate to your Grafana service in the Services list and press it.
    2. Press the dashboards icon in left menu.
    3. In the Dashboard Management screen, press **IMPORT**, and select one file to import. Repeat this step for each dashboard.

For working with Iguazio 3.2.x and later: Add access keys to your model-monitoring data source:

1. Open your Grafana service.
2. Navigate to **Configuration | Data Sources**.
3. Press **mlrun-model-monitoring**.
4. In Custom HTTP Headers, configure the cookie parameter. Set the value of `cookie` to:

```
session=j:{"sid":  "<YOUR ACCESS KEY>"}
```

The overview, details, and performance dashboards are in **Dashboards | Manage | private**

Note: You need to train and deploy a model to see results in the dashboards. The dashboards immediately display data if you already have a model that is trained and running with production data.

15.2 Enable model monitoring

Note: This is currently a beta feature.

To see tracking results, model monitoring needs to be enabled in each model.

To enable model monitoring, include `serving_fn.set_tracking()` in the model server.

To utilize drift measurement, supply the train set in the training step.

In this section

15.2.1 Model monitoring demo

Use the following code blocks to test and explore model monitoring.

```
# Set project name
project_name = "demo-project"
```

Deploy model servers

Use the following code to deploy a model server in the Iguazio instance.

```
import os
import pandas as pd
from sklearn.datasets import load_iris

from mlrun import import_function, get_dataitem, get_or_create_project
from mlrun.platforms import auto_mount

project = get_or_create_project(project_name, context="./")
project.set_model_monitoring_credentials(os.environ.get("V3IO_ACCESS_KEY"))

# Download the pre-trained Iris model
get_dataitem("https://s3.wasabisys.com/iguazio/models/iris/model.pkl").download("model.
↪pkl")

iris = load_iris()
train_set = pd.DataFrame(iris['data'],
                         columns=['sepal_length_cm', 'sepal_width_cm',
                                  'petal_length_cm', 'petal_width_cm'])

# Import the serving function from the Function Hub
serving_fn = import_function('hub://v2_model_server', project=project_name).apply(auto_
↪mount())
```

(continues on next page)

(continued from previous page)

```
model_name = "RandomForestClassifier"

# Log the model through the projects API so that it is available through the feature␣
↪store API
project.log_model(model_name, model_file="model.pkl", training_set=train_set)

# Add the model to the serving function's routing spec
serving_fn.add_model(model_name, model_path=f"store://models/{project_name}/{model_name}
↪:latest")

# Enable model monitoring
serving_fn.set_tracking()

# Deploy the function
serving_fn.deploy()
```

Simulating requests

Use the following code to simulate production data.

```
import json
from time import sleep
from random import choice, uniform

iris_data = iris['data'].tolist()

while True:
    data_point = choice(iris_data)
    serving_fn.invoke(f'v2/models/{model_name}/infer', json.dumps({'inputs': [data_
↪point]}))
    sleep(uniform(0.2, 1.7))
```

CHAPTER

SIXTEEN

API INDEX

CHAPTER

SEVENTEEN

API BY MODULE

MLRun is organized into the following modules. The most common functions are exposed in the *mlrun* module, which is the recommended starting point.

17.1 mlrun.frameworks

MLRun provides a quick and easy integration into your code with mlrun.frameworks: a collection of sub-modules for the most commonly used machine and deep learning frameworks, providing features such as automatic logging, model management, and distributed training.

17.1.1 mlrun.frameworks.auto_mlrun

class mlrun.frameworks.auto_mlrun.auto_mlrun.**AutoMLRun**
Bases: object

A library of automatic functions for managing models using MLRun's frameworks package.

static apply_mlrun(*model: Optional[mlrun.frameworks._common.utils.ModelType] = None*, *model_name: Optional[str] = None*, *tag: str = ''*, *model_path: Optional[str] = None*, *modules_map: Optional[Union[Dict[str, Union[None, str, List[str]]], str]] = None*, *custom_objects_map: Optional[Union[Dict[str, Union[str, List[str]]], str]] = None*, *custom_objects_directory: Optional[str] = None*, *context: Optional[*mlrun.execution.MLClientCtx*] = None*, *framework: Optional[str] = None*, *auto_log: bool = True*, ***kwargs*) → mlrun.frameworks._common.model_handler.ModelHandler

Use MLRun's 'apply_mlrun' of the detected given model's framework to wrap the framework relevant methods and gain the framework's features in MLRun. A ModelHandler initialized with the model will be returned.

Parameters

- **model** – The model to wrap. Can be loaded from the model path given as well.
- **model_name** – The model name to use for storing the model artifact. If not given will have a default name according to the framework.
- **tag** – The model's tag to log with.
- **model_path** – The model's store object path. Mandatory for evaluation (to know which model to update). If model is not provided, it will be loaded from this path.
- **modules_map** – A dictionary of all the modules required for loading the model. Each key is a path to a module and its value is the object name to import from it. All the modules

will be imported globally. If multiple objects needed to be imported from the same module a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "module1": None,  # import module1
    "module2": ["func1", "func2"],  # from module2 import func1,␣
↪func2
    "module3.sub_module": "func3",  # from module3.sub_module␣
↪import func3
}
```

If the model path given is of a store object, the modules map will be read from the logged modules map artifact of the model.

- **custom_objects_map** – A dictionary of all the custom objects required for loading the model. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "/.../custom_model.py": "MyModel",
    "/.../custom_objects.py": ["object1", "object2"]
}
```

All the paths will be accessed from the given 'custom_objects_directory', meaning each py file will be read from 'custom_objects_directory/<MAP VALUE>'. If the model path given is of a store object, the custom objects map will be read from the logged custom object map artifact of the model. Notice: The custom objects will be imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- **custom_objects_directory** – Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (will be extracted during the run before loading the model). If the model path given is of a store object, the custom objects files will be read from the logged custom object artifact of the model.
- **context** – A MLRun context.
- **auto_log** – Whether to enable auto-logging capabilities of MLRun or not. Auto logging will add default artifacts and metrics besides the one you can pass here.
- **framework** – The model's framework. If None, AutoMLRun will try to figure out the framework. From the provided model or model path. Default: None.
- **kwargs** – Additional parameters for the specific framework's 'apply_mlrun' function like metrics, callbacks and more (read the docs of the required framework to know more).

Returns The framework's model handler initialized with the given model.

static load_model(*model_path: str*, *model_name: Optional[str] = None*, *context: Optional[mlrun.execution.MLClientCtx] = None*, *modules_map: Optional[Union[Dict[str, Union[None, str, List[str]]], str]] = None*, *custom_objects_map: Optional[Union[Dict[str, Union[str, List[str]]], str]] = None*, *custom_objects_directory: Optional[str] = None*, *framework: Optional[str] = None*, ***kwargs*) → mlrun.frameworks._common.model_handler.ModelHandler

Load a model using MLRun's ModelHandler. The loaded model can be accessed from the model handler returned via model_handler.model. If the model is a store object uri (it is logged in MLRun) then the

framework will be read automatically, otherwise (for local path and urls) it must be given. The other parameters will be automatically read in case its a logged model in MLRun.

Parameters

- **model_path** – A store object path of a logged model object in MLRun.
- **model_name** – The model name to use for storing the model artifact. If not given will have a default name according to the framework.
- **modules_map** – A dictionary of all the modules required for loading the model. Each key is a path to a module and its value is the object name to import from it. All the modules will be imported globally. If multiple objects needed to be imported from the same module a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "module1": None,  # import module1
    "module2": ["func1", "func2"],  # from module2 import func1,␣
↪func2
    "module3.sub_module": "func3",  # from module3.sub_module␣
↪import func3
}
```

  If the model path given is of a store object, the modules map will be read from the logged modules map artifact of the model.

- **custom_objects_map** – A dictionary of all the custom objects required for loading the model. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "/.../custom_model.py": "MyModel",
    "/.../custom_objects.py": ["object1", "object2"]
}
```

  All the paths will be accessed from the given 'custom_objects_directory', meaning each py file will be read from 'custom_objects_directory/<MAP VALUE>'. If the model path given is of a store object, the custom objects map will be read from the logged custom object map artifact of the model. Notice: The custom objects will be imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- **custom_objects_directory** – Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (will be extracted during the run before loading the model). If the model path given is of a store object, the custom objects files will be read from the logged custom object artifact of the model.
- **context** – A MLRun context.
- **framework** – The model's framework. It must be provided for local paths or urls. If None, AutoMLRun will assume the model path is of a store uri model artifact and try to get the framework from it. Default: None.
- **kwargs** – Additional parameters for the specific framework's ModelHandler class.

Returns The model inside a MLRun model handler.

Raises **MLRunInvalidArgumentError** – In case the framework is incorrect or missing.

mlrun.frameworks.auto_mlrun.auto_mlrun.**framework_to_apply_mlrun**(*framework: str*) → Callable[[...], mlrun.frameworks._common.model_handler.Model

Get the 'apply_mlrun' shortcut function of the given framework's name.

Parameters **framework** – The framework's name.

Returns The framework's 'apply_mlrun' shortcut function.

Raises **MLRunInvalidArgumentError** – If the given framework is not supported by AutoMLRun or if it does not have an 'apply_mlrun' yet.

mlrun.frameworks.auto_mlrun.auto_mlrun.**framework_to_model_handler**(*framework: str*) → Type[mlrun.frameworks._common.model_handle

Get the ModelHandler class of the given framework's name.

Parameters **framework** – The framework's name.

Returns The framework's ModelHandler class.

Raises **MLRunInvalidArgumentError** – If the given framework is not supported by AutoMLRun.

mlrun.frameworks.auto_mlrun.auto_mlrun.**get_framework_by_class_name**(*model: mlrun.frameworks._common.utils.ModelType*) → str

Get the framework name of the given model by its class name.

Parameters **model** – The model to get its framework.

Returns The model's framework.

Raises **MLRunInvalidArgumentError** – If the given model's class name is not supported by AutoMLRun or not recognized.

mlrun.frameworks.auto_mlrun.auto_mlrun.**get_framework_by_instance**(*model: mlrun.frameworks._common.utils.ModelType*) → str

Get the framework name of the given model by its instance.

Parameters **model** – The model to get his framework.

Returns The model's framework.

Raises **MLRunInvalidArgumentError** – If the given model type is not supported by AutoMLRun or not recognized.

17.1.2 mlrun.frameworks.tf_keras

mlrun.frameworks.tf_keras.**apply_mlrun**(*model: Optional[tensorflow.keras.Model] = None*, *model_name: Optional[str] = None*, *tag: str = ''*, *model_path: Optional[str] = None*, *model_format: str = 'SavedModel'*, *save_traces: bool = False*, *modules_map: Optional[Union[Dict[str, Union[None, str, List[str]]], str]] = None*, *custom_objects_map: Optional[Union[Dict[str, Union[str, List[str]]], str]] = None*, *custom_objects_directory: Optional[str] = None*, *context: Optional[mlrun.execution.MLClientCtx] = None*, *auto_log: bool = True*, *tensorboard_directory: Optional[str] = None*, *mlrun_callback_kwargs: Optional[Dict[str, Any]] = None*, *tensorboard_callback_kwargs: Optional[Dict[str, Any]] = None*, *use_horovod: Optional[bool] = None*, ***kwargs*) → mlrun.frameworks.tf_keras.model_handler.TFKerasModelHandler

Wrap the given model with MLRun's interface providing it with mlrun's additional features.

Parameters

- **model** – The model to wrap. Can be loaded from the model path given as well.
- **model_name** – The model name to use for storing the model artifact. If not given, the tf.keras.Model.name will be used.
- **tag** – The model's tag to log with.
- **model_path** – The model's store object path. Mandatory for evaluation (to know which model to update). If model is not provided, it will be loaded from this path.
- **model_format** – The format to use for saving and loading the model. Should be passed as a member of the class 'ModelFormats'. Default: 'ModelFormats.SAVED_MODEL'.
- **save_traces** – Whether or not to use functions saving (only available for the 'SavedModel' format) for loading the model later without the custom objects dictionary. Only from tensorflow version >= 2.4.0. Using this setting will increase the model saving size.
- **modules_map** – A dictionary of all the modules required for loading the model. Each key is a path to a module and its value is the object name to import from it. All the modules will be imported globally. If multiple objects needed to be imported from the same module a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "module1": None,  # import module1
    "module2": ["func1", "func2"],  # from module2 import func1,␣
↪func2
    "module3.sub_module": "func3",  # from module3.sub_module import␣
↪func3
}
```

  If the model path given is of a store object, the modules map will be read from the logged modules map artifact of the model.

- **custom_objects_map** – A dictionary of all the custom objects required for loading the model. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "/.../custom_optimizer.py": "optimizer",
    "/.../custom_layers.py": ["layer1", "layer2"]
}
```

  All the paths will be accessed from the given 'custom_objects_directory', meaning each py file will be read from 'custom_objects_directory/<MAP VALUE>'. If the model path given is of a store object, the custom objects map will be read from the logged custom object map artifact of the model. Notice: The custom objects will be imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- **custom_objects_directory** – Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (will be extracted during the run before loading the model). If the model path given is of a store object, the custom objects files will be read from the logged custom object artifact of the model.
- **context** – MLRun context to work with. If no context is given it will be retrieved via 'mlrun.get_or_create_ctx(None)'

- **auto_log** – Whether or not to apply MLRun's auto logging on the model. Default: True.
- **tensorboard_directory** – If context is not given, or if wished to set the directory even with context, this will be the output for the event logs of tensorboard. If not given, the 'tensorboard_dir' parameter will be tried to be taken from the provided context. If not found in the context, the default tensorboard output directory will be: /User/.tensorboard/<PROJECT_NAME> or if working on local, the set artifacts path.
- **mlrun_callback_kwargs** – Key word arguments for the MLRun callback. For further information see the documentation of the class 'MLRunLoggingCallback'. Note that both 'context' and 'auto_log' parameters are already given here.
- **tensorboard_callback_kwargs** – Key word arguments for the tensorboard callback. For further information see the documentation of the class 'TensorboardLoggingCallback'. Note that both 'context' and 'auto_log' parameters are already given here.
- **use_horovod** – Whether or not to use horovod - a distributed training framework. Default: None, meaning it will be read from context if available and if not - False.

Returns The model with MLRun's interface.

17.1.3 mlrun.frameworks.pytorch

mlrun.frameworks.pytorch.**evaluate**(*model_path: str*, *dataset: torch.utils.data.DataLoader*, *model: Optional[torch.nn.Module] = None*, *loss_function: Optional[torch.nn.Module] = None*, *metric_functions: Optional[List[Union[Callable[[torch.Tensor, torch.Tensor], Union[int, float, numpy.ndarray, torch.Tensor]], torch.nn.Module]]] = None*, *iterations: Optional[int] = None*, *callbacks_list: Optional[List[mlrun.frameworks.pytorch.callbacks.callback.Callback]] = None*, *use_cuda: bool = True*, *use_horovod: bool = False*, *auto_log: bool = True*, *model_name: Optional[str] = None*, *modules_map: Optional[Union[Dict[str, Union[None, str, List[str]]], str]] = None*, *custom_objects_map: Optional[Union[Dict[str, Union[str, List[str]]], str]] = None*, *custom_objects_directory: Optional[str] = None*, *mlrun_callback_kwargs: Optional[Dict[str, Any]] = None*, *context: Optional[*mlrun.execution.MLClientCtx*] = None*) → Tuple[mlrun.frameworks.pytorch.model_handler.PyTorchModelHandler, List[Union[int, float, numpy.ndarray, torch.Tensor]]]

Use MLRun's PyTorch interface to evaluate the model with the given parameters. For more information and further options regarding the auto logging, see 'PyTorchMLRunInterface' documentation. Notice for auto-logging: In order to log the model to MLRun, its class (torch.Module) must be in the custom objects map or the modules map.

Parameters

- **model_path** – The model's store object path. Mandatory for evaluation (to know which model to update).
- **dataset** – A data loader for the validation process.
- **model** – The model to evaluate. IF None, the model will be loaded from the given store model path.
- **loss_function** – The loss function to use during training.
- **metric_functions** – The metrics to use on training and validation.

- **iterations** – Amount of iterations (batches) to perform on the dataset. If 'None' the entire dataset will be used.
- **callbacks_list** – The callbacks to use on this run.
- **use_cuda** – Whether or not to use cuda. Only relevant if cuda is available. Default: True.
- **use_horovod** – Whether or not to use horovod - a distributed training framework. Default: False.
- **auto_log** – Whether or not to apply auto-logging to MLRun. Default: True.
- **model_name** – The model name to use for storing the model artifact. If not given, the model's class name will be used.
- **modules_map** – A dictionary of all the modules required for loading the model. Each key is a path to a module and its value is the object name to import from it. All the modules will be imported globally. If multiple objects needed to be imported from the same module a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "module1": None,  # import module1
    "module2": ["func1", "func2"],  # from module2 import func1,␣
↪func2
    "module3.sub_module": "func3",  # from module3.sub_module import␣
↪func3
}
```

  If the model path given is of a store object, the modules map will be read from the logged modules map artifact of the model.

- **custom_objects_map** – A dictionary of all the custom objects required for loading the model. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "/.../custom_optimizer.py": "optimizer",
    "/.../custom_layers.py": ["layer1", "layer2"]
}
```

  All the paths will be accessed from the given 'custom_objects_directory', meaning each py file will be read from 'custom_objects_directory/<MAP VALUE>'. If the model path given is of a store object, the custom objects map will be read from the logged custom object map artifact of the model. Notice: The custom objects will be imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- **custom_objects_directory** – Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (will be extracted during the run before loading the model). If the model path given is of a store object, the custom objects files will be read from the logged custom object artifact of the model.
- **mlrun_callback_kwargs** – Key word arguments for the MLRun callback. For further information see the documentation of the class 'MLRunLoggingCallback'. Note that both 'context', 'custom_objects' and 'auto_log' parameters are already given here.
- **context** – The context to use for the logs.

Returns A tuple of: [0] = Initialized model handler with the evaluated model. [1] = The evaluation metrics results list.

mlrun.frameworks.pytorch.**train**(*model: torch.nn.Module*, *training_set: torch.utils.data.DataLoader*, *loss_function: torch.nn.Module*, *optimizer: torch.optim.Optimizer*, *validation_set: Optional[torch.utils.data.DataLoader] = None*, *metric_functions: Optional[List[Union[Callable[[torch.Tensor, torch.Tensor], Union[int, float, numpy.ndarray, torch.Tensor]], torch.nn.Module]]] = None*, *scheduler=None*, *scheduler_step_frequency: Union[int, float, str] = 'epoch'*, *epochs: int = 1*, *training_iterations: Optional[int] = None*, *validation_iterations: Optional[int] = None*, *callbacks_list: Optional[List[mlrun.frameworks.pytorch.callbacks.callback.Callback]] = None*, *use_cuda: bool = True*, *use_horovod: Optional[bool] = None*, *auto_log: bool = True*, *model_name: Optional[str] = None*, *modules_map: Optional[Union[Dict[str, Union[None, str, List[str]]], str]] = None*, *custom_objects_map: Optional[Union[Dict[str, Union[str, List[str]]], str]] = None*, *custom_objects_directory: Optional[str] = None*, *tensorboard_directory: Optional[str] = None*, *mlrun_callback_kwargs: Optional[Dict[str, Any]] = None*, *tensorboard_callback_kwargs: Optional[Dict[str, Any]] = None*, *context: Optional[mlrun.execution.MLClientCtx] = None*) → mlrun.frameworks.pytorch.model_handler.PyTorchModelHandler

Use MLRun's PyTorch interface to train the model with the given parameters. For more information and further options regarding the auto logging, see 'PyTorchMLRunInterface' documentation. Notice for auto-logging: In order to log the model to MLRun, its class (torch.Module) must be in the custom objects map or the modules map.

Parameters

- **model** – The model to train.
- **training_set** – A data loader for the training process.
- **loss_function** – The loss function to use during training.
- **optimizer** – The optimizer to use during the training.
- **validation_set** – A data loader for the validation process.
- **metric_functions** – The metrics to use on training and validation.
- **scheduler** – Scheduler to use on the optimizer at the end of each epoch. The scheduler must have a 'step' method with no input.
- **scheduler_step_frequency** – The frequency in which to step the given scheduler. Can be equal to one of the strings 'epoch' (for at the end of every epoch) and 'batch' (for at the end of every batch), or an integer that specify per how many iterations to step or a float percentage (0.0 < x < 1.0) for per x / iterations to step. Default: 'epoch'.
- **epochs** – Amount of epochs to perform. Default: a single epoch.
- **training_iterations** – Amount of iterations (batches) to perform on each epoch's training. If 'None' the entire training set will be used.
- **validation_iterations** – Amount of iterations (batches) to perform on each epoch's validation. If 'None' the entire validation set will be used.
- **callbacks_list** – The callbacks to use on this run.
- **use_cuda** – Whether or not to use cuda. Only relevant if cuda is available. Default: True.

- **use_horovod** – Whether or not to use horovod - a distributed training framework. Default: False.
- **auto_log** – Whether or not to apply auto-logging (to both MLRun and Tensorboard). Default: True. IF True, the custom objects are not optional.
- **model_name** – The model name to use for storing the model artifact. If not given, the model's class name will be used.
- **modules_map** – A dictionary of all the modules required for loading the model. Each key is a path to a module and its value is the object name to import from it. All the modules will be imported globally. If multiple objects needed to be imported from the same module a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "module1": None,  # import module1
    "module2": ["func1", "func2"],  # from module2 import func1,␣
↪func2
    "module3.sub_module": "func3",  # from module3.sub_module import␣
↪func3
}
```

  If the model path given is of a store object, the modules map will be read from the logged modules map artifact of the model.

- **custom_objects_map** – A dictionary of all the custom objects required for loading the model. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "/.../custom_optimizer.py": "optimizer",
    "/.../custom_layers.py": ["layer1", "layer2"]
}
```

  All the paths will be accessed from the given 'custom_objects_directory', meaning each py file will be read from 'custom_objects_directory/<MAP VALUE>'. If the model path given is of a store object, the custom objects map will be read from the logged custom object map artifact of the model. Notice: The custom objects will be imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- **custom_objects_directory** – Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (will be extracted during the run before loading the model). If the model path given is of a store object, the custom objects files will be read from the logged custom object artifact of the model.
- **tensorboard_directory** – If context is not given, or if wished to set the directory even with context, this will be the output for the event logs of tensorboard. If not given, the 'tensorboard_dir' parameter will be tried to be taken from the provided context. If not found in the context, the default tensorboard output directory will be: /User/.tensorboard/<PROJECT_NAME> or if working on local, the set artifacts path.
- **mlrun_callback_kwargs** – Key word arguments for the MLRun callback. For further information see the documentation of the class 'MLRunLoggingCallback'. Note that both 'context', 'custom_objects' and 'auto_log' parameters are already given here.
- **tensorboard_callback_kwargs** – Key word arguments for the tensorboard callback. For

further information see the documentation of the class 'TensorboardLoggingCallback'. Note that both 'context' and 'auto_log' parameters are already given here.

- **context** – The context to use for the logs.

Returns A model handler with the provided model and parameters.

Raises **ValueError** – If 'auto_log' is set to True and one all of the custom objects or modules parameters given is None.

17.1.4 mlrun.frameworks.sklearn

mlrun.frameworks.sklearn.**apply_mlrun**(*model: Union[sklearn.base.BaseEstimator, sklearn.base.BiclusterMixin, sklearn.base.ClassifierMixin, sklearn.base.ClusterMixin, sklearn.base.DensityMixin, sklearn.base.RegressorMixin, sklearn.base.TransformerMixin] = None*, *model_name: str = 'model'*, *tag: str = ''*, *model_path: str = None*, *modules_map: Union[Dict[str, Union[None, str, List[str]]], str] = None*, *custom_objects_map: Union[Dict[str, Union[str, List[str]]], str] = None*, *custom_objects_directory: str = None*, *context: mlrun.execution.MLClientCtx = None*, *artifacts: Union[List[mlrun.frameworks._ml_common.plan.MLPlan], List[str], Dict[str, dict]] = None*, *metrics: Union[List[mlrun.frameworks.sklearn.metric.Metric], List[Union[Tuple[Union[Callable, str], dict], Callable, str]], Dict[str, Union[Tuple[Union[Callable, str], dict], Callable, str]]] = None*, *x_test: Union[list, tuple, dict, numpy.ndarray, pandas.core.frame.DataFrame, pandas.core.series.Series, scipy.sparse.base.spmatrix] = None*, *y_test: Union[list, tuple, dict, numpy.ndarray, pandas.core.frame.DataFrame, pandas.core.series.Series, scipy.sparse.base.spmatrix] = None*, *sample_set: Union[list, tuple, dict, numpy.ndarray, pandas.core.frame.DataFrame, pandas.core.series.Series, scipy.sparse.base.spmatrix, mlrun.datastore.base.DataItem, str] = None*, *y_columns: Union[List[str], List[int]] = None*, *feature_vector: str = None*, *feature_weights: List[float] = None*, *labels: Dict[str, Union[str, int, float]] = None*, *parameters: Dict[str, Union[str, int, float]] = None*, *extra_data: Dict[str, Union[str, bytes, mlrun.artifacts.base.Artifact, mlrun.datastore.base.DataItem]] = None*, *auto_log: bool = True*, ***kwargs*) → mlrun.frameworks.sklearn.model_handler.SKLearnModelHandler

Wrap the given model with MLRun's interface providing it with mlrun's additional features.

Parameters

- **model** – The model to wrap. Can be loaded from the model path given as well.
- **model_name** – The model name to use for storing the model artifact. Default: "model".
- **tag** – The model's tag to log with.
- **model_path** – The model's store object path. Mandatory for evaluation (to know which model to update). If model is not provided, it will be loaded from this path.
- **modules_map** – A dictionary of all the modules required for loading the model. Each key is a path to a module and its value is the object name to import from it. All the modules will be imported globally. If multiple objects needed to be imported from the same module a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "module1": None,  # import module1
    "module2": ["func1", "func2"],  # from module2 import func1,␣
↪func2
    "module3.sub_module": "func3",  # from module3.sub_module import␣
↪func3
}
```

If the model path given is of a store object, the modules map will be read from the logged modules map artifact of the model.

- **custom_objects_map** – A dictionary of all the custom objects required for loading the model. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "/.../custom_model.py": "MyModel",
    "/.../custom_objects.py": ["object1", "object2"]
}
```

All the paths will be accessed from the given 'custom_objects_directory', meaning each py file will be read from 'custom_objects_directory/<MAP VALUE>'. If the model path given is of a store object, the custom objects map will be read from the logged custom object map artifact of the model. Notice: The custom objects will be imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- **custom_objects_directory** – Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (will be extracted during the run before loading the model). If the model path given is of a store object, the custom objects files will be read from the logged custom object artifact of the model.
- **context** – MLRun context to work with. If no context is given it will be retrieved via 'mlrun.get_or_create_ctx(None)'
- **artifacts** – A list of artifacts plans to produce during the run.
- **metrics** – A list of metrics to calculate during the run.
- **x_test** – The validation data for producing and calculating artifacts and metrics post training. Without this, validation will not be performed.
- **y_test** – The test data ground truth for producing and calculating artifacts and metrics post training or post predict / predict_proba.
- **sample_set** – A sample set of inputs for the model for logging its stats along the model in favour of model monitoring. If not given the 'x_train' will be used by default.
- **y_columns** – List of names of all the columns in the ground truth labels in case its a pd.DataFrame or a list of integers in case the dataset is a np.ndarray. If not given 'y_train' is given then the labels / indices in it will be used by default.
- **feature_vector** – Feature store feature vector uri (store://feature-vectors/<project>/<name>[:tag])
- **feature_weights** – List of feature weights, one per input column.
- **labels** – Labels to log with the model.

- **parameters** – Parameters to log with the model.
- **extra_data** – Extra data to log with the model.
- **auto_log** – Whether to apply MLRun's auto logging on the model. Auto logging will add the default artifacts and metrics to the lists of artifacts and metrics. Default: True.

Returns The model handler initialized with the provided model.

17.1.5 mlrun.frameworks.xgboost

mlrun.frameworks.xgboost.**apply_mlrun**(*model: xgboost.XGBModel = None*, *model_name: str = 'model'*, *tag: str = ''*, *model_path: str = None*, *modules_map: Union[Dict[str, Union[None, str, List[str]]], str] = None*, *custom_objects_map: Union[Dict[str, Union[str, List[str]]], str] = None*, *custom_objects_directory: str = None*, *context: mlrun.execution.MLClientCtx = None*, *artifacts: Union[List[mlrun.frameworks._ml_common.plan.MLPlan], List[str], Dict[str, dict]] = None*, *metrics: Union[List[mlrun.frameworks.sklearn.metric.Metric], List[Union[Tuple[Union[Callable, str], dict], Callable, str]], Dict[str, Union[Tuple[Union[Callable, str], dict], Callable, str]]] = None*, *x_test: Union[list, tuple, dict, numpy.ndarray, pandas.core.frame.DataFrame, pandas.core.series.Series, scipy.sparse.base.spmatrix, xgboost.DMatrix] = None*, *y_test: Union[list, tuple, dict, numpy.ndarray, pandas.core.frame.DataFrame, pandas.core.series.Series, scipy.sparse.base.spmatrix, xgboost.DMatrix] = None*, *sample_set: Union[list, tuple, dict, numpy.ndarray, pandas.core.frame.DataFrame, pandas.core.series.Series, scipy.sparse.base.spmatrix, xgboost.DMatrix, mlrun.datastore.base.DataItem, str] = None*, *y_columns: Union[List[str], List[int]] = None*, *feature_vector: str = None*, *feature_weights: List[float] = None*, *labels: Dict[str, Union[str, int, float]] = None*, *parameters: Dict[str, Union[str, int, float]] = None*, *extra_data: Dict[str, Union[str, bytes, mlrun.artifacts.base.Artifact, mlrun.datastore.base.DataItem]] = None*, *auto_log: bool = True*, ***kwargs*) → mlrun.frameworks.xgboost.model_handler.XGBoostModelHandler

Wrap the given model with MLRun's interface providing it with mlrun's additional features.

Parameters

- **model** – The model to wrap. Can be loaded from the model path given as well.
- **model_name** – The model name to use for storing the model artifact. Default: "model".
- **tag** – The model's tag to log with.
- **model_path** – The model's store object path. Mandatory for evaluation (to know which model to update). If model is not provided, it will be loaded from this path.
- **modules_map** – A dictionary of all the modules required for loading the model. Each key is a path to a module and its value is the object name to import from it. All the modules will be imported globally. If multiple objects needed to be imported from the same module a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "module1": None,  # import module1
    "module2": ["func1", "func2"],  # from module2 import func1,␣
↪func2
    "module3.sub_module": "func3",  # from module3.sub_module import␣
↪func3
}
```

If the model path given is of a store object, the modules map will be read from the logged modules map artifact of the model.

- **custom_objects_map** – A dictionary of all the custom objects required for loading the model. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "/.../custom_model.py": "MyModel",
    "/.../custom_objects.py": ["object1", "object2"]
}
```

All the paths will be accessed from the given 'custom_objects_directory', meaning each py file will be read from 'custom_objects_directory/<MAP VALUE>'. If the model path given is of a store object, the custom objects map will be read from the logged custom object map artifact of the model. Notice: The custom objects will be imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- **custom_objects_directory** – Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (will be extracted during the run before loading the model). If the model path given is of a store object, the custom objects files will be read from the logged custom object artifact of the model.
- **context** – MLRun context to work with. If no context is given it will be retrieved via 'mlrun.get_or_create_ctx(None)'
- **artifacts** – A list of artifacts plans to produce during the run.
- **metrics** – A list of metrics to calculate during the run.
- **x_test** – The validation data for producing and calculating artifacts and metrics post training. Without this, validation will not be performed.
- **y_test** – The test data ground truth for producing and calculating artifacts and metrics post training or post predict / predict_proba.
- **sample_set** – A sample set of inputs for the model for logging its stats along the model in favour of model monitoring.
- **y_columns** – List of names of all the columns in the ground truth labels in case its a pd.DataFrame or a list of integers in case the dataset is a np.ndarray. If not given but 'y_train' / 'y_test' is given then the labels / indices in it will be used by default.
- **feature_vector** – Feature store feature vector uri (store://feature-vectors/<project>/<name>[:tag])
- **feature_weights** – List of feature weights, one per input column.
- **labels** – Labels to log with the model.

- **parameters** – Parameters to log with the model.
- **extra_data** – Extra data to log with the model.
- **auto_log** – Whether to apply MLRun's auto logging on the model. Auto logging will add the default artifacts and metrics to the lists of artifacts and metrics. Default: True.

Returns The model handler initialized with the provided model.

17.1.6 mlrun.frameworks.lgbm

mlrun.frameworks.lgbm.**apply_mlrun**(*model: Union[lightgbm.LGBMModel, lightgbm.Booster] = None*, *model_name: str = 'model'*, *tag: str = ''*, *model_path: str = None*, *modules_map: Union[Dict[str, Union[None, str, List[str]]], str] = None*, *custom_objects_map: Union[Dict[str, Union[str, List[str]]], str] = None*, *custom_objects_directory: str = None*, *context: mlrun.execution.MLClientCtx = None*, *model_format: str = 'pkl'*, *artifacts: Union[List[mlrun.frameworks._ml_common.plan.MLPlan], List[str], Dict[str, dict]] = None*, *metrics: Union[List[mlrun.frameworks.sklearn.metric.Metric], List[Union[Tuple[Union[Callable, str], dict], Callable, str]], Dict[str, Union[Tuple[Union[Callable, str], dict], Callable, str]]] = None*, *x_test: Union[list, tuple, dict, numpy.ndarray, pandas.core.frame.DataFrame, pandas.core.series.Series, scipy.sparse.base.spmatrix, lightgbm.Dataset] = None*, *y_test: Union[list, tuple, dict, numpy.ndarray, pandas.core.frame.DataFrame, pandas.core.series.Series, scipy.sparse.base.spmatrix, lightgbm.Dataset] = None*, *sample_set: Union[list, tuple, dict, numpy.ndarray, pandas.core.frame.DataFrame, pandas.core.series.Series, scipy.sparse.base.spmatrix, lightgbm.Dataset, mlrun.datastore.base.DataItem, str] = None*, *y_columns: Union[List[str], List[int]] = None*, *feature_vector: str = None*, *feature_weights: List[float] = None*, *labels: Dict[str, Union[str, int, float]] = None*, *parameters: Dict[str, Union[str, int, float]] = None*, *extra_data: Dict[str, Union[str, bytes, mlrun.artifacts.base.Artifact, mlrun.datastore.base.DataItem]] = None*, *auto_log: bool = True*, *mlrun_logging_callback_kwargs: Dict[str, Any] = None*, ***kwargs*) → Optional[mlrun.frameworks.lgbm.model_handler.LGBMModelHandler]

Apply MLRun's interface on top of LightGBM by wrapping the module itself or the given model, providing both with MLRun's quality of life features.

Parameters

- **model** – The model to wrap. Can be loaded from the model path given as well.
- **model_name** – The model name to use for storing the model artifact. Default: "model".
- **tag** – The model's tag to log with.
- **model_path** – The model's store object path. Mandatory for evaluation (to know which model to update). If model is not provided, it will be loaded from this path.
- **modules_map** – A dictionary of all the modules required for loading the model. Each key is a path to a module and its value is the object name to import from it. All the modules will be imported globally. If multiple objects needed to be imported from the same module a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "module1": None,  # import module1
    "module2": ["func1", "func2"],  # from module2 import func1,␣
↪func2
    "module3.sub_module": "func3",  # from module3.sub_module import␣
↪func3
}
```

If the model path given is of a store object, the modules map will be read from the logged modules map artifact of the model.

- **custom_objects_map** – A dictionary of all the custom objects required for loading the model. Each key is a path to a python file and its value is the custom object name to import from it. If multiple objects needed to be imported from the same py file a list can be given. The map can be passed as a path to a json file as well. For example:

```
{
    "/.../custom_model.py": "MyModel",
    "/.../custom_objects.py": ["object1", "object2"]
}
```

All the paths will be accessed from the given 'custom_objects_directory', meaning each py file will be read from 'custom_objects_directory/<MAP VALUE>'. If the model path given is of a store object, the custom objects map will be read from the logged custom object map artifact of the model. Notice: The custom objects will be imported in the order they came in this dictionary (or json). If a custom object is depended on another, make sure to put it below the one it relies on.

- **custom_objects_directory** – Path to the directory with all the python files required for the custom objects. Can be passed as a zip file as well (will be extracted during the run before loading the model). If the model path given is of a store object, the custom objects files will be read from the logged custom object artifact of the model.
- **context** – MLRun context to work with. If no context is given it will be retrieved via 'mlrun.get_or_create_ctx(None)'
- **artifacts** – A list of artifacts plans to produce during the run.
- **metrics** – A list of metrics to calculate during the run.
- **x_test** – The validation data for producing and calculating artifacts and metrics post training. Without this, validation will not be performed.
- **y_test** – The test data ground truth for producing and calculating artifacts and metrics post training or post predict / predict_proba.
- **sample_set** – A sample set of inputs for the model for logging its stats along the model in favour of model monitoring.
- **y_columns** – List of names of all the columns in the ground truth labels in case its a pd.DataFrame or a list of integers in case the dataset is a np.ndarray. If not given but 'y_train' / 'y_test' is given then the labels / indices in it will be used by default.
- **feature_vector** – Feature store feature vector uri (store://feature-vectors/<project>/<name>[:tag])
- **feature_weights** – List of feature weights, one per input column.
- **labels** – Labels to log with the model.

- **parameters** – Parameters to log with the model.
- **extra_data** – Extra data to log with the model.
- **auto_log** – Whether to apply MLRun's auto logging on the model. Auto logging will add the default artifacts and metrics to the lists of artifacts and metrics. Default: True.
- **mlrun_logging_callback_kwargs** – Key word arguments for the MLRun callback. For further information see the documentation of the class 'MLRunLoggingCallback'. Note that 'context' is already given here.

Returns If a model was provided via *model* or *model_path* the model handler initialized with the provided model will be returned. Otherwise, None.

17.2 mlrun

class mlrun.**ArtifactType**(*value*)

Possible artifact types to log using the MLRun *context* decorator.

mlrun.**code_to_function**(*name: str = ''*, *project: str = ''*, *tag: str = ''*, *filename: str = ''*, *handler: str = ''*, *kind: str = ''*, *image: Optional[str] = None*, *code_output: str = ''*, *embed_code: bool = True*, *description: str = ''*, *requirements: Optional[Union[str, List[str]]] = None*, *categories: Optional[List[str]] = None*, *labels: Optional[Dict[str, str]] = None*, *with_doc: bool = True*, *ignored_tags=None*) → Union[mlrun.runtimes.mpijob.v1alpha1.MpiRuntimeV1Alpha1, mlrun.runtimes.mpijob.v1.MpiRuntimeV1, *mlrun.runtimes.function.RemoteRuntime*, *mlrun.runtimes.serving.ServingRuntime*, *mlrun.runtimes.daskjob.DaskCluster*, *mlrun.runtimes.kubejob.KubejobRuntime*, *mlrun.runtimes.local.LocalRuntime*, mlrun.runtimes.sparkjob.spark2job.Spark2Runtime, mlrun.runtimes.sparkjob.spark3job.Spark3Runtime, *mlrun.runtimes.remotesparkjob.RemoteSparkRuntime*]

Convenience function to insert code and configure an mlrun runtime.

Easiest way to construct a runtime type object. Provides the most often used configuration options for all runtimes as parameters.

Instantiated runtimes are considered 'functions' in mlrun, but they are anything from nuclio functions to generic kubernetes pods to spark jobs. Functions are meant to be focused, and as such limited in scope and size. Typically a function can be expressed in a single python module with added support from custom docker images and commands for the environment. The returned runtime object can be further configured if more customization is required.

One of the most important parameters is 'kind'. This is what is used to specify the chosen runtimes. The options are:

- local: execute a local python or shell script
- job: insert the code into a Kubernetes pod and execute it
- nuclio: insert the code into a real-time serverless nuclio function
- serving: insert code into orchestrated nuclio function(s) forming a DAG
- dask: run the specified python code / script as Dask Distributed job
- mpijob: run distributed Horovod jobs over the MPI job operator
- spark: run distributed Spark job using Spark Kubernetes Operator
- remote-spark: run distributed Spark job on remote Spark service

Learn more about function runtimes here: https://docs.mlrun.org/en/latest/runtimes/functions.html#function-runtimes

Parameters

- **name** – function name, typically best to use hyphen-case
- **project** – project used to namespace the function, defaults to 'default'
- **tag** – function tag to track multiple versions of the same function, defaults to 'latest'
- **filename** – path to .py/.ipynb file, defaults to current jupyter notebook
- **handler** – The default function handler to call for the job or nuclio function, in batch functions (job, mpijob, ..) the handler can also be specified in the *.run()* command, when not specified the entire file will be executed (as main). for nuclio functions the handler is in the form of module:function, defaults to 'main:handler'
- **kind** – function runtime type string - nuclio, job, etc. (see docstring for all options)
- **image** – base docker image to use for building the function container, defaults to None
- **code_output** – specify '.' to generate python module from the current jupyter notebook
- **embed_code** – indicates whether or not to inject the code directly into the function runtime spec, defaults to True
- **description** – short function description, defaults to ''
- **requirements** – list of python packages or pip requirements file path, defaults to None
- **categories** – list of categories for mlrun Function Hub, defaults to None
- **labels** – immutable name/value pairs to tag the function with useful metadata, defaults to None
- **with_doc** – indicates whether to document the function parameters, defaults to True
- **ignored_tags** – notebook cells to ignore when converting notebooks to py code (separated by ';')

Returns pre-configured function object from a mlrun runtime class

example:

```
import mlrun

# create job function object from notebook code and add doc/metadata
fn = mlrun.code_to_function("file_utils", kind="job",
                            handler="open_archive", image="mlrun/mlrun",
                            description = "this function opens a zip archive into a␣
↪local/mounted folder",
                            categories = ["fileutils"],
                            labels = {"author": "me"})
```

example:

```
import mlrun
from pathlib import Path

# create file
Path("mover.py").touch()
```

(continues on next page)

(continued from previous page)

```
# create nuclio function object from python module call mover.py
fn = mlrun.code_to_function("nuclio-mover", kind="nuclio",
                            filename="mover.py", image="python:3.7",
                            description = "this function moves files from one␣
↪system to another",
                            requirements = ["pandas"],
                            labels = {"author": "me"})
```

mlrun.**get_secret_or_env**(*key: str*, *secret_provider: Optional[Union[Dict, mlrun.secrets.SecretsStore, Callable]] = None*, *default: Optional[str] = None*) → str

Retrieve value of a secret, either from a user-provided secret store, or from environment variables. The function will retrieve a secret value, attempting to find it according to the following order:

1. If *secret_provider* was provided, will attempt to retrieve the secret from it
2. If an MLRun *SecretsStore* was provided, query it for the secret key
3. An environment variable with the same key
4. An MLRun-generated env. variable, mounted from a project secret (to be used in MLRun runtimes)
5. The default value

Example:

```
secrets = { "KEY1": "VALUE1" }
secret = get_secret_or_env("KEY1", secret_provider=secrets)

# Using a function to retrieve a secret
def my_secret_provider(key):
    # some internal logic to retrieve secret
    return value

secret = get_secret_or_env("KEY1", secret_provider=my_secret_provider, default="TOO-
↪MANY-SECRETS")
```

Parameters

- **key** – Secret key to look for
- **secret_provider** – Dictionary, callable or *SecretsStore* to extract the secret value from. If using a callable, it must use the signature *callable(key:str)*
- **default** – Default value to return if secret was not available through any other means

Returns The secret value if found in any of the sources, or *default* if provided.

mlrun.**get_version**()

get current mlrun version

mlrun.**handler**(*labels: Optional[Dict[str, str]] = None*, *outputs: Optional[List[Optional[Union[Tuple[str, mlrun.run.ArtifactType], Tuple[str, str], Tuple[str, mlrun.run.ArtifactType, Dict[str, Any]], Tuple[str, str, Dict[str, Any]], str]]]] = None*, *inputs: Union[bool, Dict[str, Type]] = True*)

MLRun's handler is a decorator to wrap a function and enable setting labels, automatic *mlrun.DataItem* parsing and outputs logging.

Parameters

- **labels** – Labels to add to the run. Expecting a dictionary with the labels names as keys. Default: None.
- **outputs** – Logging configurations for the function's returned values. Expecting a list of tuples and None values:
  - **str - A string in the format of '{key}:{artifact_type}'. If a string was given without ':' it will** indicate the key and the artifact type will be according to the returned value type.
  - tuple - A tuple of:
    - [0]: str - The key (name) of the artifact to use for the logged output.
    - [1]: Union[*ArtifactType*, str] = "result" - An *ArtifactType* enum or an equivalent string, that indicates how to log the returned value. The artifact types can be one of:
      - DATASET = "dataset"
      - DIRECTORY = "directory"
      - FILE = "file"
      - OBJECT = "object"
      - PLOT = "plot"
      - RESULT = "result".
    - [2]: Optional[Dict[str, Any]] - A keyword arguments dictionary with the properties to pass to the relevant logging function (one of *context.log_artifact*, *context.log_result*, *context.log_dataset*).
  - None - Do not log the output.

  The list length must be equal to the total amount of returned values from the function. Default is None - meaning no outputs will be logged.
- **inputs** – Parsing configurations for the arguments passed as inputs via the *run* method of an MLRun function. Can be passed as a boolean value or a dictionary:
  - **True - Parse all found inputs to the assigned type hint in the function's signature. If there is no** type hint assigned, the value will remain an *mlrun.DataItem*.
  - False - Do not parse inputs, leaving the inputs as *mlrun.DataItem*.
  - **Dict[str, Type] - A dictionary with argument name as key and the expected type to parse the** *mlrun.DataItem* to.

  Default: True.

Example:

```
import mlrun

@mlrun.handler(outputs=["my_array", None, "my_multiplier"])
def my_handler(array: np.ndarray, m: int):
    array = array * m
    m += 1
    return array, "I won't be logged", m

>>> mlrun_function = mlrun.code_to_function("my_code.py", kind="job")
>>> run_object = mlrun_function.run(
...     handler="my_handler",
```

(continues on next page)

(continued from previous page)

```
...     inputs={"array": "store://my_array_Artifact"},
...     params={"m": 2}
... )
>>> run_object.outputs
{'my_multiplier': 3, 'my_array': 'store://...'}
```

mlrun.**import_function**(*url=''*, *secrets=None*, *db=''*, *project=None*, *new_name=None*)

Create function object from DB or local/remote YAML file

Functions can be imported from function repositories (mlrun Function Hub (formerly Marketplace) or local db), or be read from a remote URL (http(s), s3, git, v3io, ..) containing the function YAML

special URLs:

```
function hub: hub://{name}[:{tag}]
local mlrun db:       db://{project-name}/{name}[:{tag}]
```

examples:

```
function = mlrun.import_function("hub://sklearn_classifier")
function = mlrun.import_function("./func.yaml")
function = mlrun.import_function("https://raw.githubusercontent.com/org/repo/func.
↪yaml")
```

Parameters

- **url** – path/url to Function Hub, db or function YAML file
- **secrets** – optional, credentials dict for DB or URL (s3, v3io, ...)
- **db** – optional, mlrun api/db path
- **project** – optional, target project for the function
- **new_name** – optional, override the imported function name

Returns function object

mlrun.**set_environment**(*api_path: Optional[str] = None*, *artifact_path: str = ''*, *project: str = ''*, *access_key: Optional[str] = None*, *user_project=False*, *username: Optional[str] = None*, *env_file: Optional[str] = None*, *mock_functions: Optional[str] = None*)

set and test default config for: api path, artifact_path and project

this function will try and read the configuration from the environment/api and merge it with the user provided project name, artifacts path or api path/access_key. it returns the configured artifacts path, this can be used to define sub paths.

Note: the artifact path is an mlrun data uri (e.g. *s3://bucket/path*) and can not be used with file utils.

example:

```
from os import path
project_name, artifact_path = set_environment(project='my-project')
set_environment("http://localhost:8080", artifact_path="./")
set_environment(env_file="mlrun.env")
set_environment("<remote-service-url>", access_key="xyz", username="joe")
```

Parameters

- **api_path** – location/url of mlrun api service
- **artifact_path** – path/url for storing experiment artifacts
- **project** – default project name
- **access_key** – set the remote cluster access key (V3IO_ACCESS_KEY)
- **user_project** – add the current user name to the provided project name (making it unique per user)
- **username** – name of the user to authenticate
- **env_file** – path/url to .env file (holding MLRun config and other env vars), see: set_env_from_file()
- **mock_functions** – set to True to create local/mock functions instead of real containers, set to "auto" to auto determine based on the presence of k8s/Nuclio

Returns default project name actual artifact path/url, can be used to create subpaths per task or group of artifacts

17.3 mlrun.artifacts

mlrun.artifacts.**get_model**(*model_dir*, *suffix=''*)

return model file, model spec object, and list of extra data items

this function will get the model file, metadata, and extra data the returned model file is always local, when using remote urls (such as v3io://, s3://, store://, ..) it will be copied locally.

returned extra data dict (of key, DataItem objects) allow reading additional model files/objects e.g. use DataItem.get() or .download(target) .as_df() to read

example:

```
model_file, model_artifact, extra_data = get_model(models_path, suffix='.pkl')
model = load(open(model_file, "rb"))
categories = extra_data['categories'].as_df()
```

Parameters

- **model_dir** – model dir or artifact path (store://..) or DataItem
- **suffix** – model filename suffix (when using a dir)

Returns model filename, model artifact object, extra data dict

mlrun.artifacts.**update_model**(*model_artifact*, *parameters: Optional[dict] = None*, *metrics: Optional[dict] = None*, *extra_data: Optional[dict] = None*, *inputs: Optional[List[*mlrun.features.Feature*]] = None*, *outputs: Optional[List[*mlrun.features.Feature*]] = None*, *feature_vector: Optional[str] = None*, *feature_weights: Optional[list] = None*, *key_prefix: str = ''*, *labels: Optional[dict] = None*, *write_spec_copy=True*, *store_object: bool = True*)

Update model object attributes

this method will edit or add attributes to a model object

example:

```
update_model(model_path, metrics={'speed': 100},
             extra_data={'my_data': b'some text', 'file': 's3://mybucket/..'})
```

Parameters

- **model_artifact** – model artifact object or path (store://..) or DataItem
- **parameters** – parameters dict
- **metrics** – model metrics e.g. accuracy
- **extra_data** – extra data items key, value dict (value can be: path string | bytes | artifact)
- **inputs** – list of input features (feature vector schema)
- **outputs** – list of output features (output vector schema)
- **feature_vector** – feature store feature vector uri (store://feature-vectors/<project>/<name>[:tag])
- **feature_weights** – list of feature weights, one per input column
- **key_prefix** – key prefix to add to metrics and extra data items
- **labels** – metadata labels
- **write_spec_copy** – write a YAML copy of the spec to the target dir
- **store_object** – Whether to store the model artifact updated.

17.4 mlrun.config

Configuration system.

Configuration can be in either a configuration file specified by MLRUN_CONFIG_FILE environment variable or by environment variables.

Environment variables are in the format "MLRUN_httpdb__port=8080". This will be mapped to config.httpdb.port. Values should be in JSON format.

class mlrun.config.**Config**(*cfg=None*)

Bases: object

property dask_kfp_image

See kfp_image property docstring for why we're defining this property

property dbpath

static decode_base64_config_and_load_to_object(*attribute_path: str*, *expected_type=<class 'dict'>*)

decodes and loads the config attribute to expected type :param attribute_path: the path in the default_config e.g. preemptible_nodes.node_selector :param expected_type: the object type valid values are : *dict*, *list* etc… :return: the expected type instance

dump_yaml(*stream=None*)

classmethod from_dict(*dict_*)

static get_build_args()

get_default_function_node_selector() → dict

static get_default_function_pod_requirement_resources(*requirement: str*, *with_gpu: bool = True*)

Parameters

- **requirement** – kubernetes requirement resource one of the following : requests, limits
- **with_gpu** – whether to return requirement resources with nvidia.com/gpu field (e.g. you cannot specify GPU requests without specifying GPU limits) https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

Returns a dict containing the defaults resources (cpu, memory, nvidia.com/gpu)

get_default_function_pod_resources(*with_gpu_requests=False*, *with_gpu_limits=False*)

get_default_function_security_context() → dict

static get_hub_url()

static get_parsed_igz_version() → Optional[semver.VersionInfo]

get_preemptible_node_selector() → dict

get_preemptible_tolerations() → list

static get_security_context_enrichment_group_id(*user_unix_id: int*) → int

static get_storage_auto_mount_params()

get_v3io_access_key()

static get_valid_function_priority_class_names()

property iguazio_api_url
we want to be able to run with old versions of the service who runs the API (which doesn't configure this value) so we're doing best effort to try and resolve it from other configurations TODO: Remove this hack when 0.6.x is old enough

is_api_running_on_k8s()

is_nuclio_detected()

static is_pip_ca_configured()

is_preemption_nodes_configured()

static is_running_on_iguazio() → bool

property kfp_image
When this configuration is not set we want to set it to mlrun/mlrun, but we need to use the enrich_image method. The problem is that the mlrun.utils.helpers module is importing the config (this) module, so we must import the module inside this function (and not on initialization), and then calculate this property value here.

static reload()

resolve_chief_api_url() → str

resolve_kfp_url(*namespace=None*)

resolve_runs_monitoring_missing_runtime_resources_debouncing_interval()

static resolve_ui_url()

to_dict()

update(*cfg*, *skip_errors=False*)

use_nuclio_mock(*force_mock=None*)

verify_security_context_enrichment_mode_is_allowed()

property version

mlrun.config.**is_running_as_api**()

mlrun.config.**read_env**(*env=None*, *prefix='MLRUN_'*)
Read configuration from environment

17.5 mlrun.datastore

class mlrun.datastore.**BigQuerySource**(*name: str = ''*, *table: Optional[str] = None*, *max_results_for_table: Optional[int] = None*, *query: Optional[str] = None*, *materialization_dataset: Optional[str] = None*, *chunksize: Optional[int] = None*, *key_field: Optional[str] = None*, *time_field: Optional[str] = None*, *schedule: Optional[str] = None*, *start_time=None*, *end_time=None*, *gcp_project: Optional[str] = None*, *spark_options: Optional[dict] = None*)
Bases: mlrun.datastore.sources.BaseSourceDriver

Reads Google BigQuery query results as input source for a flow.

example:

```
# use sql query
query_string = "SELECT * FROM `the-psf.pypi.downloads20210328` LIMIT 5000"
source = BigQuerySource("bq1", query=query_string,
                        gcp_project="my_project",
                        materialization_dataset="dataviews")

# read a table
source = BigQuerySource("bq2", table="the-psf.pypi.downloads20210328", gcp_project=
↪"my_project")
```

Parameters

- **name** – source name
- **table** – table name/path, cannot be used together with query
- **query** – sql query string
- **materialization_dataset** – for query with spark, The target dataset for the materialized view. This dataset should be in same location as the view or the queried tables. must be set to a dataset where the GCP user has table creation permission
- **chunksize** – number of rows per chunk (default large single chunk)
- **key_field** – the column to be used as the key for events. Can be a list of keys.
- **time_field** – the column to be parsed as the timestamp for events. Defaults to None
- **schedule** – string to configure scheduling of the ingestion job. For example *'*/30 * * * *'* will cause the job to run every 30 minutes
- **start_time** – filters out data before this time

- **end_time** – filters out data after this time
- **gcp_project** – google cloud project name
- **spark_options** – additional spark read options

is_iterator()

kind = 'bigquery'

support_spark = True

support_storey = False

to_dataframe()

to_spark_df(*session*, *named_view=False*)

class mlrun.datastore.**CSVSource**(*name: str = ''*, *path: Optional[str] = None*, *attributes: Optional[Dict[str, str]] = None*, *key_field: Optional[str] = None*, *time_field: Optional[str] = None*, *schedule: Optional[str] = None*, *parse_dates: Optional[Union[List[int], List[str]]] = None*)

Bases: mlrun.datastore.sources.BaseSourceDriver

Reads CSV file as input source for a flow.

Parameters

- **name** – name of the source
- **path** – path to CSV file
- **key_field** – the CSV field to be used as the key for events. May be an int (field index) or string (field name) if with_header is True. Defaults to None (no key). Can be a list of keys.
- **time_field** – the CSV field to be parsed as the timestamp for events. May be an int (field index) or string (field name) if with_header is True. Defaults to None (no timestamp field). The field will be parsed from isoformat (ISO-8601 as defined in datetime.fromisoformat()). In case the format is not isoformat, timestamp_format (as defined in datetime.strptime()) should be passed in attributes.
- **schedule** – string to configure scheduling of the ingestion job.
- **attributes** – additional parameters to pass to storey. For example: attributes={"timestamp_format": '%Y%m%d%H'}
- **parse_dates** – Optional. List of columns (names or integers, other than time_field) that will be attempted to parse as date column.

get_spark_options()

is_iterator()

kind = 'csv'

support_spark = True

support_storey = True

to_dataframe()

to_spark_df(*session*, *named_view=False*)

to_step(*key_field=None*, *time_field=None*, *context=None*)

class mlrun.datastore.**CSVTarget**(*name: str = ''*, *path=None*, *attributes: Optional[Dict[str, str]] = None*, *after_step=None*, *columns=None*, *partitioned: bool = False*, *key_bucketing_number: Optional[int] = None*, *partition_cols: Optional[List[str]] = None*, *time_partitioning_granularity: Optional[str] = None*, *after_state=None*, *max_events: Optional[int] = None*, *flush_after_seconds: Optional[int] = None*, *storage_options: Optional[Dict[str, str]] = None*)

Bases: mlrun.datastore.targets.BaseStoreTarget

add_writer_state(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*)

add_writer_step(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*, *featureset_status=None*)

as_df(*columns=None*, *df_module=None*, *entities=None*, *start_time=None*, *end_time=None*, *time_column=None*, ***kwargs*)

return the target data as dataframe

get_spark_options(*key_column=None*, *timestamp_key=None*, *overwrite=True*)

is_offline = True

is_single_file()

kind: str = 'csv'

prepare_spark_df(*df*)

suffix = '.csv'

support_spark = True

support_storey = True

class mlrun.datastore.**DataItem**(*key: str*, *store: mlrun.datastore.base.DataStore*, *subpath: str*, *url: str = ''*, *meta=None*, *artifact_url=None*)

Bases: object

Data input/output class abstracting access to various local/remote data sources

DataItem objects are passed into functions and can be used inside the function, when a function run completes users can access the run data via the run.artifact(key) which returns a DataItem object. users can also convert a data url (e.g. s3://bucket/key.csv) to a DataItem using *mlrun.get_dataitem(url)*.

Example:

```
# using data item inside a function
def my_func(context, data: DataItem):
    df = data.as_df()


# reading run results using DataItem (run.artifact())
train_run = train_iris_func.run(inputs={'dataset': dataset},
                                params={'label_column': 'label'})

train_run.artifact('confusion-matrix').show()
test_set = train_run.artifact('test_set').as_df()

# create and use DataItem from uri
data = mlrun.get_dataitem('http://xyz/data.json').get()
```

property artifact_url
DataItem artifact url (when its an artifact) or url for simple dataitems

as_df(*columns=None*, *df_module=None*, *format=''*, ***kwargs*)
return a dataframe object (generated from the dataitem).

Parameters

- **columns** – optional, list of columns to select
- **df_module** – optional, py module used to create the DataFrame (e.g. pd, dd, cudf, ..)
- **format** – file format, if not specified it will be deducted from the suffix

delete()
delete the object from the datastore

download(*target_path*)
download to the target dir/path

Parameters **target_path** – local target path for the downloaded item

get(*size=None*, *offset=0*, *encoding=None*)
read all or a byte range and return the content

Parameters

- **size** – number of bytes to get
- **offset** – fetch from offset (in bytes)
- **encoding** – encoding (e.g. "utf-8") for converting bytes to str

property key
DataItem key

property kind
DataItem store kind (file, s3, v3io, ..)

listdir()
return a list of child file names

local()
get the local path of the file, download to tmp first if its a remote object

ls()
return a list of child file names

property meta
Artifact Metadata, when the DataItem is read from the artifacts store

open(*mode*)
return fsspec file handler, if supported

put(*data*, *append=False*)
write/upload the data, append is only supported by some datastores

Parameters

- **data** – data (bytes/str) to write
- **append** – append data to the end of the object, NOT SUPPORTED BY SOME OBJECT STORES!

show(*format=None*)
show the data object content in Jupyter

Parameters **format** – format to use (when there is no/wrong suffix), e.g. 'png'

stat()
return FileStats class (size, modified, content_type)

property store
DataItem store object

property suffix
DataItem suffix (file extension) e.g. '.png'

upload(*src_path*)
upload the source file (src_path)

Parameters **src_path** – source file path to read from and upload

property url
//bucket/path

Type DataItem url e.g. /dir/path, s3

class mlrun.datastore.**HttpSource**(*name: Optional[str] = None*, *path: Optional[str] = None*, *attributes: Optional[Dict[str, str]] = None*, *key_field: Optional[str] = None*, *time_field: Optional[str] = None*, *workers: Optional[int] = None*)
Bases: mlrun.datastore.sources.OnlineSource

add_nuclio_trigger(*function*)

kind = 'http'

class mlrun.datastore.**KafkaSource**(*brokers=None*, *topics=None*, *group='serving'*, *initial_offset='earliest'*, *partitions=None*, *sasl_user=None*, *sasl_pass=None*, *attributes=None*, ***kwargs*)
Bases: mlrun.datastore.sources.OnlineSource

Sets kafka source for the flow

Sets kafka source for the flow

Parameters

- **brokers** – list of broker IP addresses
- **topics** – list of topic names on which to listen.
- **group** – consumer group. Default "serving"
- **initial_offset** – from where to consume the stream. Default earliest
- **partitions** – Optional, A list of partitions numbers for which the function receives events.
- **sasl_user** – Optional, user name to use for sasl authentications
- **sasl_pass** – Optional, password to use for sasl authentications
- **attributes** – Optional, extra attributes to be passed to kafka trigger

add_nuclio_trigger(*function*)

kind = 'kafka'

class mlrun.datastore.**NoSqlTarget**(**args*, ***kwargs*)
Bases: mlrun.datastore.targets.NoSqlBaseTarget

get_table_object()
get storey Table object

kind: str = 'nosql'

support_spark = True

writer_step_name = 'NoSqlTarget'

class mlrun.datastore.**ParquetSource**(*name: str = ''*, *path: Optional[str] = None*, *attributes: Optional[Dict[str, str]] = None*, *key_field: Optional[str] = None*, *time_field: Optional[str] = None*, *schedule: Optional[str] = None*, *start_time: Optional[Union[datetime.datetime, str]] = None*, *end_time: Optional[Union[datetime.datetime, str]] = None*)

Bases: mlrun.datastore.sources.BaseSourceDriver

Reads Parquet file/dir as input source for a flow.

Parameters

- **name** – name of the source
- **path** – path to Parquet file or directory
- **key_field** – the column to be used as the key for events. Can be a list of keys.
- **time_field** – the column to be parsed as the timestamp for events. Defaults to None
- **start_filter** – datetime. If not None, the results will be filtered by partitions and 'filter_column' > start_filter. Default is None
- **end_filter** – datetime. If not None, the results will be filtered by partitions 'filter_column' <= end_filter. Default is None
- **filter_column** – Optional. if not None, the results will be filtered by this column and start_filter & end_filter
- **schedule** – string to configure scheduling of the ingestion job. For example *'*/30 * * * *'* will cause the job to run every 30 minutes
- **start_time** – filters out data before this time
- **end_time** – filters out data after this time
- **attributes** – additional parameters to pass to storey.

property end_time

get_spark_options()

kind = 'parquet'

property start_time

support_spark = True

support_storey = True

to_dataframe()

to_step(*key_field=None*, *time_field=None*, *start_time=None*, *end_time=None*, *context=None*)

class mlrun.datastore.**ParquetTarget**(*name: str = ''*, *path=None*, *attributes: Optional[Dict[str, str]] = None*, *after_step=None*, *columns=None*, *partitioned: Optional[bool] = None*, *key_bucketing_number: Optional[int] = None*, *partition_cols: Optional[List[str]] = None*, *time_partitioning_granularity: Optional[str] = None*, *after_state=None*, *max_events: Optional[int] = 10000*, *flush_after_seconds: Optional[int] = 900*, *storage_options: Optional[Dict[str, str]] = None*)

Bases: mlrun.datastore.targets.BaseStoreTarget

parquet target storage driver, used to materialize feature set/vector data into parquet files

Parameters

- **name** – optional, target name. By default will be called ParquetTarget
- **path** – optional, Output path. Can be either a file or directory. This parameter is forwarded as-is to pandas.DataFrame.to_parquet(). Default location v3io:///projects/{project}/FeatureStore/{name}/parquet/
- **attributes** – optional, extra attributes for storey.ParquetTarget
- **after_step** – optional, after what step in the graph to add the target
- **columns** – optional, which columns from data to write
- **partitioned** – optional, whether to partition the file, False by default, if True without passing any other partition field, the data will be partitioned by /year/month/day/hour
- **key_bucketing_number** – optional, None by default will not partition by key, 0 will partition by the key as is, any other number X will create X partitions and hash the keys to one of them
- **partition_cols** – optional, name of columns from the data to partition by
- **time_partitioning_granularity** – optional. the smallest time unit to partition the data by. For example "hour" will yield partitions of the format /year/month/day/hour
- **max_events** – optional. Maximum number of events to write at a time. All events will be written on flow termination, or after flush_after_seconds (if flush_after_seconds is set). Default 10k events
- **flush_after_seconds** – optional. Maximum number of seconds to hold events before they are written. All events will be written on flow termination, or after max_events are accumulated (if max_events is set). Default 15 minutes

add_writer_state(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*)

add_writer_step(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*, *featureset_status=None*)

as_df(*columns=None*, *df_module=None*, *entities=None*, *start_time=None*, *end_time=None*, *time_column=None*, ***kwargs*)
return the target data as dataframe

get_dask_options()

get_spark_options(*key_column=None*, *timestamp_key=None*, *overwrite=True*)

is_offline = True

is_single_file()

kind: str = 'parquet'

support_append = True

support_dask = True

support_spark = True

support_storey = True

class mlrun.datastore.**StreamSource**(*name='stream'*, *group='serving'*, *seek_to='earliest'*, *shards=1*, *retention_in_hours=24*, *extra_attributes: Optional[dict] = None*, ***kwargs*)

Bases: mlrun.datastore.sources.OnlineSource

Sets stream source for the flow. If stream doesn't exist it will create it

Sets stream source for the flow. If stream doesn't exist it will create it

Parameters

- **name** – stream name. Default "stream"
- **group** – consumer group. Default "serving"
- **seek_to** – from where to consume the stream. Default earliest
- **shards** – number of shards in the stream. Default 1
- **retention_in_hours** – if stream doesn't exist and it will be created set retention time. Default 24h
- **extra_attributes** – additional nuclio trigger attributes (key/value dict)

add_nuclio_trigger(*function*)

kind = 'v3ioStream'

class mlrun.datastore.**StreamTarget**(*name: str = ''*, *path=None*, *attributes: Optional[Dict[str, str]] = None*, *after_step=None*, *columns=None*, *partitioned: bool = False*, *key_bucketing_number: Optional[int] = None*, *partition_cols: Optional[List[str]] = None*, *time_partitioning_granularity: Optional[str] = None*, *after_state=None*, *max_events: Optional[int] = None*, *flush_after_seconds: Optional[int] = None*, *storage_options: Optional[Dict[str, str]] = None*)

Bases: mlrun.datastore.targets.BaseStoreTarget

add_writer_state(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*)

add_writer_step(*graph*, *after*, *features*, *key_columns=None*, *timestamp_key=None*, *featureset_status=None*)

as_df(*columns=None*, *df_module=None*, ***kwargs*)
return the target data as dataframe

is_online = False

is_table = False

kind: str = 'stream'

support_append = True

support_spark = False

support_storey = True

mlrun.datastore.**get_store_resource**(*uri*, *db=None*, *secrets=None*, *project=None*, *data_store_secrets=None*)

get store resource object by uri

17.6 mlrun.db

class mlrun.db.httpdb.**HTTPRunDB**(*base_url*, *user=''*, *password=''*, *token=''*)

Bases: mlrun.db.base.RunDBInterface

Interface for accessing and manipulating the *mlrun* persistent store, maintaining the full state and catalog of objects that MLRun uses. The *HTTPRunDB* class serves as a client-side proxy to the MLRun API service which maintains the actual data-store, accesses the server through REST APIs.

The class provides functions for accessing and modifying the various objects that are used by MLRun in its operation. The functions provided follow some standard guidelines, which are:

- Every object in MLRun exists in the context of a project (except projects themselves). When referencing an object through any API, a project name must be provided. The default for most APIs is for an empty project name, which will be replaced by the name of the default project (usually `default`). Therefore, if performing an API to list functions, for example, and not providing a project name - the result will not be functions from all projects but rather from the `default` project.
- Many objects can be assigned labels, and listed/queried by label. The label parameter for query APIs allows for listing objects that:
  - Have a specific label, by asking for `label="<label_name>"`. In this case the actual value of the label doesn't matter and every object with that label will be returned
  - Have a label with a specific value. This is done by specifying `label="<label_name>=<label_value>"`. In this case only objects whose label matches the value will be returned
- Most objects have a `create` method as well as a `store` method. Create can only be called when such an does not exist yet, while store allows for either creating a new object or overwriting an existing object.
- Some objects have a `versioned` option, in which case overwriting the same object with a different version of it does not delete the previous version, but rather creates a new version of the object and keeps both versions. Versioned objects usually have a `uid` property which is based on their content and allows to reference a specific version of an object (other than tagging objects, which also allows for easy referencing).
- Many objects have both a `store` function and a `patch` function. These are used in the same way as the corresponding REST verbs - a `store` is passed a full object and will basically perform a PUT operation, replacing the full object (if it exists) while `patch` receives just a dictionary containing the differences to be applied to the object, and will merge those changes to the existing object. The `patch` operation also has a strategy assigned to it which determines how the merge logic should behave. The strategy can be either `replace` or `additive`. For further details on those strategies, refer to https://pypi.org/project/mergedeep/

abort_run(*uid*, *project=''*, *iter=0*)

Abort a running run - will remove the run's runtime resources and mark its state as aborted

api_call(*method*, *path*, *error=None*, *params=None*, *body=None*, *json=None*, *headers=None*, *timeout=45*, *version=None*)

Perform a direct REST API call on the *mlrun* API server.

> **Caution:** For advanced usage - prefer using the various APIs exposed through this class, rather than directly invoking REST calls.

Parameters

- **method** – REST method (POST, GET, PUT…)
- **path** – Path to endpoint executed, for example `"projects"`
- **error** – Error to return if API invocation fails
- **body** – Payload to be passed in the call. If using JSON objects, prefer using the `json` param
- **json** – JSON payload to be passed in the call
- **headers** – REST headers, passed as a dictionary: `{"<header-name>": "<header-value>"}`
- **timeout** – API call timeout
- **version** – API version to use, None (the default) will mean to use the default value from config, for un-versioned api set an empty string.

Returns Python HTTP response object

connect(*secrets=None*)

Connect to the MLRun API server. Must be called prior to executing any other method. The code utilizes the URL for the API server from the configuration - `mlconf.dbpath`.

For example:

```
mlconf.dbpath = mlconf.dbpath or 'http://mlrun-api:8080'
db = get_run_db().connect()
```

create_feature_set(*feature_set: Union[dict, mlrun.api.schemas.feature_store.FeatureSet, mlrun.feature_store.feature_set.FeatureSet], project='', versioned=True*) → dict

Create a new *FeatureSet* and save in the *mlrun* DB. The feature-set must not previously exist in the DB.

Parameters

- **feature_set** – The new *FeatureSet* to create.
- **project** – Name of project this feature-set belongs to.
- **versioned** – Whether to maintain versions for this feature-set. All versions of a versioned object will be kept in the DB and can be retrieved until explicitly deleted.

Returns The *FeatureSet* object (as dict).

create_feature_vector(*feature_vector: Union[dict, mlrun.api.schemas.feature_store.FeatureVector, mlrun.feature_store.feature_vector.FeatureVector], project='', versioned=True*) → dict

Create a new *FeatureVector* and save in the *mlrun* DB.

Parameters

- **feature_vector** – The new *FeatureVector* to create.
- **project** – Name of project this feature-vector belongs to.
- **versioned** – Whether to maintain versions for this feature-vector. All versions of a versioned object will be kept in the DB and can be retrieved until explicitly deleted.

Returns The *FeatureVector* object (as dict).

create_marketplace_source(*source: Union[dict, mlrun.api.schemas.marketplace.IndexedMarketplaceSource]*)

Add a new marketplace source.

MLRun maintains an ordered list of marketplace sources ("sources") Each source has its details registered and its order within the list. When creating a new source, the special order `-1` can be used to mark this source as last in the list. However, once the source is in the MLRun list, its order will always be `>0`.

The global marketplace source always exists in the list, and is always the last source (`order = -1`). It cannot be modified nor can it be moved to another order in the list.

The source object may contain credentials which are needed to access the datastore where the source is stored. These credentials are not kept in the MLRun DB, but are stored inside a kubernetes secret object maintained by MLRun. They are not returned through any API from MLRun.

Example:

```
import mlrun.api.schemas

# Add a private source as the last one (will be #1 in the list)
private_source = mlrun.api.schemas.IndexedMarketplaceSource(
    order=-1,
    source=mlrun.api.schemas.MarketplaceSource(
        metadata=mlrun.api.schemas.MarketplaceObjectMetadata(name="priv",␣
↪description="a private source"),
        spec=mlrun.api.schemas.MarketplaceSourceSpec(path="/local/path/to/
↪source", channel="development")
    )
)
db.create_marketplace_source(private_source)

# Add another source as 1st in the list - will push previous one to be #2
another_source = mlrun.api.schemas.IndexedMarketplaceSource(
    order=1,
    source=mlrun.api.schemas.MarketplaceSource(
        metadata=mlrun.api.schemas.MarketplaceObjectMetadata(name="priv-2",␣
↪description="another source"),
        spec=mlrun.api.schemas.MarketplaceSourceSpec(
            path="/local/path/to/source/2",
            channel="development",
            credentials={...}
        )
    )
)
db.create_marketplace_source(another_source)
```

Parameters **source** – The source and its order, of type `IndexedMarketplaceSource`, or in dictionary form.

Returns The source object as inserted into the database, with credentials stripped.

create_or_patch_model_endpoint(*project: str*, *endpoint_id: str*, *model_endpoint: mlrun.api.schemas.model_endpoints.ModelEndpoint*, *access_key: Optional[str] = None*)

Creates or updates a KV record with the given model_endpoint record

Parameters

- **project** – The name of the project
- **endpoint_id** – The id of the endpoint
- **model_endpoint** – An object representing the model endpoint
- **access_key** – V3IO access key, when None, will be look for in environ

create_project(*project: Union[dict,* mlrun.projects.project.MlrunProject*, mlrun.api.schemas.project.Project]*) → *mlrun.projects.project.MlrunProject*

Create a new project. A project with the same name must not exist prior to creation.

create_project_secrets(*project: str*, *provider: Union[str,* mlrun.api.schemas.secret.SecretProviderName*] = SecretProviderName.kubernetes*, *secrets: Optional[dict] = None*)

Create project-context secrets using either `vault` or `kubernetes` provider. When using with Vault, this will create needed Vault structures for storing secrets in project-context, and store a set of secret values. The method generates Kubernetes service-account and the Vault authentication structures that are required for function Pods to authenticate with Vault and be able to extract secret values passed as part of their context.

Note: This method used with Vault is currently in technical preview, and requires a HashiCorp Vault infrastructure properly set up and connected to the MLRun API server.

When used with Kubernetes, this will make sure that the project-specific k8s secret is created, and will populate it with the secrets provided, replacing their values if they exist.

Parameters

- **project** – The project context for which to generate the infra and store secrets.
- **provider** – The name of the secrets-provider to work with. Accepts a *SecretProviderName* enum.
- **secrets** – A set of secret values to store. Example:

```
secrets = {'password': 'myPassw0rd', 'aws_key': '111222333'}
db.create_project_secrets(
    "project1",
    provider=mlrun.api.schemas.SecretProviderName.kubernetes,
    secrets=secrets
)
```

create_schedule(*project: str*, *schedule: mlrun.api.schemas.schedule.ScheduleInput*)

Create a new schedule on the given project. The details on the actual object to schedule as well as the schedule itself are within the schedule object provided. The `ScheduleCronTrigger` follows the guidelines in https://apscheduler.readthedocs.io/en/v3.6.3/modules/triggers/cron.html. It also supports a `from_crontab()` function that accepts a crontab-formatted string (see https://en.wikipedia.org/wiki/Cron for more information on the format).

Example:

```
from mlrun.api import schemas

# Execute the get_data_func function every Tuesday at 15:30
schedule = schemas.ScheduleInput(
    name="run_func_on_tuesdays",
    kind="job",
    scheduled_object=get_data_func,
```

(continues on next page)

(continued from previous page)

```
    cron_trigger=schemas.ScheduleCronTrigger(day_of_week='tue', hour=15,␣
↪minute=30),
)
db.create_schedule(project_name, schedule)
```

create_user_secrets(*user: str*, *provider: Union[str, mlrun.api.schemas.secret.SecretProviderName] = SecretProviderName.vault*, *secrets: Optional[dict] = None*)

Create user-context secret in Vault. Please refer to *create_project_secrets()* for more details and status of this functionality.

Note: This method is currently in technical preview, and requires a HashiCorp Vault infrastructure properly set up and connected to the MLRun API server.

Parameters

- **user** – The user context for which to generate the infra and store secrets.
- **provider** – The name of the secrets-provider to work with. Currently only `vault` is supported.
- **secrets** – A set of secret values to store within the Vault.

del_artifact(*key*, *tag=None*, *project=''*)

Delete an artifact.

del_artifacts(*name=None*, *project=None*, *tag=None*, *labels=None*, *days_ago=0*)

Delete artifacts referenced by the parameters.

Parameters

- **name** – Name of artifacts to delete. Note that this is a like query, and is case-insensitive. See *list_artifacts()* for more details.
- **project** – Project that artifacts belong to.
- **tag** – Choose artifacts who are assigned this tag.
- **labels** – Choose artifacts which are labeled.
- **days_ago** – This parameter is deprecated and not used.

del_run(*uid*, *project=''*, *iter=0*)

Delete details of a specific run from DB.

Parameters

- **uid** – Unique ID for the specific run to delete.
- **project** – Project that the run belongs to.
- **iter** – Iteration within a specific task.

del_runs(*name=None*, *project=None*, *labels=None*, *state=None*, *days_ago=0*)

Delete a group of runs identified by the parameters of the function.

Example:

```
db.del_runs(state='completed')
```

Parameters

- **name** – Name of the task which the runs belong to.
- **project** – Project to which the runs belong.
- **labels** – Filter runs that are labeled using these specific label values.
- **state** – Filter only runs which are in this state.
- **days_ago** – Filter runs whose start time is newer than this parameter.

delete_artifacts_tags(*artifacts*, *project: str*, *tag_name: str*)
Delete tag from a list of artifacts.

Parameters

- **artifacts** – The artifacts to delete the tag from. Can be a list of `Artifact` objects or dictionaries, or a single object.
- **project** – Project which contains the artifacts.
- **tag_name** – The tag to set on the artifacts.

delete_feature_set(*name*, *project=''*, *tag=None*, *uid=None*)
Delete a *FeatureSet* object from the DB. If `tag` or `uid` are specified, then just the version referenced by them will be deleted. Using both is not allowed. If none are specified, then all instances of the object whose name is `name` will be deleted.

delete_feature_vector(*name*, *project=''*, *tag=None*, *uid=None*)
Delete a *FeatureVector* object from the DB. If `tag` or `uid` are specified, then just the version referenced by them will be deleted. Using both is not allowed. If none are specified, then all instances of the object whose name is `name` will be deleted.

delete_function(*name: str*, *project: str = ''*)
Delete a function belonging to a specific project.

delete_marketplace_source(*source_name: str*)
Delete a marketplace source from the DB. The source will be deleted from the list, and any following sources will be promoted - for example, if the 1st source is deleted, the 2nd source will become #1 in the list. The global marketplace source cannot be deleted.

Parameters **source_name** – Name of the marketplace source to delete.

delete_model_endpoint_record(*project: str*, *endpoint_id: str*, *access_key: Optional[str] = None*)
Deletes the KV record of a given model endpoint, project and endpoint_id are used for lookup

Parameters

- **project** – The name of the project
- **endpoint_id** – The id of the endpoint
- **access_key** – V3IO access key, when None, will be look for in environ

delete_objects_tag(*project: str*, *tag_name: str*, *tag_objects: Union[mlrun.api.schemas.tag.TagObjects, dict]*)
Delete a tag from a list of objects.

Parameters

- **project** – Project which contains the objects.
- **tag_name** – The tag to delete from the objects.
- **tag_objects** – The objects to delete the tag from.

delete_project(*name: str*, *deletion_strategy: Union[str, mlrun.api.schemas.constants.DeletionStrategy] = DeletionStrategy.restricted*)

Delete a project.

Parameters

- **name** – Name of the project to delete.
- **deletion_strategy** – How to treat child objects of the project. Possible values are:
  - `restrict` (default) - Project must not have any child objects when deleted. If using this mode while child objects exist, the operation will fail.
  - `cascade` - Automatically delete all child objects when deleting the project.

delete_project_secrets(*project: str*, *provider: Union[str, mlrun.api.schemas.secret.SecretProviderName] = SecretProviderName.kubernetes*, *secrets: Optional[List[str]] = None*)

Delete project-context secrets from Kubernetes.

Parameters

- **project** – The project name.
- **provider** – The name of the secrets-provider to work with. Currently only `kubernetes` is supported.
- **secrets** – A list of secret names to delete. An empty list will delete all secrets assigned to this specific project.

delete_runtime(*kind: str*, *label_selector: Optional[str] = None*, *force: bool = False*, *grace_period: Optional[int] = None*)

Deprecated use *delete_runtime_resources()* (with kind filter) instead

delete_runtime_object(*kind: str*, *object_id: str*, *label_selector: Optional[str] = None*, *force: bool = False*, *grace_period: Optional[int] = None*)

Deprecated use *delete_runtime_resources()* (with kind and object_id filter) instead

delete_runtime_resources(*project: Optional[str] = None*, *label_selector: Optional[str] = None*, *kind: Optional[str] = None*, *object_id: Optional[str] = None*, *force: bool = False*, *grace_period: Optional[int] = None*) → Dict[str, Dict[str, mlrun.api.schemas.runtime_resource.RuntimeResources]]

Delete all runtime resources which are in terminal state.

Parameters

- **project** – Delete only runtime resources of a specific project, by default None, which will delete only from the projects you're authorized to delete from.
- **label_selector** – Delete only runtime resources matching the label selector.
- **kind** – The kind of runtime to delete. May be one of *['dask', 'job', 'spark', 'remote-spark', 'mpijob']*
- **object_id** – The identifier of the mlrun object to delete its runtime resources. for most function runtimes, runtime resources are per Run, for which the identifier is the Run's UID. For dask runtime, the runtime resources are per Function, for which the identifier is the Function's name.
- **force** – Force deletion - delete the runtime resource even if it's not in terminal state or if the grace period didn't pass.
- **grace_period** – Grace period given to the runtime resource before they are actually removed, counted from the moment they moved to terminal state.

Returns `GroupedByProjectRuntimeResourcesOutput` listing the runtime resources that were removed.

delete_runtimes(*label_selector: Optional[str] = None*, *force: bool = False*, *grace_period: Optional[int] = None*)
Deprecated use `delete_runtime_resources()` instead

delete_schedule(*project: str*, *name: str*)
Delete a specific schedule by name.

static get_api_path_prefix(*version: Optional[str] = None*) → str

Parameters **version** – API version to use, None (the default) will mean to use the default value from mlconf, for un-versioned api set an empty string.

get_background_task(*name: str*) → mlrun.api.schemas.background_task.BackgroundTask
Retrieve updated information on a background task being executed.

get_base_api_url(*path: str*, *version: Optional[str] = None*) → str

get_builder_status(*func:* mlrun.runtimes.base.BaseRuntime, *offset=0*, *logs=True*, *last_log_timestamp=0*, *verbose=False*)
Retrieve the status of a build operation currently in progress.

Parameters

- **func** – Function object that is being built.
- **offset** – Offset into the build logs to retrieve logs from.
- **logs** – Should build logs be retrieved.
- **last_log_timestamp** – Last timestamp of logs that were already retrieved. Function will return only logs later than this parameter.
- **verbose** – Add verbose logs into the output.

Returns

The following parameters:

- Text of builder logs.
- Timestamp of last log retrieved, to be used in subsequent calls to this function.

The function also updates internal members of the `func` object to reflect build process info.

get_feature_set(*name: str*, *project: str = ''*, *tag: Optional[str] = None*, *uid: Optional[str] = None*) → *mlrun.feature_store.feature_set.FeatureSet*
Retrieve a ~mlrun.feature_store.FeatureSet` object. If both `tag` and `uid` are not specified, then the object tagged `latest` will be retrieved.

Parameters

- **name** – Name of object to retrieve.
- **project** – Project the FeatureSet belongs to.
- **tag** – Tag of the specific object version to retrieve.
- **uid** – uid of the object to retrieve (can only be used for versioned objects).

get_feature_vector(*name: str*, *project: str = ''*, *tag: Optional[str] = None*, *uid: Optional[str] = None*) → *mlrun.feature_store.feature_vector.FeatureVector*
Return a specific feature-vector referenced by its tag or uid. If none are provided, `latest` tag will be used.

get_function(*name*, *project=''*, *tag=None*, *hash_key=''*)
Retrieve details of a specific function, identified by its name and potentially a tag or function hash.

get_log(*uid*, *project=''*, *offset=0*, *size=- 1*)
Retrieve a log.

Parameters

- **uid** – Log unique ID
- **project** – Project name for which the log belongs
- **offset** – Retrieve partial log, get up to `size` bytes starting at offset `offset` from beginning of log
- **size** – See `offset`. If set to `-1` (the default) will retrieve all data to end of log.

Returns

The following objects:

- state - The state of the runtime object which generates this log, if it exists. In case no known state exists, this will be `unknown`.
- content - The actual log content.

get_marketplace_catalog(*source_name: str*, *channel: Optional[str] = None*, *version: Optional[str] = None*, *tag: Optional[str] = None*, *force_refresh: bool = False*)
Retrieve the item catalog for a specified marketplace source. The list of items can be filtered according to various filters, using item's metadata to filter.

Parameters

- **source_name** – Name of the source.
- **channel** – Filter items according to their channel. For example `development`.
- **version** – Filter items according to their version.
- **tag** – Filter items based on tag.
- **force_refresh** – Make the server fetch the catalog from the actual marketplace source, rather than rely on cached information which may exist from previous get requests. For example, if the source was re-built, this will make the server get the updated information. Default is `False`.

Returns `MarketplaceCatalog` object, which is essentially a list of `MarketplaceItem` entries.

get_marketplace_item(*source_name: str*, *item_name: str*, *channel: str = 'development'*, *version: Optional[str] = None*, *tag: str = 'latest'*, *force_refresh: bool = False*)
Retrieve a specific marketplace item.

Parameters

- **source_name** – Name of source.
- **item_name** – Name of the item to retrieve, as it appears in the catalog.
- **channel** – Get the item from the specified channel. Default is `development`.
- **version** – Get a specific version of the item. Default is `None`.
- **tag** – Get a specific version of the item identified by tag. Default is `latest`.
- **force_refresh** – Make the server fetch the information from the actual marketplace source, rather than rely on cached information. Default is `False`.

Returns MarketplaceItem.

get_marketplace_source(*source_name: str*)
Retrieve a marketplace source from the DB.

Parameters **source_name** – Name of the marketplace source to retrieve.

get_model_endpoint(*project: str*, *endpoint_id: str*, *start: Optional[str] = None*, *end: Optional[str] = None*, *metrics: Optional[List[str]] = None*, *feature_analysis: bool = False*, *access_key: Optional[str] = None*) → mlrun.api.schemas.model_endpoints.ModelEndpoint
Returns a ModelEndpoint object with additional metrics and feature related data.

Parameters

- **project** – The name of the project
- **endpoint_id** – The id of the model endpoint
- **metrics** – A list of metrics to return for each endpoint, read more in 'TimeMetric'
- **start** – The start time of the metrics
- **end** – The end time of the metrics
- **feature_analysis** – When True, the base feature statistics and current feature statistics will be added to the output of the resulting object
- **access_key** – V3IO access key, when None, will be look for in environ

get_pipeline(*run_id: str*, *namespace: Optional[str] = None*, *timeout: int = 10*, *format_: Union[str, mlrun.api.schemas.pipeline.PipelinesFormat] = PipelinesFormat.summary*, *project: Optional[str] = None*)
Retrieve details of a specific pipeline using its run ID (as provided when the pipeline was executed).

get_project(*name: str*) → *mlrun.projects.project.MlrunProject*
Get details for a specific project.

get_project_background_task(*project: str*, *name: str*) → mlrun.api.schemas.background_task.BackgroundTask
Retrieve updated information on a project background task being executed.

get_runtime(*kind: str*, *label_selector: Optional[str] = None*) → Dict
Deprecated use *list_runtime_resources()* (with kind filter) instead

get_schedule(*project: str*, *name: str*, *include_last_run: bool = False*) → mlrun.api.schemas.schedule.ScheduleOutput
Retrieve details of the schedule in question. Besides returning the details of the schedule object itself, this function also returns the next scheduled run for this specific schedule, as well as potentially the results of the last run executed through this schedule.

Parameters

- **project** – Project name.
- **name** – Name of the schedule object to query.
- **include_last_run** – Whether to include the results of the schedule's last run in the response.

invoke_schedule(*project: str*, *name: str*)
Execute the object referenced by the schedule immediately.

kind = 'http'

list_artifact_tags(*project=None*, *category: Optional[Union[str, mlrun.api.schemas.artifact.ArtifactCategories]] = None*) → List[str]

Return a list of all the tags assigned to artifacts in the scope of the given project.

list_artifacts(*name=None*, *project=None*, *tag=None*, *labels=None*, *since=None*, *until=None*, *iter: Optional[int] = None*, *best_iteration: bool = False*, *kind: Optional[str] = None*, *category: Optional[Union[str, mlrun.api.schemas.artifact.ArtifactCategories]] = None*) → mlrun.lists.ArtifactList

List artifacts filtered by various parameters.

Examples:

```
# Show latest version of all artifacts in project
latest_artifacts = db.list_artifacts('', tag='latest', project='iris')
# check different artifact versions for a specific artifact
result_versions = db.list_artifacts('results', tag='*', project='iris')
```

Parameters

- **name** – Name of artifacts to retrieve. Name is used as a like query, and is not case-sensitive. This means that querying for `name` may return artifacts named `my_Name_1` or `surname`.
- **project** – Project name.
- **tag** – Return artifacts assigned this tag.
- **labels** – Return artifacts that have these labels.
- **since** – Not in use in *HTTPRunDB*.
- **until** – Not in use in *HTTPRunDB*.
- **iter** – Return artifacts from a specific iteration (where `iter=0` means the root iteration). If `None` (default) return artifacts from all iterations.
- **best_iteration** – Returns the artifact which belongs to the best iteration of a given run, in the case of artifacts generated from a hyper-param run. If only a single iteration exists, will return the artifact from that iteration. If using `best_iter`, the `iter` parameter must not be used.
- **kind** – Return artifacts of the requested kind.
- **category** – Return artifacts of the requested category.

list_entities(*project: str*, *name: Optional[str] = None*, *tag: Optional[str] = None*, *labels: Optional[List[str]] = None*) → List[dict]

Retrieve a list of entities and their mapping to the containing feature-sets. This function is similar to the *list_features()* function, and uses the same logic. However, the entities are matched against the name rather than the features.

list_feature_sets(*project: str = ''*, *name: Optional[str] = None*, *tag: Optional[str] = None*, *state: Optional[str] = None*, *entities: Optional[List[str]] = None*, *features: Optional[List[str]] = None*, *labels: Optional[List[str]] = None*, *partition_by: Optional[Union[mlrun.api.schemas.constants.FeatureStorePartitionByField, str]] = None*, *rows_per_partition: int = 1*, *partition_sort_by: Optional[Union[mlrun.api.schemas.constants.SortField, str]] = None*, *partition_order: Union[mlrun.api.schemas.constants.OrderType, str] = OrderType.desc*) → List[*mlrun.feature_store.feature_set.FeatureSet*]

Retrieve a list of feature-sets matching the criteria provided.

Parameters

- **project** – Project name.
- **name** – Name of feature-set to match. This is a like query, and is case-insensitive.
- **tag** – Match feature-sets with specific tag.
- **state** – Match feature-sets with a specific state.
- **entities** – Match feature-sets which contain entities whose name is in this list.
- **features** – Match feature-sets which contain features whose name is in this list.
- **labels** – Match feature-sets which have these labels.
- **partition_by** – Field to group results by. Only allowed value is *name*. When *partition_by* is specified, the *partition_sort_by* parameter must be provided as well.
- **rows_per_partition** – How many top rows (per sorting defined by *partition_sort_by* and *partition_order*) to return per group. Default value is 1.
- **partition_sort_by** – What field to sort the results by, within each partition defined by *partition_by*. Currently the only allowed value are *created* and *updated*.
- **partition_order** – Order of sorting within partitions - *asc* or *desc*. Default is *desc*.

Returns List of matching `FeatureSet` objects.

list_feature_vectors(*project: str = ''*, *name: Optional[str] = None*, *tag: Optional[str] = None*, *state: Optional[str] = None*, *labels: Optional[List[str]] = None*, *partition_by: Optional[Union[mlrun.api.schemas.constants.FeatureStorePartitionByField, str]] = None*, *rows_per_partition: int = 1*, *partition_sort_by: Optional[Union[mlrun.api.schemas.constants.SortField, str]] = None*, *partition_order: Union[mlrun.api.schemas.constants.OrderType, str] = OrderType.desc*) → List[*mlrun.feature_store.feature_vector.FeatureVector*]

Retrieve a list of feature-vectors matching the criteria provided.

Parameters

- **project** – Project name.
- **name** – Name of feature-vector to match. This is a like query, and is case-insensitive.
- **tag** – Match feature-vectors with specific tag.
- **state** – Match feature-vectors with a specific state.
- **labels** – Match feature-vectors which have these labels.
- **partition_by** – Field to group results by. Only allowed value is *name*. When *partition_by* is specified, the *partition_sort_by* parameter must be provided as well.
- **rows_per_partition** – How many top rows (per sorting defined by *partition_sort_by* and *partition_order*) to return per group. Default value is 1.
- **partition_sort_by** – What field to sort the results by, within each partition defined by *partition_by*. Currently the only allowed values are *created* and *updated*.
- **partition_order** – Order of sorting within partitions - *asc* or *desc*. Default is *desc*.

Returns List of matching `FeatureVector` objects.

list_features(*project: str*, *name: Optional[str] = None*, *tag: Optional[str] = None*, *entities: Optional[List[str]] = None*, *labels: Optional[List[str]] = None*) → List[dict]

List feature-sets which contain specific features. This function may return multiple versions of the same feature-set if a specific tag is not requested. Note that the various filters of this function actually refer to the feature-set object containing the features, not to the features themselves.

Parameters

- **project** – Project which contains these features.
- **name** – Name of the feature to look for. The name is used in a like query, and is not case-sensitive. For example, looking for `feat` will return features which are named `MyFeature` as well as `defeat`.
- **tag** – Return feature-sets which contain the features looked for, and are tagged with the specific tag.
- **entities** – Return only feature-sets which contain an entity whose name is contained in this list.
- **labels** – Return only feature-sets which are labeled as requested.

Returns A list of mapping from feature to a digest of the feature-set, which contains the feature-set meta-data. Multiple entries may be returned for any specific feature due to multiple tags or versions of the feature-set.

list_functions(*name=None*, *project=None*, *tag=None*, *labels=None*)

Retrieve a list of functions, filtered by specific criteria.

Parameters

- **name** – Return only functions with a specific name.
- **project** – Return functions belonging to this project. If not specified, the default project is used.
- **tag** – Return function versions with specific tags.
- **labels** – Return functions that have specific labels assigned to them.

Returns List of function objects (as dictionary).

list_marketplace_sources()

List marketplace sources in the MLRun DB.

list_model_endpoints(*project: str*, *model: Optional[str] = None*, *function: Optional[str] = None*, *labels: Optional[List[str]] = None*, *start: str = 'now-1h'*, *end: str = 'now'*, *metrics: Optional[List[str]] = None*, *access_key: Optional[str] = None*, *top_level: bool = False*, *uids: Optional[List[str]] = None*) → mlrun.api.schemas.model_endpoints.ModelEndpointList

Returns a list of ModelEndpointState objects. Each object represents the current state of a model endpoint. This functions supports filtering by the following parameters: 1) model 2) function 3) labels By default, when no filters are applied, all available endpoints for the given project will be listed.

In addition, this functions provides a facade for listing endpoint related metrics. This facade is time-based and depends on the 'start' and 'end' parameters. By default, when the metrics parameter is None, no metrics are added to the output of this function.

Parameters

- **project** – The name of the project
- **model** – The name of the model to filter by

- **function** – The name of the function to filter by
- **labels** – A list of labels to filter by. Label filters work by either filtering a specific value of a label (i.e. list("key==value")) or by looking for the existence of a given key (i.e. "key")
- **metrics** – A list of metrics to return for each endpoint, read more in 'TimeMetric'
- **start** – The start time of the metrics
- **end** – The end time of the metrics
- **access_key** – V3IO access key, when None, will be look for in environ
- **top_level** – if true will return only routers and endpoint that are NOT children of any router
- **uids** – if passed will return ModelEndpointList of endpoints with uid in uids

list_pipelines(*project: str*, *namespace: Optional[str] = None*, *sort_by: str = ''*, *page_token: str = ''*, *filter_: str = ''*, *format_: Union[str, mlrun.api.schemas.pipeline.PipelinesFormat] = PipelinesFormat.metadata_only*, *page_size: Optional[int] = None*) → mlrun.api.schemas.pipeline.PipelinesOutput

Retrieve a list of KFP pipelines. This function can be invoked to get all pipelines from all projects, by specifying `project=*`, in which case pagination can be used and the various sorting and pagination properties can be applied. If a specific project is requested, then the pagination options cannot be used and pagination is not applied.

Parameters

- **project** – Project name. Can be * for query across all projects.
- **namespace** – Kubernetes namespace in which the pipelines are executing.
- **sort_by** – Field to sort the results by.
- **page_token** – Use for pagination, to retrieve next page.
- **filter** – Kubernetes filter to apply to the query, can be used to filter on specific object fields.
- **format** – Result format. Can be one of:
  - `full` - return the full objects.
  - `metadata_only` (default) - return just metadata of the pipelines objects.
  - `name_only` - return just the names of the pipeline objects.
- **page_size** – Size of a single page when applying pagination.

list_project_secret_keys(*project: str*, *provider: Union[str, mlrun.api.schemas.secret.SecretProviderName] = SecretProviderName.kubernetes*, *token: Optional[str] = None*) → mlrun.api.schemas.secret.SecretKeysData

Retrieve project-context secret keys from Vault or Kubernetes.

Note: This method for Vault functionality is currently in technical preview, and requires a HashiCorp Vault infrastructure properly set up and connected to the MLRun API server.

Parameters

- **project** – The project name.

- **provider** – The name of the secrets-provider to work with. Accepts a *SecretProviderName* enum.
- **token** – Vault token to use for retrieving secrets. Only in use if `provider` is `vault`. Must be a valid Vault token, with permissions to retrieve secrets of the project in question.

list_project_secrets(*project: str*, *token: Optional[str] = None*, *provider: Union[str, mlrun.api.schemas.secret.SecretProviderName] = SecretProviderName.kubernetes*, *secrets: Optional[List[str]] = None*) → mlrun.api.schemas.secret.SecretsData

Retrieve project-context secrets from Vault.

Note: This method for Vault functionality is currently in technical preview, and requires a HashiCorp Vault infrastructure properly set up and connected to the MLRun API server.

Parameters

- **project** – The project name.
- **token** – Vault token to use for retrieving secrets. Must be a valid Vault token, with permissions to retrieve secrets of the project in question.
- **provider** – The name of the secrets-provider to work with. Currently only `vault` is accepted.
- **secrets** – A list of secret names to retrieve. An empty list `[]` will retrieve all secrets assigned to this specific project. `kubernetes` provider only supports an empty list.

list_projects(*owner: Optional[str] = None*, *format_: Union[str, mlrun.api.schemas.project.ProjectsFormat] = ProjectsFormat.full*, *labels: Optional[List[str]] = None*, *state: Optional[Union[str, mlrun.api.schemas.project.ProjectState]] = None*) → List[Union[*mlrun.projects.project.MlrunProject*, str]]

Return a list of the existing projects, potentially filtered by specific criteria.

Parameters

- **owner** – List only projects belonging to this specific owner.
- **format** – Format of the results. Possible values are:
  - `full` (default value) - Return full project objects.
  - `name_only` - Return just the names of the projects.
- **labels** – Filter by labels attached to the project.
- **state** – Filter by project's state. Can be either `online` or `archived`.

list_runs(*name=None*, *uid=None*, *project=None*, *labels=None*, *state=None*, *sort=True*, *last=0*, *iter=False*, *start_time_from: Optional[datetime.datetime] = None*, *start_time_to: Optional[datetime.datetime] = None*, *last_update_time_from: Optional[datetime.datetime] = None*, *last_update_time_to: Optional[datetime.datetime] = None*, *partition_by: Optional[Union[mlrun.api.schemas.constants.RunPartitionByField, str]] = None*, *rows_per_partition: int = 1*, *partition_sort_by: Optional[Union[mlrun.api.schemas.constants.SortField, str]] = None*, *partition_order: Union[mlrun.api.schemas.constants.OrderType, str] = OrderType.desc*, *max_partitions: int = 0*) → mlrun.lists.RunList

Retrieve a list of runs, filtered by various options. Example:

```
runs = db.list_runs(name='download', project='iris', labels='owner=admin')
# If running in Jupyter, can use the .show() function to display the results
db.list_runs(name='', project=project_name).show()
```

Parameters

- **name** – Name of the run to retrieve.
- **uid** – Unique ID of the run.
- **project** – Project that the runs belongs to.
- **labels** – List runs that have a specific label assigned. Currently only a single label filter can be applied, otherwise result will be empty.
- **state** – List only runs whose state is specified.
- **sort** – Whether to sort the result according to their start time. Otherwise, results will be returned by their internal order in the DB (order will not be guaranteed).
- **last** – Deprecated - currently not used.
- **iter** – If `True` return runs from all iterations. Otherwise, return only runs whose `iter` is 0.
- **start_time_from** – Filter by run start time in `[start_time_from, start_time_to]`.
- **start_time_to** – Filter by run start time in `[start_time_from, start_time_to]`.
- **last_update_time_from** – Filter by run last update time in `(last_update_time_from, last_update_time_to)`.
- **last_update_time_to** – Filter by run last update time in `(last_update_time_from, last_update_time_to)`.
- **partition_by** – Field to group results by. Only allowed value is *name*. When *partition_by* is specified, the *partition_sort_by* parameter must be provided as well.
- **rows_per_partition** – How many top rows (per sorting defined by *partition_sort_by* and *partition_order*) to return per group. Default value is 1.
- **partition_sort_by** – What field to sort the results by, within each partition defined by *partition_by*. Currently the only allowed values are *created* and *updated*.
- **partition_order** – Order of sorting within partitions - *asc* or *desc*. Default is *desc*.
- **max_partitions** – Maximal number of partitions to include in the result. Default is *0* which means no limit.

list_runtime_resources(*project: Optional[str] = None*, *label_selector: Optional[str] = None*, *kind: Optional[str] = None*, *object_id: Optional[str] = None*, *group_by: Optional[mlrun.api.schemas.runtime_resource.ListRuntimeResourcesGroupByField] = None*) → Union[List[mlrun.api.schemas.runtime_resource.KindRuntimeResources], Dict[str, Dict[str, mlrun.api.schemas.runtime_resource.RuntimeResources]]]

List current runtime resources, which are usually (but not limited to) Kubernetes pods or CRDs. Function applies for runs of type *['dask', 'job', 'spark', 'remote-spark', 'mpijob']*, and will return per runtime kind a list of the runtime resources (which may have already completed their execution).

Parameters

- **project** – Get only runtime resources of a specific project, by default None, which will return only the projects you're authorized to see.
- **label_selector** – A label filter that will be passed to Kubernetes for filtering the results according to their labels.
- **kind** – The kind of runtime to query. May be one of *['dask', 'job', 'spark', 'remote-spark', 'mpijob']*
- **object_id** – The identifier of the mlrun object to query its runtime resources. for most function runtimes, runtime resources are per Run, for which the identifier is the Run's UID. For dask runtime, the runtime resources are per Function, for which the identifier is the Function's name.
- **group_by** – Object to group results by. Allowed values are *job* and *project*.

list_runtimes(*label_selector: Optional[str] = None*) → List

Deprecated use *list_runtime_resources()* instead

list_schedules(*project: str*, *name: Optional[str] = None*, *kind: Optional[mlrun.api.schemas.schedule.ScheduleKinds] = None*, *include_last_run: bool = False*) → mlrun.api.schemas.schedule.SchedulesOutput

Retrieve list of schedules of specific name or kind.

Parameters

- **project** – Project name.
- **name** – Name of schedule to retrieve. Can be omitted to list all schedules.
- **kind** – Kind of schedule objects to retrieve, can be either `job` or `pipeline`.
- **include_last_run** – Whether to return for each schedule returned also the results of the last run of that schedule.

patch_feature_set(*name*, *feature_set_update: dict*, *project=''*, *tag=None*, *uid=None*, *patch_mode: Union[str, mlrun.api.schemas.constants.PatchMode] = PatchMode.replace*)

Modify (patch) an existing *FeatureSet* object. The object is identified by its name (and project it belongs to), as well as optionally a `tag` or its `uid` (for versioned object). If both `tag` and `uid` are omitted then the object with tag `latest` is modified.

Parameters

- **name** – Name of the object to patch.
- **feature_set_update** – The modifications needed in the object. This parameter only has the changes in it, not a full object. Example:

```
feature_set_update = {"status": {"processed" : True}}
```

  Will apply the field `status.processed` to the existing object.
- **project** – Project which contains the modified object.
- **tag** – The tag of the object to modify.
- **uid** – uid of the object to modify.
- **patch_mode** – The strategy for merging the changes with the existing object. Can be either `replace` or `additive`.

patch_feature_vector(*name*, *feature_vector_update: dict*, *project=''*, *tag=None*, *uid=None*, *patch_mode: Union[str, mlrun.api.schemas.constants.PatchMode] = PatchMode.replace*)

Modify (patch) an existing `FeatureVector` object. The object is identified by its name (and project it belongs to), as well as optionally a `tag` or its `uid` (for versioned object). If both `tag` and `uid` are omitted then the object with tag `latest` is modified.

Parameters

- **name** – Name of the object to patch.
- **feature_vector_update** – The modifications needed in the object. This parameter only has the changes in it, not a full object.
- **project** – Project which contains the modified object.
- **tag** – The tag of the object to modify.
- **uid** – uid of the object to modify.
- **patch_mode** – The strategy for merging the changes with the existing object. Can be either `replace` or `additive`.

patch_project(*name: str*, *project: dict*, *patch_mode: Union[str, mlrun.api.schemas.constants.PatchMode] = PatchMode.replace*) → *mlrun.projects.project.MlrunProject*

Patch an existing project object.

Parameters

- **name** – Name of project to patch.
- **project** – The actual changes to the project object.
- **patch_mode** – The strategy for merging the changes with the existing object. Can be either `replace` or `additive`.

read_artifact(*key*, *tag=None*, *iter=None*, *project=''*)

Read an artifact, identified by its key, tag and iteration.

read_run(*uid*, *project=''*, *iter=0*)

Read the details of a stored run from the DB.

Parameters

- **uid** – The run's unique ID.
- **project** – Project name.
- **iter** – Iteration within a specific execution.

remote_builder(*func*, *with_mlrun*, *mlrun_version_specifier=None*, *skip_deployed=False*, *builder_env=None*)

Build the pod image for a function, for execution on a remote cluster. This is executed by the MLRun API server, and creates a Docker image out of the function provided and any specific build instructions provided within. This is a pre-requisite for remotely executing a function, unless using a pre-deployed image.

Parameters

- **func** – Function to build.
- **with_mlrun** – Whether to add MLRun package to the built package. This is not required if using a base image that already has MLRun in it.
- **mlrun_version_specifier** – Version of MLRun to include in the built image.
- **skip_deployed** – Skip the build if we already have an image for the function.

- **builder_env** – Kaniko builder pod env vars dict (for config/credentials)

remote_start(*func_url*) → mlrun.api.schemas.background_task.BackgroundTask
Execute a function remotely, Used for `dask` functions.

Parameters **func_url** – URL to the function to be executed.

Returns A BackgroundTask object, with details on execution process and its status.

remote_status(*project*, *name*, *kind*, *selector*)
Retrieve status of a function being executed remotely (relevant to `dask` functions).

Parameters

- **project** – The project of the function
- **name** – The name of the function
- **kind** – The kind of the function, currently `dask` is supported.
- **selector** – Selector clause to be applied to the Kubernetes status query to filter the results.

store_artifact(*key*, *artifact*, *uid*, *iter=None*, *tag=None*, *project=''*)
Store an artifact in the DB.

Parameters

- **key** – Identifying key of the artifact.
- **artifact** – The actual artifact to store.
- **uid** – A unique ID for this specific version of the artifact.
- **iter** – The task iteration which generated this artifact. If `iter` is not `None` the iteration will be added to the key provided to generate a unique key for the artifact of the specific iteration.
- **tag** – Tag of the artifact.
- **project** – Project that the artifact belongs to.

store_feature_set(*feature_set: Union[dict, mlrun.api.schemas.feature_store.FeatureSet, mlrun.feature_store.feature_set.FeatureSet]*, *name=None*, *project=''*, *tag=None*, *uid=None*, *versioned=True*) → dict
Save a *FeatureSet* object in the *mlrun* DB. The feature-set can be either a new object or a modification to existing object referenced by the params of the function.

Parameters

- **feature_set** – The *FeatureSet* to store.
- **project** – Name of project this feature-set belongs to.
- **tag** – The `tag` of the object to replace in the DB, for example `latest`.
- **uid** – The `uid` of the object to replace in the DB. If using this parameter, the modified object must have the same `uid` of the previously-existing object. This cannot be used for non-versioned objects.
- **versioned** – Whether to maintain versions for this feature-set. All versions of a versioned object will be kept in the DB and can be retrieved until explicitly deleted.

Returns The *FeatureSet* object (as dict).

store_feature_vector(*feature_vector: Union[dict, mlrun.api.schemas.feature_store.FeatureVector, mlrun.feature_store.feature_vector.FeatureVector]*, *name=None*, *project=''*, *tag=None*, *uid=None*, *versioned=True*) → dict

Store a *FeatureVector* object in the *mlrun* DB. The feature-vector can be either a new object or a modification to existing object referenced by the params of the function.

Parameters

- **feature_vector** – The *FeatureVector* to store.
- **project** – Name of project this feature-vector belongs to.
- **tag** – The `tag` of the object to replace in the DB, for example `latest`.
- **uid** – The `uid` of the object to replace in the DB. If using this parameter, the modified object must have the same `uid` of the previously-existing object. This cannot be used for non-versioned objects.
- **versioned** – Whether to maintain versions for this feature-vector. All versions of a versioned object will be kept in the DB and can be retrieved until explicitly deleted.

Returns The *FeatureVector* object (as dict).

store_function(*function*, *name*, *project=''*, *tag=None*, *versioned=False*)

Store a function object. Function is identified by its name and tag, and can be versioned.

store_log(*uid*, *project=''*, *body=None*, *append=False*)

Save a log persistently.

Parameters

- **uid** – Log unique ID
- **project** – Project name for which this log belongs
- **body** – The actual log to store
- **append** – Whether to append the log provided in `body` to an existing log with the same `uid` or to create a new log. If set to `False`, an existing log with same `uid` will be overwritten

store_marketplace_source(*source_name: str*, *source: Union[dict, mlrun.api.schemas.marketplace.IndexedMarketplaceSource]*)

Create or replace a marketplace source. For an example of the source format and explanation of the source order logic, please see *create_marketplace_source()*. This method can be used to modify the source itself or its order in the list of sources.

Parameters

- **source_name** – Name of the source object to modify/create. It must match the `source.metadata.name` parameter in the source itself.
- **source** – Source object to store in the database.

Returns The source object as stored in the DB.

store_project(*name: str*, *project: Union[dict, mlrun.projects.project.MlrunProject, mlrun.api.schemas.project.Project]*) → *mlrun.projects.project.MlrunProject*

Store a project in the DB. This operation will overwrite existing project of the same name if exists.

store_run(*struct*, *uid*, *project=''*, *iter=0*)

Store run details in the DB. This method is usually called from within other *mlrun* flows and not called directly by the user.

submit_job(*runspec*, *schedule: Optional[Union[str, mlrun.api.schemas.schedule.ScheduleCronTrigger]] = None*)
Submit a job for remote execution.

Parameters

- **runspec** – The runtime object spec (Task) to execute.
- **schedule** – Whether to schedule this job using a Cron trigger. If not specified, the job will be submitted immediately.

submit_pipeline(*project*, *pipeline*, *arguments=None*, *experiment=None*, *run=None*, *namespace=None*, *artifact_path=None*, *ops=None*, *ttl=None*)
Submit a KFP pipeline for execution.

Parameters

- **project** – The project of the pipeline
- **pipeline** – Pipeline function or path to .yaml/.zip pipeline file.
- **arguments** – A dictionary of arguments to pass to the pipeline.
- **experiment** – A name to assign for the specific experiment.
- **run** – A name for this specific run.
- **namespace** – Kubernetes namespace to execute the pipeline in.
- **artifact_path** – A path to artifacts used by this pipeline.
- **ops** – Transformers to apply on all ops in the pipeline.
- **ttl** – Set the TTL for the pipeline after its completion.

tag_artifacts(*artifacts: Union[List[mlrun.artifacts.base.Artifact], List[dict], mlrun.artifacts.base.Artifact, dict]*, *project: str*, *tag_name: str*, *replace: bool = False*)
Tag a list of artifacts.

Parameters

- **artifacts** – The artifacts to tag. Can be a list of `Artifact` objects or dictionaries, or a single object.
- **project** – Project which contains the artifacts.
- **tag_name** – The tag to set on the artifacts.
- **replace** – If True, replace existing tags, otherwise append to existing tags.

tag_objects(*project: str*, *tag_name: str*, *objects: Union[mlrun.api.schemas.tag.TagObjects, dict]*, *replace: bool = False*)
Tag a list of objects.

Parameters

- **project** – Project which contains the objects.
- **tag_name** – The tag to set on the objects.
- **objects** – The objects to tag.
- **replace** – Whether to replace the existing tags of the objects or to add the new tag to them.

trigger_migrations() → Optional[mlrun.api.schemas.background_task.BackgroundTask]
Trigger migrations (will do nothing if no migrations are needed) and wait for them to finish if actually triggered :returns: `BackgroundTask`.

update_run(*updates: dict*, *uid*, *project=''*, *iter=0*)
Update the details of a stored run in the DB.

update_schedule(*project: str*, *name: str*, *schedule: mlrun.api.schemas.schedule.ScheduleUpdate*)
Update an existing schedule, replace it with the details contained in the schedule object.

verify_authorization(*authorization_verification_input: mlrun.api.schemas.auth.AuthorizationVerificationInput*)
Verifies authorization for the provided action on the provided resource.

Parameters **authorization_verification_input** – Instance of `AuthorizationVerificationInput` that includes all the needed parameters for the auth verification

watch_log(*uid*, *project=''*, *watch=True*, *offset=0*)
Retrieve logs of a running process, and watch the progress of the execution until it completes. This method will print out the logs and continue to periodically poll for, and print, new logs as long as the state of the runtime which generates this log is either `pending` or `running`.

Parameters

- **uid** – The uid of the log object to watch.
- **project** – Project that the log belongs to.
- **watch** – If set to `True` will continue tracking the log as described above. Otherwise this function is practically equivalent to the *get_log()* function.
- **offset** – Minimal offset in the log to watch.

Returns The final state of the log being watched.

class mlrun.api.schemas.secret.**SecretProviderName**(*value*)
Bases: `str`, `enum.Enum`

Enum containing names of valid providers for secrets.

kubernetes = 'kubernetes'

vault = 'vault'

17.7 mlrun.execution

class mlrun.execution.**MLClientCtx**(*autocommit=False*, *tmp=''*, *log_stream=None*)
Bases: `object`

ML Execution Client Context

the context is generated and injected to the function using the `function.run()` or manually using the *get_or_create_ctx()* call and provides an interface to use run params, metadata, inputs, and outputs

base metadata include: uid, name, project, and iteration (for hyper params) users can set labels and annotations using *set_label()*, *set_annotation()* access parameters and secrets using *get_param()*, *get_secret()* access input data objects using *get_input()* store results, artifacts, and real-time metrics using the *log_result()*, *log_artifact()*, *log_dataset()* and *log_model()* methods

see doc for the individual params and methods

property annotations
dictionary with annotations (read-only)

artifact_subpath(**subpaths*)
subpaths under output path artifacts path

example:

```
data_path=context.artifact_subpath('data')
```

property artifacts
dictionary of artifacts (read-only)

commit(*message: str = ''*, *completed=True*)
save run state and optionally add a commit message

Parameters

- **message** – commit message to save in the run
- **completed** – mark run as completed

classmethod from_dict(*attrs: dict*, *rundb=''*, *autocommit=False*, *tmp=''*, *host=None*, *log_stream=None*, *is_api=False*, *update_db=True*)
create execution context from dict

get_cached_artifact(*key*)
return an logged artifact from cache (for potential updates)

get_child_context(*with_parent_params=False*, ***params*)
get child context (iteration)

allow sub experiments (epochs, hyper-param, ..) under a parent will create a new iteration, log_xx will update the child only use commit_children() to save all the children and specify the best run

example:

```
def handler(context: mlrun.MLClientCtx, data: mlrun.DataItem):
    df = data.as_df()
    best_accuracy = accuracy_sum = 0
    for param in param_list:
        with context.get_child_context(myparam=param) as child:
            accuracy = child_handler(child, df, **child.parameters)
            accuracy_sum += accuracy
            child.log_result('accuracy', accuracy)
            if accuracy > best_accuracy:
                child.mark_as_best()
                best_accuracy = accuracy

    context.log_result('avg_accuracy', accuracy_sum / len(param_list))
```

Parameters

- **params** – extra (or override) params to parent context
- **with_parent_params** – child will copy the parent parameters and add to them

Returns child context

get_dataitem(*url*, *secrets: Optional[dict] = None*)
get mlrun dataitem from url

example:

```
data = context.get_dataitem("s3://my-bucket/file.csv").as_df()
```

Parameters

- **url** – data-item uri/path
- **secrets** – additional secrets to use when accessing the data-item

get_input(*key: str*, *url: str = ''*)
get an input `DataItem` object, data objects have methods such as .get(), .download(), .url, .. to access the actual data

example:

```
data = context.get_input("my_data").get()
```

get_meta() → dict
Reserved for internal use

get_param(*key: str*, *default=None*)
get a run parameter, or use the provided default if not set

example:

```
p1 = context.get_param("p1", 0)
```

get_project_param(*key: str*, *default=None*)
get a parameter from the run's project's parameters

get_secret(*key: str*)
get a key based secret e.g. DB password from the context secrets can be specified when invoking a run through vault, files, env, ..

example:

```
access_key = context.get_secret("ACCESS_KEY")
```

get_store_resource(*url*, *secrets: Optional[dict] = None*)
get mlrun data resource (feature set/vector, artifact, item) from url

example:

```
feature_vector = context.get_store_resource("store://feature-vectors/default/
↪myvec")
dataset = context.get_store_resource("store://artifacts/default/mydata")
```

Parameters

- **url** – store resource uri/path, store://<type>/<project>/<name>:<version> types: artifacts | feature-sets | feature-vectors
- **secrets** – additional secrets to use when accessing the store resource

property in_path
default input path for data objects

property inputs
dictionary of input data items (read-only)

property iteration
child iteration index, for hyper parameters

kind = 'run'

property labels
dictionary with labels (read-only)

log_artifact(*item*, *body=None*, *local_path=None*, *artifact_path=None*, *tag=''*, *viewer=None*, *target_path=''*, *src_path=None*, *upload=None*, *labels=None*, *format=None*, *db_key=None*, ***kwargs*)
log an output artifact and optionally upload it to datastore

example:

```
context.log_artifact(
    "some-data",
    body=b"abc is 123",
    local_path="model.txt",
    labels={"framework": "xgboost"},
)
```

Parameters

- **item** – artifact key or artifact class ()
- **body** – will use the body as the artifact content
- **local_path** – path to the local file we upload, will also be use as the destination subpath (under "artifact_path")
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **tag** – version tag
- **viewer** – kubeflow viewer type
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **src_path** – deprecated, use local_path
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **format** – optional, format to use (e.g. csv, parquet, ..)
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_dataset(*key*, *df*, *tag=''*, *local_path=None*, *artifact_path=None*, *upload=True*, *labels=None*, *format=''*, *preview=None*, *stats=None*, *db_key=None*, *target_path=''*, *extra_data=None*, *label_column: Optional[str] = None*, ***kwargs*)
log a dataset artifact and optionally upload it to datastore

example:

```
raw_data = {
    "first_name": ["Jason", "Molly", "Tina", "Jake", "Amy"],
    "last_name": ["Miller", "Jacobson", "Ali", "Milner", "Cooze"],
    "age": [42, 52, 36, 24, 73],
    "testScore": [25, 94, 57, 62, 70],
}
df = pd.DataFrame(raw_data, columns=["first_name", "last_name", "age",
↪"testScore"])
context.log_dataset("mydf", df=df, stats=True)
```

Parameters

- **key** – artifact key
- **df** – dataframe object
- **label_column** – name of the label column (the one holding the target (y) values)
- **local_path** – path to the local dataframe file that exists locally. The given file extension will be used to save the dataframe to a file If the file exists, it will be uploaded to the datastore instead of the given df.
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **tag** – version tag
- **format** – optional, format to use (e.g. csv, parquet, ..)
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **preview** – number of lines to store as preview in the artifact metadata
- **stats** – calculate and store dataset stats in the artifact metadata
- **extra_data** – key/value list of extra files/charts to link with this dataset
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_iteration_results(*best*, *summary: list*, *task: dict*, *commit=False*)
Reserved for internal use

property log_level
get the logging level, e.g. 'debug', 'info', 'error'

log_metric(*key: str*, *value*, *timestamp=None*, *labels=None*)
TBD, log a real-time time-series metric

log_metrics(*keyvals: dict*, *timestamp=None*, *labels=None*)
TBD, log a set of real-time time-series metrics

log_model(*key*, *body=None*, *framework=''*, *tag=''*, *model_dir=None*, *model_file=None*, *algorithm=None*, *metrics=None*, *parameters=None*, *artifact_path=None*, *upload=True*, *labels=None*, *inputs: Optional[List[*mlrun.features.Feature*]] = None*, *outputs: Optional[List[*mlrun.features.Feature*]] = None*, *feature_vector: Optional[str] = None*, *feature_weights: Optional[list] = None*, *training_set=None*, *label_column: Optional[Union[str, list]] = None*, *extra_data=None*, *db_key=None*, ***kwargs*)

log a model artifact and optionally upload it to datastore

example:

```
context.log_model("model", body=dumps(model),
                  model_file="model.pkl",
                  metrics=context.results,
                  training_set=training_df,
                  label_column='label',
                  feature_vector=feature_vector_uri,
                  labels={"app": "fraud"})
```

Parameters

- **key** – artifact key or artifact class ()
- **body** – will use the body as the artifact content
- **model_file** – path to the local model file we upload (see also model_dir) or to a model file data url (e.g. http://host/path/model.pkl)
- **model_dir** – path to the local dir holding the model file and extra files
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **framework** – name of the ML framework
- **algorithm** – training algorithm name
- **tag** – version tag
- **metrics** – key/value dict of model metrics
- **parameters** – key/value dict of model parameters
- **inputs** – ordered list of model input features (name, type, ..)
- **outputs** – ordered list of model output/result elements (name, type, ..)
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **feature_vector** – feature store feature vector uri (store://feature-vectors/<project>/<name>[:tag])
- **feature_weights** – list of feature weights, one per input column
- **training_set** – training set dataframe, used to infer inputs & outputs
- **label_column** – which columns in the training set are the label (target) columns
- **extra_data** – key/value list of extra files/charts to link with this dataset value can be absolute path | relative path (to model dir) | bytes | artifact object
- **db_key** – the key to use in the artifact DB table, by default its run name + '_' + key db_key=False will not register it in the artifacts table

Returns artifact object

log_result(*key: str*, *value*, *commit=False*)
log a scalar result value

example:

```
context.log_result('accuracy', 0.85)
```

Parameters

- **key** – result key
- **value** – result value
- **commit** – commit (write to DB now vs wait for the end of the run)

log_results(*results: dict*, *commit=False*)
log a set of scalar result values

example:

```
context.log_results({'accuracy': 0.85, 'loss': 0.2})
```

Parameters

- **results** – key/value dict or results
- **commit** – commit (write to DB now vs wait for the end of the run)

property logger
built-in logger interface

example:

```
context.logger.info("started experiment..", param=5)
```

mark_as_best()
mark a child as the best iteration result, see .get_child_context()

property out_path
default output path for artifacts

property parameters
dictionary of run parameters (read-only)

property project
project name, runs can be categorized by projects

property results
dictionary of results (read-only)

set_annotation(*key: str*, *value*, *replace: bool = True*)
set/record a specific annotation

example:

```
context.set_annotation("comment", "some text")
```

set_hostname(*host: str*)
update the hostname, for internal use

set_label(*key: str*, *value*, *replace: bool = True*)
set/record a specific label

example:

```
context.set_label("framework", "sklearn")
```

set_logger_stream(*stream*)

set_state(*state: Optional[str] = None*, *error: Optional[str] = None*, *commit=True*)
modify and store the run state or mark an error

Parameters

- **state** – set run state
- **error** – error message (if exist will set the state to error)
- **commit** – will immediately update the state in the DB

property tag
run tag (uid or workflow id if exists)

to_dict()
convert the run context to a dictionary

to_json()
convert the run context to a json buffer

to_yaml()
convert the run context to a yaml buffer

property uid
Unique run id

update_artifact(*artifact_object*)
update an artifact object in the cache and the DB

update_child_iterations(*best_run=0*, *commit_children=False*, *completed=True*)
update children results in the parent, and optionally mark the best

Parameters

- **best_run** – marks the child iteration number (starts from 1)
- **commit_children** – commit all child runs to the db
- **completed** – mark children as completed

17.8 mlrun.feature_store

class mlrun.feature_store.**Entity**(*name: Optional[str] = None*, *value_type: Optional[mlrun.data_types.data_types.ValueType] = None*, *description: Optional[str] = None*, *labels: Optional[Dict[str, str]] = None*)

Bases: mlrun.model.ModelObj

data entity (index)

data entity (index key)

Parameters

- **name** – entity name

- **value_type** – type of the entity, e.g. ValueType.STRING, ValueType.INT
- **description** – test description of the entity
- **labels** – a set of key/value labels (tags)

class mlrun.feature_store.**Feature**(*value_type: Optional[str] = None*, *dims: Optional[List[int]] = None*, *description: Optional[str] = None*, *aggregate: Optional[bool] = None*, *name: Optional[str] = None*, *validator=None*, *default: Optional[str] = None*, *labels: Optional[Dict[str, str]] = None*)

Bases: mlrun.model.ModelObj

data feature

data feature

Features can be specified manually or inferred automatically (during ingest/preview)

Parameters

- **value_type** – type of the feature. Use the ValueType constants library e.g. ValueType.STRING, ValueType.INT
- **dims** – list of dimensions for vectors/tensors, e.g. [2, 2]
- **description** – text description of the feature
- **aggregate** – is it an aggregated value
- **name** – name of the feature
- **validator** – feature validation policy
- **default** – default value
- **labels** – a set of key/value labels (tags)

property validator

class mlrun.feature_store.**FeatureSet**(*name: Optional[str] = None*, *description: Optional[str] = None*, *entities: Optional[List[Union[*mlrun.features.Entity*, str]]] = None*, *timestamp_key: Optional[str] = None*, *engine: Optional[str] = None*, *label_column: Optional[str] = None*)

Bases: mlrun.model.ModelObj

Feature set object, defines a set of features and their data pipeline

Feature set object, defines a set of features and their data pipeline

example:

```
import mlrun.feature_store as fstore
ticks = fstore.FeatureSet("ticks", entities=["stock"], timestamp_key="timestamp")
fstore.ingest(ticks, df)
```

Parameters

- **name** – name of the feature set
- **description** – text description
- **entities** – list of entity (index key) names or Entity
- **timestamp_key** – timestamp column name
- **engine** – name of the processing engine (storey, pandas, or spark), defaults to storey

- **label_column** – name of the label column (the one holding the target (y) values)

add_aggregation(*column*, *operations*, *windows*, *period=None*, *name=None*, *step_name=None*, *after=None*, *before=None*, *state_name=None*, *emit_policy: Optional[storey.dtypes.EmitPolicy] = None*)

add feature aggregation rule

example:

```
myset.add_aggregation("ask", ["sum", "max"], "1h", "10m", name="asks")
```

Parameters

- **column** – name of column/field aggregate. Do not name columns starting with either _ or *aggr_*. They are reserved for internal use, and the data does not ingest correctly. When using the pandas engine, do not use spaces (` ) or periods (.`) in the column names; they cause errors in the ingestion.
- **operations** – aggregation operations, e.g. ['sum', 'std']
- **windows** – time windows, can be a single window, e.g. '1h', '1d', or a list of same unit windows e.g. ['1h', '6h'] windows are transformed to fixed windows or sliding windows depending whether period parameter provided.
  - Sliding window is fixed-size overlapping windows that slides with time. The window size determines the size of the sliding window and the period determines the step size to slide. Period must be integral divisor of the window size. If the period is not provided then fixed windows is used.
  - Fixed window is fixed-size, non-overlapping, gap-less window. The window is referred to as a tumbling window. In this case, each record on an in-application stream belongs to a specific window. It is processed only once (when the query processes the window to which the record belongs).
- **period** – optional, sliding window granularity, e.g. '20s' '10m' '3h' '7d'
- **name** – optional, aggregation name/prefix. Must be unique per feature set. If not passed, the column will be used as name.
- **step_name** – optional, graph step name
- **state_name** – *Deprecated* - use step_name instead
- **after** – optional, after which graph step it runs
- **before** – optional, comes before graph step
- **emit_policy** – optional, which emit policy to use when performing the aggregations. Use the derived classes of `storey.EmitPolicy`. The default is to emit every period for Spark engine and emit every event for storey. Currently the only other supported option is to use `emit_policy=storey.EmitEveryEvent()` when using the Spark engine to emit every event

add_entity(*name: str*, *value_type: Optional[mlrun.data_types.data_types.ValueType] = None*, *description: Optional[str] = None*, *labels: Optional[Dict[str, str]] = None*)

add/set an entity (dataset index)

example:

```
import mlrun.feature_store as fstore

ticks = fstore.FeatureSet("ticks",
                entities=["stock"],
                timestamp_key="timestamp")
ticks.add_entity("country",
                mlrun.data_types.ValueType.STRING,
                description="stock country")
ticks.add_entity("year", mlrun.data_types.ValueType.INT16)
ticks.save()
```

Parameters

- **name** – entity name
- **value_type** – type of the entity (default to ValueType.STRING)
- **description** – description of the entity
- **labels** – label tags dict

add_feature(*feature:* mlrun.features.Feature, *name=None*)
add/set a feature

example:

```
import mlrun.feature_store as fstore
from mlrun.features import Feature

ticks = fstore.FeatureSet("ticks",
                entities=["stock"],
                timestamp_key="timestamp")
ticks.add_feature(Feature(value_type=mlrun.data_types.ValueType.STRING,
                description="client consistency"),"ABC01")
ticks.add_feature(Feature(value_type=mlrun.data_types.ValueType.FLOAT,
                description="client volatility"),"SAB")
ticks.save()
```

Parameters

- **feature** – setting of Feature
- **name** – feature name

property fullname: str
{tag}]

Type full name in the form {project}/{name}[

get_stats_table()
get feature statistics table (as dataframe)

get_target_path(*name=None*)
get the url/path for an offline or specified data target

property graph
feature set transformation graph/DAG

has_valid_source()
check if object's spec has a valid (non empty) source definition

kind = 'FeatureSet'

link_analysis(*name*, *uri*)
add a linked file/artifact (chart, data, ..)

property metadata: mlrun.model.VersionedObjMetadata

plot(*filename=None*, *format=None*, *with_targets=False*, ***kw*)
generate graphviz plot

purge_targets(*target_names: Optional[List[str]] = None*, *silent: bool = False*)
Delete data of specific targets :param target_names: List of names of targets to delete (default: delete all ingested targets) :param silent: Fail silently if target doesn't exist in featureset status

reload(*update_spec=True*)
reload/sync the feature vector status and spec from the DB

save(*tag=''*, *versioned=False*)
save to mlrun db

set_targets(*targets=None*, *with_defaults=True*, *default_final_step=None*, *default_final_state=None*)
set the desired target list or defaults

Parameters

- **targets** – list of target type names ('csv', 'nosql', ..) or target objects CSVTarget(), ParquetTarget(), NoSqlTarget(), StreamTarget(), ..
- **with_defaults** – add the default targets (as defined in the central config)
- **default_final_step** – the final graph step after which we add the target writers, used when the graph branches and the end cant be determined automatically
- **default_final_state** – *Deprecated* - use default_final_step instead

property spec: *mlrun.feature_store.feature_set.FeatureSetSpec*

property status: *mlrun.feature_store.feature_set.FeatureSetStatus*

to_dataframe(*columns=None*, *df_module=None*, *target_name=None*, *start_time=None*, *end_time=None*, *time_column=None*, ***kwargs*)
return featureset (offline) data as dataframe

Parameters

- **columns** – list of columns to select (if not all)
- **df_module** – py module used to create the DataFrame (pd for Pandas, dd for Dask, ..)
- **target_name** – select a specific target (material view)
- **start_time** – filter by start time
- **end_time** – filter by end time
- **time_column** – specify the time column name in the file
- **kwargs** – additional reader (csv, parquet, ..) args

Returns DataFrame

update_targets_for_ingest(*targets: List[*mlrun.model.DataTargetBase*]*, *overwrite: Optional[bool] = None*)

property uri
fully qualified feature set uri

class mlrun.feature_store.**FeatureVector**(*name=None*, *features=None*, *label_feature=None*, *description=None*, *with_indexes=None*)

Bases: mlrun.model.ModelObj

Feature vector, specify selected features, their metadata and material views

Feature vector, specify selected features, their metadata and material views

example:

```
import mlrun.feature_store as fstore
features = ["quotes.bid", "quotes.asks_sum_5h as asks_5h", "stocks.*"]
vector = fstore.FeatureVector("my-vec", features)

# get the vector as a dataframe
df = fstore.get_offline_features(vector).to_dataframe()

# return an online/real-time feature service
svc = fs.get_online_feature_service(vector, impute_policy={"*": "$mean"})
resp = svc.get([{"stock": "GOOG"}])
```

Parameters

- **name** – List of names of targets to delete (default: delete all ingested targets)
- **features** – list of feature to collect to this vector. Format [<project>/]<feature_set>.<feature_name or *> [as <alias>]
- **label_feature** – feature name to be used as label data
- **description** – text description of the vector
- **with_indexes** – whether to keep the entity and timestamp columns in the response

get_feature_aliases()

get_stats_table()
get feature statistics table (as dataframe)

get_target_path(*name=None*)

kind = 'FeatureVector'

link_analysis(*name*, *uri*)
add a linked file/artifact (chart, data, ..)

property metadata: mlrun.model.VersionedObjMetadata

parse_features(*offline=True*, *update_stats=False*)
parse and validate feature list (from vector) and add metadata from feature sets

:returns feature_set_objects: cache of used feature set objects feature_set_fields: list of field (name, alias) per featureset

reload(*update_spec=True*)
reload/sync the feature set status and spec from the DB

save(*tag=''*, *versioned=False*)
save to mlrun db

property spec: mlrun.feature_store.feature_vector.FeatureVectorSpec

property status: mlrun.feature_store.feature_vector.FeatureVectorStatus

to_dataframe(*df_module=None*, *target_name=None*)
return feature vector (offline) data as dataframe

property uri
fully qualified feature vector uri

class mlrun.feature_store.FixedWindowType(*value*)
Bases: enum.Enum

An enumeration.

CurrentOpenWindow = 1

LastClosedWindow = 2

to_qbk_fixed_window_type()

class mlrun.feature_store.OfflineVectorResponse(*merger*)
Bases: object

get_offline_features response object

property status
vector prep job status (ready, running, error)

to_csv(*target_path*, ***kw*)
return results as csv file

to_dataframe(*to_pandas=True*)
return result as dataframe

to_parquet(*target_path*, ***kw*)
return results as parquet file

class mlrun.feature_store.OnlineVectorService(*vector*, *graph*, *index_columns*, *impute_policy: Optional[dict] = None*)
Bases: object

get_online_feature_service response object

close()
terminate the async loop

get(*entity_rows: List[Union[dict, list]]*, *as_list=False*)
get feature vector given the provided entity inputs

take a list of input vectors/rows and return a list of enriched feature vectors each input and/or output vector can be a list of values or a dictionary of field names and values, to return the vector as a list of values set the *as_list* to True.

if the input is a list of list (vs a list of dict), the values in the list will correspond to the index/entity values, i.e. [["GOOG"], ["MSFT"]] means "GOOG" and "MSFT" are the index/entity fields.

example:

```
# accept list of dict, return list of dict
svc = fs.get_online_feature_service(vector)
resp = svc.get([{"name": "joe"}, {"name": "mike"}])

# accept list of list, return list of list
```

(continues on next page)

(continued from previous page)

```
svc = fs.get_online_feature_service(vector, as_list=True)
resp = svc.get([["joe"], ["mike"]])
```

Parameters

- **entity_rows** – list of list/dict with input entity data/rows
- **as_list** – return a list of list (list input is required by many ML frameworks)

initialize()
internal, init the feature service and prep the imputing logic

property status
vector merger function status (ready, running, error)

class mlrun.feature_store.**RunConfig**(*function: Optional[Union[str, mlrun.runtimes.function_reference.FunctionReference, mlrun.runtimes.base.BaseRuntime]] = None*, *local: Optional[bool] = None*, *image: Optional[str] = None*, *kind: Optional[str] = None*, *handler: Optional[str] = None*, *parameters: Optional[dict] = None*, *watch: Optional[bool] = None*, *owner=None*, *credentials: Optional[mlrun.model.Credentials] = None*, *code: Optional[str] = None*, *requirements: Optional[Union[str, List[str]]] = None*, *extra_spec: Optional[dict] = None*, *auth_info=None*)

Bases: object

class for holding function and run specs for jobs and serving functions

class for holding function and run specs for jobs and serving functions

when running feature ingestion or merging tasks we use the RunConfig class to pass the desired function and job configuration. the apply() method is used to set resources like volumes, the with_secret() method adds secrets

Most attributes are optional, if not specified a proper default value will be set

examples:

```
# config for local run emulation
config = RunConfig(local=True)

# config for using empty/default code
config = RunConfig()

# config for using .py/.ipynb file with image and extra package requirements
config = RunConfig("mycode.py", image="mlrun/mlrun", requirements=["spacy"])

# config for using function object
function = mlrun.import_function("hub://some_function")
config = RunConfig(function)
```

Parameters

- **function** – this can be function uri or function object or path to function code (.py/.ipynb) or a FunctionReference the function define the code, dependencies, and resources
- **local** – use True to simulate local job run or mock service
- **image** – function container image

- **kind** – function runtime kind (job, serving, spark, ..), required when function points to code
- **handler** – the function handler to execute (for jobs or nuclio)
- **parameters** – job parameters
- **watch** – in batch jobs will wait for the job completion and print job logs to the console
- **owner** – job owner
- **credentials** – job credentials
- **code** – function source code (as string)
- **requirements** – python requirements file path or list of packages
- **extra_spec** – additional dict with function spec fields/values to add to the function
- **auth_info** – authentication info. *For internal use* when running on server

apply(*modifier*)
apply a modifier to add/set function resources like volumes

example:

```
run_config.apply(mlrun.platforms.auto_mount())
```

copy()

property function

to_function(*default_kind=None*, *default_image=None*)
internal, generate function object

with_secret(*kind*, *source*)
register a secrets source (file, env or dict)

read secrets from a source provider to be used in jobs, example:

```
run_config.with_secrets('file', 'file.txt')
run_config.with_secrets('inline', {'key': 'val'})
run_config.with_secrets('env', 'ENV1,ENV2')
run_config.with_secrets('vault', ['secret1', 'secret2'...])
```

Parameters

- **kind** – secret type (file, inline, env, vault)
- **source** – secret data or link (see example)

Returns This (self) object

mlrun.feature_store.**delete_feature_set**(*name*, *project=''*, *tag=None*, *uid=None*, *force=False*)
Delete a *FeatureSet* object from the DB. :param name: Name of the object to delete :param project: Name of the object's project :param tag: Specific object's version tag :param uid: Specific object's uid :param force: Delete feature set without purging its targets

If tag or uid are specified, then just the version referenced by them will be deleted. Using both is not allowed. If none are specified, then all instances of the object whose name is name will be deleted.

mlrun.feature_store.**delete_feature_vector**(*name*, *project=''*, *tag=None*, *uid=None*)
Delete a *FeatureVector* object from the DB. :param name: Name of the object to delete :param project: Name of the object's project :param tag: Specific object's version tag :param uid: Specific object's uid

If `tag` or `uid` are specified, then just the version referenced by them will be deleted. Using both is not allowed. If none are specified, then all instances of the object whose name is `name` will be deleted.

mlrun.feature_store.**deploy_ingestion_service**(*featureset: Union[mlrun.feature_store.feature_set.FeatureSet, str]*, *source: Optional[mlrun.model.DataSource] = None*, *targets: Optional[List[mlrun.model.DataTargetBase]] = None*, *name: Optional[str] = None*, *run_config: Optional[mlrun.feature_store.common.RunConfig] = None*, *verbose=False*)
Start real-time ingestion service using nuclio function

Deploy a real-time function implementing feature ingestion pipeline the source maps to Nuclio event triggers (http, kafka, v3io stream, etc.)

the *run_config* parameter allow specifying the function and job configuration, see: *RunConfig*

example:

```
source = HTTPSource()
func = mlrun.code_to_function("ingest", kind="serving").apply(mount_v3io())
config = RunConfig(function=func)
fs.deploy_ingestion_service(my_set, source, run_config=config)
```

Parameters

- **featureset** – feature set object or uri
- **source** – data source object describing the online or offline source
- **targets** – list of data target objects
- **name** – name for the job/function
- **run_config** – service runtime configuration (function object/uri, resources, etc..)
- **verbose** – verbose log

mlrun.feature_store.**get_feature_set**(*uri*, *project=None*)
get feature set object from the db

Parameters

- **uri** – a feature set uri({project}/{name}[:version])
- **project** – project name if not specified in uri or not using the current/default

mlrun.feature_store.**get_feature_vector**(*uri*, *project=None*)
get feature vector object from the db

Parameters

- **uri** – a feature vector uri({project}/{name}[:version])
- **project** – project name if not specified in uri or not using the current/default

mlrun.feature_store.**get_offline_features**(*feature_vector: Union[str, mlrun.feature_store.feature_vector.FeatureVector]*, *entity_rows=None*, *entity_timestamp_column: Optional[str] = None*, *target: Optional[mlrun.model.DataTargetBase] = None*, *run_config: Optional[mlrun.feature_store.common.RunConfig] = None*, *drop_columns: Optional[List[str]] = None*, *start_time: Optional[Union[str, pandas._libs.tslibs.timestamps.Timestamp]] = None*, *end_time: Optional[Union[str, pandas._libs.tslibs.timestamps.Timestamp]] = None*, *with_indexes: bool = False*, *update_stats: bool = False*, *engine: Optional[str] = None*, *engine_args: Optional[dict] = None*, *query: Optional[str] = None*) → *mlrun.feature_store.feature_vector.OfflineVectorResponse*

retrieve offline feature vector results

specify a feature vector object/uri and retrieve the desired features, their metadata and statistics. returns *OfflineVectorResponse*, results can be returned as a dataframe or written to a target

The start_time and end_time attributes allow filtering the data to a given time range, they accept string values or pandas *Timestamp* objects, string values can also be relative, for example: "now", "now - 1d2h", "now+5m", where a valid pandas Timedelta string follows the verb "now", for time alignment you can use the verb "floor" e.g. "now -1d floor 1H" will align the time to the last hour (the floor string is passed to pandas.Timestamp.floor(), can use D, H, T, S for day, hour, min, sec alignment). Another option to filter the data is by the *query* argument - can be seen in the example. example:

```
features = [
    "stock-quotes.bid",
    "stock-quotes.asks_sum_5h",
    "stock-quotes.ask as mycol",
    "stocks.*",
]
vector = FeatureVector(features=features)
resp = get_offline_features(
    vector, entity_rows=trades, entity_timestamp_column="time", query="ticker in [
↪'GOOG'] and bid>100"
)
print(resp.to_dataframe())
print(vector.get_stats_table())
resp.to_parquet("./out.parquet")
```

Parameters

- **feature_vector** – feature vector uri or FeatureVector object. passing feature vector obj requires update permissions
- **entity_rows** – dataframe with entity rows to join with
- **target** – where to write the results to
- **drop_columns** – list of columns to drop from the final result
- **entity_timestamp_column** – timestamp column name in the entity rows dataframe
- **run_config** – function and/or run configuration see *RunConfig*

- **start_time** – datetime, low limit of time needed to be filtered. Optional. entity_timestamp_column must be passed when using time filtering.
- **end_time** – datetime, high limit of time needed to be filtered. Optional. entity_timestamp_column must be passed when using time filtering.
- **with_indexes** – return vector with index columns and timestamp_key from the feature sets (default False)
- **update_stats** – update features statistics from the requested feature sets on the vector. Default is False.
- **engine** – processing engine kind ("local", "dask", or "spark")
- **engine_args** – kwargs for the processing engine
- **query** – The query string used to filter rows

mlrun.feature_store.**get_online_feature_service**(*feature_vector: Union[str, mlrun.feature_store.feature_vector.FeatureVector]*, *run_config: Optional[mlrun.feature_store.common.RunConfig] = None*, *fixed_window_type: mlrun.feature_store.feature_vector.FixedWindowType = FixedWindowType.LastClosedWindow*, *impute_policy: Optional[dict] = None*, *update_stats: bool = False*) → *mlrun.feature_store.feature_vector.OnlineVectorService*

initialize and return online feature vector service api, returns *OnlineVectorService*

Usage There are two ways to use the function:

1. As context manager

   Example:

```
with get_online_feature_service(vector_uri) as svc:
    resp = svc.get([{"ticker": "GOOG"}, {"ticker": "MSFT"}])
    print(resp)
    resp = svc.get([{"ticker": "AAPL"}], as_list=True)
    print(resp)
```

   Example with imputing:

```
with get_online_feature_service(vector_uri, impute_policy={
↪"*": "$mean", "amount": 0)) as svc:
    resp = svc.get([{"id": "C123487"}])
```

2. as simple function, note that in that option you need to close the session.

   Example:

```
svc = get_online_feature_service(vector_uri)
try:
    resp = svc.get([{"ticker": "GOOG"}, {"ticker": "MSFT"}])
    print(resp)
    resp = svc.get([{"ticker": "AAPL"}], as_list=True)
    print(resp)
```

(continues on next page)

(continued from previous page)

```
finally:
    svc.close()
```

Example with imputing:

```
svc = get_online_feature_service(vector_uri, impute_policy={
↪"*": "$mean", "amount": 0))
try:
    resp = svc.get([{"id": "C123487"}])
except Exception as e:
    handling exception...
finally:
    svc.close()
```

Parameters

- **feature_vector** – feature vector uri or FeatureVector object. passing feature vector obj requires update permissions
- **run_config** – function and/or run configuration for remote jobs/services
- **impute_policy** – a dict with *impute_policy* per feature, the dict key is the feature name and the dict value indicate which value will be used in case the feature is NaN/empty, the replaced value can be fixed number for constants or $mean, $max, $min, $std, $count for statistical values. "*" is used to specify the default for all features, example: *{"*": "$mean"}*
- **fixed_window_type** – determines how to query the fixed window values which were previously inserted by ingest
- **update_stats** – update features statistics from the requested feature sets on the vector. Default: False.

mlrun.feature_store.**ingest**(*featureset: Optional[Union[*mlrun.feature_store.feature_set.FeatureSet*, str]] = None*, *source=None*, *targets: Optional[List[*mlrun.model.DataTargetBase*]] = None*, *namespace=None*, *return_df: bool = True*, *infer_options: mlrun.data_types.data_types.InferOptions = 63*, *run_config: Optional[*mlrun.feature_store.common.RunConfig*] = None*, *mlrun_context=None*, *spark_context=None*, *overwrite=None*) → Optional[pandas.core.frame.DataFrame]

Read local DataFrame, file, URL, or source into the feature store Ingest reads from the source, run the graph transformations, infers metadata and stats and writes the results to the default of specified targets

when targets are not specified data is stored in the configured default targets (will usually be NoSQL for real-time and Parquet for offline).

the *run_config* parameter allow specifying the function and job configuration, see: *RunConfig*

example:

```
stocks_set = FeatureSet("stocks", entities=[Entity("ticker")])
stocks = pd.read_csv("stocks.csv")
df = ingest(stocks_set, stocks, infer_options=fstore.InferOptions.default())

# for running as remote job
config = RunConfig(image='mlrun/mlrun')
df = ingest(stocks_set, stocks, run_config=config)
```

(continues on next page)

(continued from previous page)

```
# specify source and targets
source = CSVSource("mycsv", path="measurements.csv")
targets = [CSVTarget("mycsv", path="./mycsv.csv")]
ingest(measurements, source, targets)
```

Parameters

- **featureset** – feature set object or featureset.uri. (uri must be of a feature set that is in the DB, call *.save()* if it's not)
- **source** – source dataframe or other sources (e.g. parquet source see: *ParquetSource* and other classes in mlrun.datastore with suffix Source)
- **targets** – optional list of data target objects
- **namespace** – namespace or module containing graph classes
- **return_df** – indicate if to return a dataframe with the graph results
- **infer_options** – schema and stats infer options
- **run_config** – function and/or run configuration for remote jobs, see *RunConfig*
- **mlrun_context** – mlrun context (when running as a job), for internal use !
- **spark_context** – local spark session for spark ingestion, example for creating the spark context: *spark = SparkSession.builder.appName("Spark function").getOrCreate()* For remote spark ingestion, this should contain the remote spark service name
- **overwrite** – delete the targets' data prior to ingestion (default: True for non scheduled ingest - deletes the targets that are about to be ingested. False for scheduled ingest - does not delete the target)

Returns if return_df is True, a dataframe will be returned based on the graph

mlrun.feature_store.**preview**(*featureset:* mlrun.feature_store.feature_set.FeatureSet, *source*, *entity_columns: Optional[list] = None*, *timestamp_key: Optional[str] = None*, *namespace=None*, *options: Optional[mlrun.data_types.data_types.InferOptions] = None*, *verbose: bool = False*, *sample_size: Optional[int] = None*) → pandas.core.frame.DataFrame

run the ingestion pipeline with local DataFrame/file data and infer features schema and stats

example:

```
quotes_set = FeatureSet("stock-quotes", entities=[Entity("ticker")])
quotes_set.add_aggregation("ask", ["sum", "max"], ["1h", "5h"], "10m")
quotes_set.add_aggregation("bid", ["min", "max"], ["1h"], "10m")
df = preview(
    quotes_set,
    quotes_df,
    entity_columns=["ticker"],
    timestamp_key="time",
)
```

Parameters

- **featureset** – feature set object or uri

- **source** – source dataframe or csv/parquet file path
- **entity_columns** – list of entity (index) column names
- **timestamp_key** – timestamp column name
- **namespace** – namespace or module containing graph classes
- **options** – schema and stats infer options (`InferOptions`)
- **verbose** – verbose log
- **sample_size** – num of rows to sample from the dataset (for large datasets)

class mlrun.feature_store.feature_set.**FeatureSetSpec**(*owner=None*, *description=None*, *entities=None*, *features=None*, *partition_keys=None*, *timestamp_key=None*, *label_column=None*, *relations=None*, *source=None*, *targets=None*, *graph=None*, *function=None*, *analysis=None*, *engine=None*, *output_path=None*)

Feature set spec object, defines the feature-set's configuration.

Warning: This class should not be modified directly. It is managed by the parent feature-set object or using feature-store APIs. Modifying the spec manually may result in unpredictable behaviour.

Parameters

- **description** – text description (copied from parent feature-set)
- **entities** – list of entity (index key) names or `Entity`
- **features** – list of features - `Feature`
- **partition_keys** – list of fields to partition results by (other than the default timestamp key)
- **timestamp_key** – timestamp column name
- **label_column** – name of the label column (the one holding the target (y) values)
- **targets** – list of data targets
- **graph** – the processing graph
- **function** – MLRun runtime to execute the feature-set in
- **engine** – name of the processing engine (storey, pandas, or spark), defaults to storey
- **output_path** – default location where to store results (defaults to MLRun's artifact path)

class mlrun.feature_store.feature_set.**FeatureSetStatus**(*state=None*, *targets=None*, *stats=None*, *preview=None*, *function_uri=None*, *run_uri=None*)

Feature set status object, containing the current feature-set's status.

Warning: This class should not be modified directly. It is managed by the parent feature-set object or using feature-store APIs. Modifying the status manually may result in unpredictable behaviour.

Parameters

- **state** – object's current state
- **targets** – list of the data targets used in the last ingestion operation
- **stats** – feature statistics calculated in the last ingestion (if stats calculation was requested)
- **preview** – preview of the feature-set contents (if preview generation was requested)
- **function_uri** – function used to execute the feature-set graph
- **run_uri** – last run used for ingestion

class mlrun.feature_store.steps.**DateExtractor**(*parts: Union[Dict[str, str], List[str]]*, *timestamp_col: Optional[str] = None*, ***kwargs*)

Date Extractor allows you to extract a date-time component

Date Extractor extract a date-time component into new columns

The extracted date part will appear as *<timestamp_col>_<date_part>* feature.

Supports part values:

- asm8: Return numpy datetime64 format in nanoseconds.
- day_of_week: Return day of the week.
- day_of_year: Return the day of the year.
- dayofweek: Return day of the week.
- dayofyear: Return the day of the year.
- days_in_month: Return the number of days in the month.
- daysinmonth: Return the number of days in the month.
- freqstr: Return the total number of days in the month.
- is_leap_year: Return True if year is a leap year.
- is_month_end: Return True if date is last day of month.
- is_month_start: Return True if date is first day of month.
- is_quarter_end: Return True if date is last day of the quarter.
- is_quarter_start: Return True if date is first day of the quarter.
- is_year_end: Return True if date is last day of the year.
- is_year_start: Return True if date is first day of the year.
- quarter: Return the quarter of the year.
- tz: Alias for tzinfo.
- week: Return the week number of the year.
- weekofyear: Return the week number of the year.

example:

```
# (taken from the fraud-detection end-to-end feature store demo)
# Define the Transactions FeatureSet
transaction_set = fs.FeatureSet("transactions",
                                entities=[fs.Entity("source")],
```

(continues on next page)

(continued from previous page)

```
                                timestamp_key='timestamp',
                                description="transactions feature set")

# Get FeatureSet computation graph
transaction_graph = transaction_set.graph

# Add the custom `DateExtractor` step
# to the computation graph
transaction_graph.to(
        class_name='DateExtractor',
        name='Extract Dates',
        parts = ['hour', 'day_of_week'],
        timestamp_col = 'timestamp',
    )
```

Parameters

- **parts** – list of pandas style date-time parts you want to extract.
- **timestamp_col** – The name of the column containing the timestamps to extract from, by default "timestamp"

__init__(*parts: Union[Dict[str, str], List[str]]*, *timestamp_col: Optional[str] = None*, ***kwargs*)

Date Extractor extract a date-time component into new columns

The extracted date part will appear as *<timestamp_col>_<date_part>* feature.

Supports part values:

- asm8: Return numpy datetime64 format in nanoseconds.
- day_of_week: Return day of the week.
- day_of_year: Return the day of the year.
- dayofweek: Return day of the week.
- dayofyear: Return the day of the year.
- days_in_month: Return the number of days in the month.
- daysinmonth: Return the number of days in the month.
- freqstr: Return the total number of days in the month.
- is_leap_year: Return True if year is a leap year.
- is_month_end: Return True if date is last day of month.
- is_month_start: Return True if date is first day of month.
- is_quarter_end: Return True if date is last day of the quarter.
- is_quarter_start: Return True if date is first day of the quarter.
- is_year_end: Return True if date is last day of the year.
- is_year_start: Return True if date is first day of the year.
- quarter: Return the quarter of the year.
- tz: Alias for tzinfo.

- week: Return the week number of the year.
- weekofyear: Return the week number of the year.

example:

```
# (taken from the fraud-detection end-to-end feature store demo)
# Define the Transactions FeatureSet
transaction_set = fs.FeatureSet("transactions",
                                entities=[fs.Entity("source")],
                                timestamp_key='timestamp',
                                description="transactions feature set")

# Get FeatureSet computation graph
transaction_graph = transaction_set.graph

# Add the custom `DateExtractor` step
# to the computation graph
transaction_graph.to(
        class_name='DateExtractor',
        name='Extract Dates',
        parts = ['hour', 'day_of_week'],
        timestamp_col = 'timestamp',
    )
```

Parameters

- **parts** – list of pandas style date-time parts you want to extract.
- **timestamp_col** – The name of the column containing the timestamps to extract from, by default "timestamp"

class mlrun.feature_store.steps.**DropFeatures**(*features: List[str]*, ***kwargs*)

Drop all the features from feature list

Parameters **features** – string list of the features names to drop

example:

```
feature_set = fs.FeatureSet("fs-new",
                            entities=[fs.Entity("id")],
                            description="feature set",
                            engine="pandas",
                            )
# Pre-processing grpah steps
feature_set.graph.to(DropFeatures(features=["age"]))
df_pandas = fs.ingest(feature_set, data)
```

__init__(*features: List[str]*, ***kwargs*)

Drop all the features from feature list

Parameters **features** – string list of the features names to drop

example:

```
feature_set = fs.FeatureSet("fs-new",
                            entities=[fs.Entity("id")],
```

(continues on next page)

(continued from previous page)

```
                            description="feature set",
                            engine="pandas",
                            )
# Pre-processing grpah steps
feature_set.graph.to(DropFeatures(features=["age"]))
df_pandas = fs.ingest(feature_set, data)
```

class mlrun.feature_store.steps.**FeaturesetValidator**(*featureset=None*, *columns=None*, *name=None*, ***kwargs*)

Validate feature values according to the feature set validation policy

Validate feature values according to the feature set validation policy

Parameters

- **featureset** – feature set uri (or "." for current feature set pipeline)
- **columns** – names of the columns/fields to validate
- **name** – step name
- **kwargs** – optional kwargs (for storey)

__init__(*featureset=None*, *columns=None*, *name=None*, ***kwargs*)

Validate feature values according to the feature set validation policy

Parameters

- **featureset** – feature set uri (or "." for current feature set pipeline)
- **columns** – names of the columns/fields to validate
- **name** – step name
- **kwargs** – optional kwargs (for storey)

class mlrun.feature_store.steps.**Imputer**(*method: str = 'avg'*, *default_value=None*, *mapping: Optional[Dict[str, Any]] = None*, ***kwargs*)

Replace None values with default values

Parameters

- **method** – for future use
- **default_value** – default value if not specified per column
- **mapping** – a dict of per column default value
- **kwargs** – optional kwargs (for storey)

__init__(*method: str = 'avg'*, *default_value=None*, *mapping: Optional[Dict[str, Any]] = None*, ***kwargs*)

Replace None values with default values

Parameters

- **method** – for future use
- **default_value** – default value if not specified per column
- **mapping** – a dict of per column default value
- **kwargs** – optional kwargs (for storey)

class mlrun.feature_store.steps.**MLRunStep**(***kwargs*)

Abstract class for mlrun step. Can be used in pandas/storey feature set ingestion

__init__(***kwargs*)
Abstract class for mlrun step. Can be used in pandas/storey feature set ingestion

do(*event*)
This method defines the do method of this class according to the first event type.

class mlrun.feature_store.steps.**MapValues**(*mapping: Dict[str, Dict[str, Any]]*, *with_original_features: bool = False*, *suffix: str = 'mapped'*, ***kwargs*)
Map column values to new values

Map column values to new values

example:

```
# replace the value "U" with '0' in the age column
graph.to(MapValues(mapping={'age': {'U': '0'}}, with_original_features=True))

# replace integers, example
graph.to(MapValues(mapping={'not': {0: 1, 1: 0}}))

# replace by range, use -inf and inf for extended range
graph.to(MapValues(mapping={'numbers': {'ranges': {'negative': [-inf, 0], 'positive
↪': [0, inf]}}}))
```

Parameters

- **mapping** – a dict with entry per column and the associated old/new values map
- **with_original_features** – set to True to keep the original features
- **suffix** – the suffix added to the column name <column>_<suffix> (default is "mapped")
- **kwargs** – optional kwargs (for storey)

__init__(*mapping: Dict[str, Dict[str, Any]]*, *with_original_features: bool = False*, *suffix: str = 'mapped'*, ***kwargs*)
Map column values to new values

example:

```
# replace the value "U" with '0' in the age column
graph.to(MapValues(mapping={'age': {'U': '0'}}, with_original_features=True))

# replace integers, example
graph.to(MapValues(mapping={'not': {0: 1, 1: 0}}))

# replace by range, use -inf and inf for extended range
graph.to(MapValues(mapping={'numbers': {'ranges': {'negative': [-inf, 0],
↪'positive': [0, inf]}}}))
```

Parameters

- **mapping** – a dict with entry per column and the associated old/new values map
- **with_original_features** – set to True to keep the original features
- **suffix** – the suffix added to the column name <column>_<suffix> (default is "mapped")
- **kwargs** – optional kwargs (for storey)

class mlrun.feature_store.steps.**OneHotEncoder**(*mapping: Dict[str, List[Union[int, str]]]*, ***kwargs*)
Create new binary fields, one per category (one hot encoded)

example:

```
mapping = {'category': ['food', 'health', 'transportation'],
           'gender': ['male', 'female']}
graph.to(OneHotEncoder(mapping=one_hot_encoder_mapping))
```

Parameters

- **mapping** – a dict of per column categories (to map to binary fields)
- **kwargs** – optional kwargs (for storey)

__init__(*mapping: Dict[str, List[Union[int, str]]]*, ***kwargs*)
Create new binary fields, one per category (one hot encoded)

example:

```
mapping = {'category': ['food', 'health', 'transportation'],
           'gender': ['male', 'female']}
graph.to(OneHotEncoder(mapping=one_hot_encoder_mapping))
```

Parameters

- **mapping** – a dict of per column categories (to map to binary fields)
- **kwargs** – optional kwargs (for storey)

class mlrun.feature_store.steps.**SetEventMetadata**(*id_path: Optional[str] = None*, *key_path: Optional[str] = None*, *time_path: Optional[str] = None*, *random_id: Optional[bool] = None*, ***kwargs*)
Set the event metadata (id, key, timestamp) from the event body

Set the event metadata (id, key, timestamp) from the event body

set the event metadata fields (id, key, and time) from the event body data structure the xx_path attribute defines the key or path to the value in the body dict, "." in the path string indicate the value is in a nested dict e.g. *"x.y"* means *{"x": {"y": value}}*

example:

```
flow = function.set_topology("flow")
# build a graph and use the SetEventMetadata step to extract the id, key and path␣
↪from the event body
# ("myid", "mykey" and "mytime" fields), the metadata will be used for following␣
↪data processing steps
# (e.g. feature store ops, time/key aggregations, write to databases/streams, etc.)
flow.to(SetEventMetadata(id_path="myid", key_path="mykey", time_path="mytime"))
    .to(...)  # additional steps

server = function.to_mock_server()
event = {"myid": "34", "mykey": "123", "mytime": "2022-01-18 15:01"}
resp = server.test(body=event)
```

Parameters

- **id_path** – path to the id value
- **key_path** – path to the key value
- **time_path** – path to the time value (value should be of type str or datetime)
- **random_id** – if True will set the event.id to a random value

__init__(*id_path: Optional[str] = None*, *key_path: Optional[str] = None*, *time_path: Optional[str] = None*, *random_id: Optional[bool] = None*, ***kwargs*)

Set the event metadata (id, key, timestamp) from the event body

set the event metadata fields (id, key, and time) from the event body data structure the xx_path attribute defines the key or path to the value in the body dict, "." in the path string indicate the value is in a nested dict e.g. *"x.y"* means *{"x": {"y": value}}*

example:

```
flow = function.set_topology("flow")
# build a graph and use the SetEventMetadata step to extract the id, key and␣
↪path from the event body
# ("myid", "mykey" and "mytime" fields), the metadata will be used for␣
↪following data processing steps
# (e.g. feature store ops, time/key aggregations, write to databases/streams,␣
↪etc.)
flow.to(SetEventMetadata(id_path="myid", key_path="mykey", time_path="mytime"))
    .to(...)  # additional steps

server = function.to_mock_server()
event = {"myid": "34", "mykey": "123", "mytime": "2022-01-18 15:01"}
resp = server.test(body=event)
```

Parameters

- **id_path** – path to the id value
- **key_path** – path to the key value
- **time_path** – path to the time value (value should be of type str or datetime)
- **random_id** – if True will set the event.id to a random value

17.9 mlrun.model

class mlrun.model.**DataSource**(*name: Optional[str] = None*, *path: Optional[str] = None*, *attributes: Optional[Dict[str, str]] = None*, *key_field: Optional[str] = None*, *time_field: Optional[str] = None*, *schedule: Optional[str] = None*, *start_time: Optional[Union[datetime.datetime, str]] = None*, *end_time: Optional[Union[datetime.datetime, str]] = None*)

Bases: mlrun.model.ModelObj

online or offline data source spec

class mlrun.model.**DataTarget**(*kind: Optional[str] = None*, *name: str = ''*, *path=None*, *online=None*)

Bases: *mlrun.model.DataTargetBase*

data target with extra status information (used in the feature-set/vector status)

class mlrun.model.**DataTargetBase**(*kind: Optional[str] = None*, *name: str = ''*, *path=None*, *attributes: Optional[Dict[str, str]] = None*, *after_step=None*, *partitioned: bool = False*, *key_bucketing_number: Optional[int] = None*, *partition_cols: Optional[List[str]] = None*, *time_partitioning_granularity: Optional[str] = None*, *max_events: Optional[int] = None*, *flush_after_seconds: Optional[int] = None*, *after_state=None*, *storage_options: Optional[Dict[str, str]] = None*)

Bases: mlrun.model.ModelObj

data target spec, specify a destination for the feature set data

classmethod from_dict(*struct=None*, *fields=None*)

create an object from a python dictionary

class mlrun.model.**FeatureSetProducer**(*kind=None*, *name=None*, *uri=None*, *owner=None*, *sources=None*)

Bases: mlrun.model.ModelObj

information about the task/job which produced the feature set data

class mlrun.model.**HyperParamOptions**(*param_file=None*, *strategy=None*, *selector: Optional[mlrun.model.HyperParamStrategies] = None*, *stop_condition=None*, *parallel_runs=None*, *dask_cluster_uri=None*, *max_iterations=None*, *max_errors=None*, *teardown_dask=None*)

Bases: mlrun.model.ModelObj

Hyper Parameter Options

Parameters

- **param_file** (*str*) – hyper params input file path/url, instead of inline
- **strategy** (*str*) – hyper param strategy - grid, list or random
- **selector** (*str*) – selection criteria for best result ([min|max.]<result>), e.g. max.accuracy
- **stop_condition** (*str*) – early stop condition e.g. "accuracy > 0.9"
- **parallel_runs** (*int*) – number of param combinations to run in parallel (over Dask)
- **dask_cluster_uri** (*str*) – db uri for a deployed dask cluster function, e.g. db://myproject/dask
- **max_iterations** (*int*) – max number of runs (in random strategy)
- **max_errors** (*int*) – max number of child runs errors for the overall job to fail
- **teardown_dask** (*bool*) – kill the dask cluster pods after the runs

mlrun.model.**NewTask**(*name=None*, *project=None*, *handler=None*, *params=None*, *hyper_params=None*, *param_file=None*, *selector=None*, *strategy=None*, *inputs=None*, *outputs=None*, *in_path=None*, *out_path=None*, *artifact_path=None*, *secrets=None*, *base=None*)

Creates a new task - see new_task

class mlrun.model.**RunMetadata**(*uid=None*, *name=None*, *project=None*, *labels=None*, *annotations=None*, *iteration=None*)

Bases: mlrun.model.ModelObj

Run metadata

class mlrun.model.**RunObject**(*spec: Optional[mlrun.model.RunSpec] = None*, *metadata: Optional[mlrun.model.RunMetadata] = None*, *status: Optional[mlrun.model.RunStatus] = None*)

Bases: *mlrun.model.RunTemplate*

A run

artifact(*key*) → *mlrun.datastore.base.DataItem*
return artifact DataItem by key

logs(*watch=True*, *db=None*, *offset=0*)
return or watch on the run logs

output(*key*)
return the value of a specific result or artifact by key

property outputs
return a dict of outputs, result values and artifact uris

refresh()
refresh run state from the db

show()
show the current status widget, in jupyter notebook

state()
current run state

property ui_url: str
UI URL (for relevant runtimes)

uid()
run unique id

wait_for_completion(*sleep=3*, *timeout=0*, *raise_on_failure=True*, *show_logs=None*, *logs_interval=None*)
Wait for remote run to complete. Default behavior is to wait until reached terminal state or timeout passed, if timeout is 0 then wait forever It pulls the run status from the db every sleep seconds. If show_logs is not False and logs_interval is not None, it will print the logs when run reached terminal state If show_logs is not False and logs_interval is defined, it will print the logs every logs_interval seconds if show_logs is False it will not print the logs, will still pull the run state until it reaches terminal state

class mlrun.model.**RunSpec**(*parameters=None*, *hyperparams=None*, *param_file=None*, *selector=None*, *handler=None*, *inputs=None*, *outputs=None*, *input_path=None*, *output_path=None*, *function=None*, *secret_sources=None*, *data_stores=None*, *strategy=None*, *verbose=None*, *scrape_metrics=None*, *hyper_param_options=None*, *allow_empty_resources=None*)
Bases: mlrun.model.ModelObj

Run specification

to_dict(*fields=None*, *exclude=None*)
convert the object to a python dictionary

class mlrun.model.**RunStatus**(*state=None*, *error=None*, *host=None*, *commit=None*, *status_text=None*, *results=None*, *artifacts=None*, *start_time=None*, *last_update=None*, *iterations=None*, *ui_url=None*)
Bases: mlrun.model.ModelObj

Run status

class mlrun.model.**RunTemplate**(*spec: Optional[*mlrun.model.RunSpec*] = None*, *metadata: Optional[*mlrun.model.RunMetadata*] = None*)
Bases: mlrun.model.ModelObj

Run template

set_label(*key*, *value*)
set a key/value label for the task

with_hyper_params(*hyperparams*, *selector=None*, *strategy: Optional[mlrun.model.HyperParamStrategies] = None*, ***options*)

set hyper param values and configurations, see parameters in: *HyperParamOptions*

example:

```
grid_params = {"p1": [2,4,1], "p2": [10,20]}
task = mlrun.new_task("grid-search")
task.with_hyper_params(grid_params, selector="max.accuracy")
```

with_input(*key*, *path*)

set task data input, path is an Mlrun global DataItem uri

examples:

```
task.with_input("data", "/file-dir/path/to/file")
task.with_input("data", "s3://<bucket>/path/to/file")
task.with_input("data", "v3io://[<remote-host>]/<data-container>/path/to/file")
```

with_param_file(*param_file*, *selector=None*, *strategy: Optional[mlrun.model.HyperParamStrategies] = None*, ***options*)

set hyper param values (from a file url) and configurations, see parameters in: *HyperParamOptions*

example:

```
grid_params = "s3://<my-bucket>/path/to/params.json"
task = mlrun.new_task("grid-search")
task.with_param_file(grid_params, selector="max.accuracy")
```

with_params(***kwargs*)

set task parameters using key=value, key2=value2, ..

with_secrets(*kind*, *source*)

register a secrets source (file, env or dict)

read secrets from a source provider to be used in workflows, example:

```
task.with_secrets('file', 'file.txt')
task.with_secrets('inline', {'key': 'val'})
task.with_secrets('env', 'ENV1,ENV2')

task.with_secrets('vault', ['secret1', 'secret2'...])

# If using with k8s secrets, the k8s secret is managed by MLRun, through the␣
↪project-secrets
# mechanism. The secrets will be attached to the running pod as environment␣
↪variables.
task.with_secrets('kubernetes', ['secret1', 'secret2'])

# If using an empty secrets list [] then all accessible secrets will be␣
↪available.
task.with_secrets('vault', [])

# To use with Azure key vault, a k8s secret must be created with the following␣
↪keys:
# kubectl -n <namespace> create secret generic azure-key-vault-secret \
```

(continues on next page)

(continued from previous page)

```
#     --from-literal=tenant_id=<service principal tenant ID> \
#     --from-literal=client_id=<service principal client ID> \
#     --from-literal=secret=<service principal secret key>

task.with_secrets('azure_vault', {
    'name': 'my-vault-name',
    'k8s_secret': 'azure-key-vault-secret',
    # An empty secrets list may be passed ('secrets': []) to access all vault␣
↪secrets.
    'secrets': ['secret1', 'secret2'...]
})
```

Parameters

- **kind** – secret type (file, inline, env)
- **source** – secret data or link (see example)

Returns The RunTemplate object

class mlrun.model.**TargetPathObject**(*base_path=None*, *run_id=None*, *is_single_file=False*)

Bases: object

Class configuring the target path This class will take consideration of a few parameters to create the correct end result path:

- **run_id** if run_id is provided target will be considered as run_id mode which require to contain a {run_id} place holder in the path.
- **is_single_file** if true then run_id must be the directory containing the output file or generated before the file name (run_id/output.file).
- **base_path** if contains the place holder for run_id, run_id must not be None. if run_id passed and place holder doesn't exist the place holder will be generated in the correct place.

mlrun.model.**new_task**(*name=None*, *project=None*, *handler=None*, *params=None*, *hyper_params=None*, *param_file=None*, *selector=None*, *hyper_param_options=None*, *inputs=None*, *outputs=None*, *in_path=None*, *out_path=None*, *artifact_path=None*, *secrets=None*, *base=None*) → *mlrun.model.RunTemplate*

Creates a new task

Parameters

- **name** – task name
- **project** – task project
- **handler** – code entry-point/handler name
- **params** – input parameters (dict)
- **hyper_params** – dictionary of hyper parameters and list values, each hyper param holds a list of values, the run will be executed for every parameter combination (GridSearch)
- **param_file** – a csv file with parameter combinations, first row hold the parameter names, following rows hold param values
- **selector** – selection criteria for hyper params e.g. "max.accuracy"
- **hyper_param_options** – hyper parameter options, see: *HyperParamOptions*

- **inputs** – dictionary of input objects + optional paths (if path is omitted the path will be the in_path/key)
- **outputs** – dictionary of input objects + optional paths (if path is omitted the path will be the out_path/key)
- **in_path** – default input path/url (prefix) for inputs
- **out_path** – default output path/url (prefix) for artifacts
- **artifact_path** – default artifact output path
- **secrets** – extra secrets specs, will be injected into the runtime e.g. ['file=<filename>', 'env=ENV_KEY1,ENV_KEY2']
- **base** – task instance to use as a base instead of a fresh new task instance

17.10 mlrun.platforms

mlrun.platforms.**VolumeMount**
alias of mlrun.platforms.iguazio.Mount

mlrun.platforms.**auto_mount**(*pvc_name=''*, *volume_mount_path=''*, *volume_name=None*)
choose the mount based on env variables and params

volume will be selected by the following order: - k8s PVC volume when both pvc_name and volume_mount_path are set - k8s PVC volume when env var is set: MLRUN_PVC_MOUNT=<pvc-name>:<mount-path> - k8s PVC volume if it's configured as the auto mount type - iguazio v3io volume when V3IO_ACCESS_KEY and V3IO_USERNAME env vars are set

mlrun.platforms.**mount_configmap**(*configmap_name*, *mount_path*, *volume_name='configmap'*, *items=None*)
Modifier function to mount kubernetes configmap as files(s)

Parameters

- **configmap_name** – k8s configmap name
- **mount_path** – path to mount inside the container
- **volume_name** – unique volume name
- **items** – If unspecified, each key-value pair in the Data field of the referenced Configmap will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present.

mlrun.platforms.**mount_hostpath**(*host_path*, *mount_path*, *volume_name='hostpath'*)
Modifier function to mount kubernetes configmap as files(s)

Parameters

- **host_path** – host path
- **mount_path** – path to mount inside the container
- **volume_name** – unique volume name

mlrun.platforms.**mount_pvc**(*pvc_name=None*, *volume_name='pipeline'*, *volume_mount_path='/mnt/pipeline'*)
Modifier function to apply to a Container Op to simplify volume, volume mount addition and enable better reuse of volumes, volume claims across container ops.

Usage:

```
train = train_op(...)
train.apply(mount_pvc('claim-name', 'pipeline', '/mnt/pipeline'))
```

mlrun.platforms.**mount_s3**(*secret_name=None*, *aws_access_key=''*, *aws_secret_key=''*, *endpoint_url=None*, *prefix=''*, *aws_region=None*, *non_anonymous=False*)

Modifier function to add s3 env vars or secrets to container

Warning: Using this function to configure AWS credentials will expose these credentials in the pod spec of the runtime created. It is recommended to use the *secret_name* parameter, or set the credentials as project-secrets and avoid using this function.

Parameters

- **secret_name** – kubernetes secret name (storing the access/secret keys)
- **aws_access_key** – AWS_ACCESS_KEY_ID value. If this parameter is not specified and AWS_ACCESS_KEY_ID env. variable is defined, the value will be taken from the env. variable
- **aws_secret_key** – AWS_SECRET_ACCESS_KEY value. If this parameter is not specified and AWS_SECRET_ACCESS_KEY env. variable is defined, the value will be taken from the env. variable
- **endpoint_url** – s3 endpoint address (for non AWS s3)
- **prefix** – string prefix to add before the env var name (for working with multiple s3 data stores)
- **aws_region** – amazon region
- **non_anonymous** – force the S3 API to use non-anonymous connection, even if no credentials are provided (for authenticating externally, such as through IAM instance-roles)

mlrun.platforms.**mount_secret**(*secret_name*, *mount_path*, *volume_name='secret'*, *items=None*)

Modifier function to mount kubernetes secret as files(s)

Parameters

- **secret_name** – k8s secret name
- **mount_path** – path to mount inside the container
- **volume_name** – unique volume name
- **items** – If unspecified, each key-value pair in the Data field of the referenced Secret will be projected into the volume as a file whose name is the key and content is the value. If specified, the listed keys will be projected into the specified paths, and unlisted keys will not be present.

mlrun.platforms.**mount_v3io**(*name='v3io'*, *remote=''*, *mount_path=''*, *access_key=''*, *user=''*, *secret=None*, *volume_mounts=None*)

Modifier function to apply to a Container Op to volume mount a v3io path

Parameters

- **name** – the volume name
- **remote** – the v3io path to use for the volume. ~/ prefix will be replaced with /users/<username>/
- **mount_path** – the volume mount path (deprecated, exists for backwards compatibility, prefer to use mounts instead)

- **access_key** – the access key used to auth against v3io. if not given V3IO_ACCESS_KEY env var will be used
- **user** – the username used to auth against v3io. if not given V3IO_USERNAME env var will be used
- **secret** – k8s secret name which would be used to get the username and access key to auth against v3io.
- **volume_mounts** – list of VolumeMount. empty volume mounts & remote will default to mount /v3io & /User.

mlrun.platforms.**mount_v3io_extended**(*name='v3io'*, *remote=''*, *mounts=None*, *access_key=''*, *user=''*, *secret=None*)

Modifier function to apply to a Container Op to volume mount a v3io path

Parameters

- **name** – the volume name
- **remote** – the v3io path to use for the volume. ~/ prefix will be replaced with /users/<username>/
- **mounts** – list of mount & volume sub paths (type Mount). empty mounts & remote mount /v3io & /User
- **access_key** – the access key used to auth against v3io. if not given V3IO_ACCESS_KEY env var will be used
- **user** – the username used to auth against v3io. if not given V3IO_USERNAME env var will be used
- **secret** – k8s secret name which would be used to get the username and access key to auth against v3io.

mlrun.platforms.**mount_v3io_legacy**(*name='v3io'*, *remote='~/'*, *mount_path='/User'*, *access_key=''*, *user=''*, *secret=None*)

Modifier function to apply to a Container Op to volume mount a v3io path :param name: the volume name :param remote: the v3io path to use for the volume. ~/ prefix will be replaced with /users/<username>/ :param mount_path: the volume mount path :param access_key: the access key used to auth against v3io. if not given V3IO_ACCESS_KEY env var will be used :param user: the username used to auth against v3io. if not given V3IO_USERNAME env var will be used :param secret: k8s secret name which would be used to get the username and access key to auth against v3io.

mlrun.platforms.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*)

Pretty-print a Python object to a stream [default is sys.stdout].

mlrun.platforms.**set_env_variables**(*env_vars_dict: Optional[Dict[str, str]] = None*, ***kwargs*)

Modifier function to apply a set of environment variables to a runtime. Variables may be passed as either a dictionary of name-value pairs, or as arguments to the function. See *KubeResource.apply* for more information on modifiers.

Usage:

```
function.apply(set_env_variables({"ENV1": "value1", "ENV2": "value2"}))
or
function.apply(set_env_variables(ENV1=value1, ENV2=value2))
```

Parameters

- **env_vars_dict** – dictionary of env. variables

- **kwargs** – environment variables passed as args

mlrun.platforms.**sleep**(*seconds*)
Delay execution for a given number of seconds. The argument may be a floating point number for subsecond precision.

mlrun.platforms.**v3io_cred**(*api=''*, *user=''*, *access_key=''*)
Modifier function to copy local v3io env vars to container

Usage:

```
train = train_op(...)
train.apply(use_v3io_cred())
```

mlrun.platforms.**watch_stream**(*url*, *shard_ids: Optional[list] = None*, *seek_to: Optional[str] = None*, *interval=None*, *is_json=False*, ***kwargs*)
watch on a v3io stream and print data every interval

example:

```
watch_stream('v3io:///users/admin/mystream')
```

Parameters

- **url** – stream url
- **shard_ids** – range or list of shard IDs
- **seek_to** – where to start/seek ('EARLIEST', 'LATEST', 'TIME', 'SEQUENCE')
- **interval** – watch interval time in seconds, 0 to run once and return
- **is_json** – indicate the payload is json (will be deserialized)

17.11 mlrun.projects

class mlrun.projects.**MlrunProject**(*name=None*, *description=None*, *params=None*, *functions=None*, *workflows=None*, *artifacts=None*, *artifact_path=None*, *conda=None*, *metadata=None*, *spec=None*, *default_requirements: Optional[Union[str, List[str]]] = None*)
Bases: mlrun.model.ModelObj

property artifact_path: str
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

property artifacts: list
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

build_function(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *with_mlrun: Optional[bool] = None*, *skip_deployed: bool = False*, *image=None*, *base_image=None*, *commands: Optional[list] = None*, *secret_name=''*, *requirements: Optional[Union[str, List[str]]] = None*, *mlrun_version_specifier=None*, *builder_env: Optional[dict] = None*, *overwrite_build_params: bool = False*) → Union[mlrun.projects.operations.BuildStatus, kfp.dsl._container_op.ContainerOp]
deploy ML function, build container with its dependencies

Parameters

- **function** – name of the function (in the project) or function object
- **with_mlrun** – add the current mlrun package to the container build
- **skip_deployed** – skip the build if we already have an image for the function
- **image** – target image name/path
- **base_image** – base image name/path (commands and source code will be added to it)
- **commands** – list of docker build (RUN) commands e.g. ['pip install pandas']
- **secret_name** – k8s secret for accessing the docker registry
- **requirements** – list of python packages or pip requirements file path, defaults to None
- **mlrun_version_specifier** – which mlrun package version to include (if not current)
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}, does not work yet in KFP
- **overwrite_build_params** – overwrite the function build parameters with the provided ones, or attempt to add to existing parameters

clear_context()
delete all files and clear the context dir

property context: str
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

create_remote(*url*, *name='origin'*, *branch=None*)
create remote for the project git

Parameters

- **url** – remote git url
- **name** – name for the remote (default is 'origin')
- **branch** – Git branch to use as source

create_vault_secrets(*secrets*)

deploy_function(*function: Union[str,* mlrun.runtimes.base.BaseRuntime*]*, *dashboard: str = ''*, *models: Optional[list] = None*, *env: Optional[dict] = None*, *tag: Optional[str] = None*, *verbose: Optional[bool] = None*, *builder_env: Optional[dict] = None*, *mock: Optional[bool] = None*) → Union[mlrun.projects.operations.DeployStatus, kfp.dsl._container_op.ContainerOp]
deploy real-time (nuclio based) functions

Parameters

- **function** – name of the function (in the project) or function object
- **dashboard** – url of the remote Nuclio dashboard (when not local)
- **models** – list of model items
- **env** – dict of extra environment variables
- **tag** – extra version tag

- **verbose** – add verbose prints/logs
- **builder_env** – env vars dict for source archive config/credentials e.g. *builder_env={"GIT_TOKEN": token}*
- **mock** – deploy mock server vs a real Nuclio function (for local simulations)

property description: str
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

export(*filepath=None*, *include_files: Optional[str] = None*)
save the project object into a yaml file or zip archive (default to project.yaml)

By default the project object is exported to a yaml file, when the filepath suffix is '.zip' the project context dir (code files) are also copied into the zip, the archive path can include DataItem urls (for remote object storage, e.g. s3://<bucket>/<path>).

Parameters

- **filepath** – path to store project .yaml or .zip (with the project dir content)
- **include_files** – glob filter string for selecting files to include in the zip archive

func(*key*, *sync=False*) → *mlrun.runtimes.base.BaseRuntime*
get function object by name

Parameters **sync** – will reload/reinit the function

Returns function object

property functions: list
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

get_artifact(*key*, *tag=None*, *iter=None*)
Return an artifact object

Parameters

- **key** – artifact key
- **tag** – version tag
- **iter** – iteration number (for hyper-param tasks)

Returns Artifact object

get_artifact_uri(*key: str*, *category: str = 'artifact'*, *tag: Optional[str] = None*) → str
return the project artifact uri (store://..) from the artifact key

example:

```
uri = project.get_artifact_uri("my_model", category="model", tag="prod")
```

Parameters

- **key** – artifact key/name
- **category** – artifact category (artifact, model, feature-vector, ..)
- **tag** – artifact version tag, default to latest version

get_function(*key*, *sync=False*, *enrich=False*, *ignore_cache=False*, *copy_function=True*) → *mlrun.runtimes.base.BaseRuntime*

get function object by name

Parameters

- **key** – name of key for search
- **sync** – will reload/reinit the function from the project spec
- **enrich** – add project info/config/source info to the function object
- **ignore_cache** – read the function object from the DB (ignore the local cache)
- **copy_function** – return a copy of the function object

Returns function object

get_function_objects() → Dict[str, *mlrun.runtimes.base.BaseRuntime*]

"get a virtual dict with all the project functions ready for use in a pipeline

get_param(*key: str*, *default=None*)

get project param by key

get_run_status(*run*, *timeout=None*, *expected_statuses=None*, *notifiers: Optional[mlrun.utils.notifications.notification_pusher.CustomNotificationPusher] = None*)

get_secret(*key: str*)

get a key based secret e.g. DB password from the context secrets can be specified when invoking a run through files, env, ..

get_store_resource(*uri*)

get store resource object by uri

get_vault_secrets(*secrets=None*, *local=False*)

import_artifact(*item_path: str*, *new_key=None*, *artifact_path=None*, *tag=None*)

Import an artifact object/package from .yaml, .json, or .zip file

Parameters

- **item_path** – dataitem url or file path to the file/package
- **new_key** – overwrite the artifact key/name
- **artifact_path** – target artifact path (when not using the default)
- **tag** – artifact tag to set

Returns artifact object

kind = 'project'

list_artifacts(*name=None*, *tag=None*, *labels=None*, *since=None*, *until=None*, *iter: Optional[int] = None*, *best_iteration: bool = False*, *kind: Optional[str] = None*, *category: Optional[Union[str, mlrun.api.schemas.artifact.ArtifactCategories]] = None*) → mlrun.lists.ArtifactList

List artifacts filtered by various parameters.

The returned result is an *ArtifactList* (list of dict), use *.to_objects()* to convert it to a list of RunObjects, *.show()* to view graphically in Jupyter, and *.to_df()* to convert to a DataFrame.

Examples:

```
# Get latest version of all artifacts in project
latest_artifacts = project.list_artifacts('', tag='latest')
# check different artifact versions for a specific artifact, return as objects␣
↪list
result_versions = project.list_artifacts('results', tag='*').to_objects()
```

Parameters

- **name** – Name of artifacts to retrieve. Name is used as a like query, and is not case-sensitive. This means that querying for `name` may return artifacts named `my_Name_1` or `surname`.
- **tag** – Return artifacts assigned this tag.
- **labels** – Return artifacts that have these labels.
- **since** – Not in use in `HTTPRunDB`.
- **until** – Not in use in `HTTPRunDB`.
- **iter** – Return artifacts from a specific iteration (where `iter=0` means the root iteration). If `None` (default) return artifacts from all iterations.
- **best_iteration** – Returns the artifact which belongs to the best iteration of a given run, in the case of artifacts generated from a hyper-param run. If only a single iteration exists, will return the artifact from that iteration. If using `best_iter`, the `iter` parameter must not be used.
- **kind** – Return artifacts of the requested kind.
- **category** – Return artifacts of the requested category.

list_functions(*name=None*, *tag=None*, *labels=None*)

Retrieve a list of functions, filtered by specific criteria.

example:

```
functions = project.list_functions(tag="latest")
```

Parameters

- **name** – Return only functions with a specific name.
- **tag** – Return function versions with specific tags.
- **labels** – Return functions that have specific labels assigned to them.

Returns List of function objects.

list_models(*name=None*, *tag=None*, *labels=None*, *since=None*, *until=None*, *iter: Optional[int] = None*, *best_iteration: bool = False*)

List models in project, filtered by various parameters.

Examples:

```
# Get latest version of all models in project
latest_models = project.list_models('', tag='latest')
```

Parameters

- **name** – Name of artifacts to retrieve. Name is used as a like query, and is not case-sensitive. This means that querying for `name` may return artifacts named `my_Name_1` or `surname`.
- **tag** – Return artifacts assigned this tag.
- **labels** – Return artifacts that have these labels.
- **since** – Not in use in `HTTPRunDB`.
- **until** – Not in use in `HTTPRunDB`.
- **iter** – Return artifacts from a specific iteration (where `iter=0` means the root iteration). If `None` (default) return artifacts from all iterations.
- **best_iteration** – Returns the artifact which belongs to the best iteration of a given run, in the case of artifacts generated from a hyper-param run. If only a single iteration exists, will return the artifact from that iteration. If using `best_iter`, the `iter` parameter must not be used.

list_runs(*name=None*, *uid=None*, *labels=None*, *state=None*, *sort=True*, *last=0*, *iter=False*, *start_time_from: Optional[datetime.datetime] = None*, *start_time_to: Optional[datetime.datetime] = None*, *last_update_time_from: Optional[datetime.datetime] = None*, *last_update_time_to: Optional[datetime.datetime] = None*, ***kwargs*) → mlrun.lists.RunList

Retrieve a list of runs, filtered by various options.

The returned result is a `` (list of dict), use *.to_objects()* to convert it to a list of RunObjects, *.show()* to view graphically in Jupyter, *.to_df()* to convert to a DataFrame, and *compare()* to generate comparison table and PCP plot.

Example:

```
# return a list of runs matching the name and label and compare
runs = project.list_runs(name='download', labels='owner=admin')
runs.compare()
# If running in Jupyter, can use the .show() function to display the results
project.list_runs(name='').show()
```

Parameters

- **name** – Name of the run to retrieve.
- **uid** – Unique ID of the run.
- **project** – Project that the runs belongs to.
- **labels** – List runs that have a specific label assigned. Currently only a single label filter can be applied, otherwise result will be empty.
- **state** – List only runs whose state is specified.
- **sort** – Whether to sort the result according to their start time. Otherwise, results will be returned by their internal order in the DB (order will not be guaranteed).
- **last** – Deprecated - currently not used.
- **iter** – If `True` return runs from all iterations. Otherwise, return only runs whose `iter` is 0.
- **start_time_from** – Filter by run start time in `[start_time_from, start_time_to]`.

- **start_time_to** – Filter by run start time in [start_time_from, start_time_to].
- **last_update_time_from** – Filter by run last update time in (last_update_time_from, last_update_time_to).
- **last_update_time_to** – Filter by run last update time in (last_update_time_from, last_update_time_to).

log_artifact(*item*, *body=None*, *tag=''*, *local_path=''*, *artifact_path=None*, *format=None*, *upload=None*, *labels=None*, *target_path=None*, ***kwargs*)

log an output artifact and optionally upload it to datastore

example:

```
project.log_artifact(
    "some-data",
    body=b"abc is 123",
    local_path="model.txt",
    labels={"framework": "xgboost"},
)
```

Parameters

- **item** – artifact key or artifact class ()
- **body** – will use the body as the artifact content
- **local_path** – path to the local file we upload, will also be use as the destination subpath (under "artifact_path")
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **format** – artifact file format: csv, png, ..
- **tag** – version tag
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with

Returns artifact object

log_dataset(*key*, *df*, *tag=''*, *local_path=None*, *artifact_path=None*, *upload=None*, *labels=None*, *format=''*, *preview=None*, *stats=None*, *target_path=''*, *extra_data=None*, *label_column: Optional[str] = None*, ***kwargs*) → mlrun.artifacts.dataset.DatasetArtifact

log a dataset artifact and optionally upload it to datastore

example:

```
raw_data = {
    "first_name": ["Jason", "Molly", "Tina", "Jake", "Amy"],
    "last_name": ["Miller", "Jacobson", "Ali", "Milner", "Cooze"],
    "age": [42, 52, 36, 24, 73],
    "testScore": [25, 94, 57, 62, 70],
}
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame(raw_data, columns=["first_name", "last_name", "age",
↪"testScore"])
project.log_dataset("mydf", df=df, stats=True)
```

Parameters

- **key** – artifact key
- **df** – dataframe object
- **label_column** – name of the label column (the one holding the target (y) values)
- **local_path** – path to the local dataframe file that exists locally. The given file extension will be used to save the dataframe to a file If the file exists, it will be uploaded to the datastore instead of the given df.
- **artifact_path** – target artifact path (when not using the default). to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **tag** – version tag
- **format** – optional, format to use (e.g. csv, parquet, ..)
- **target_path** – absolute target path (instead of using artifact_path + local_path)
- **preview** – number of lines to store as preview in the artifact metadata
- **stats** – calculate and store dataset stats in the artifact metadata
- **extra_data** – key/value list of extra files/charts to link with this dataset
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with

Returns artifact object

log_model(*key*, *body=None*, *framework=''*, *tag=''*, *model_dir=None*, *model_file=None*, *algorithm=None*, *metrics=None*, *parameters=None*, *artifact_path=None*, *upload=None*, *labels=None*, *inputs: Optional[List[*mlrun.features.Feature*]] = None*, *outputs: Optional[List[*mlrun.features.Feature*]] = None*, *feature_vector: Optional[str] = None*, *feature_weights: Optional[list] = None*, *training_set=None*, *label_column=None*, *extra_data=None*, ***kwargs*)

log a model artifact and optionally upload it to datastore

example:

```
project.log_model("model", body=dumps(model),
                  model_file="model.pkl",
                  metrics=context.results,
                  training_set=training_df,
                  label_column='label',
                  feature_vector=feature_vector_uri,
                  labels={"app": "fraud"})
```

Parameters

- **key** – artifact key or artifact class ()
- **body** – will use the body as the artifact content

- **model_file** – path to the local model file we upload (see also model_dir) or to a model file data url (e.g. http://host/path/model.pkl)
- **model_dir** – path to the local dir holding the model file and extra files
- **artifact_path** – target artifact path (when not using the default) to define a subpath under the default location use: *artifact_path=context.artifact_subpath('data')*
- **framework** – name of the ML framework
- **algorithm** – training algorithm name
- **tag** – version tag
- **metrics** – key/value dict of model metrics
- **parameters** – key/value dict of model parameters
- **inputs** – ordered list of model input features (name, type, ..)
- **outputs** – ordered list of model output/result elements (name, type, ..)
- **upload** – upload to datastore (default is True)
- **labels** – a set of key/value labels to tag the artifact with
- **feature_vector** – feature store feature vector uri (store://feature-vectors/<project>/<name>[:tag])
- **feature_weights** – list of feature weights, one per input column
- **training_set** – training set dataframe, used to infer inputs & outputs
- **label_column** – which columns in the training set are the label (target) columns
- **extra_data** – key/value list of extra files/charts to link with this dataset value can be absolute path | relative path (to model dir) | bytes | artifact object

Returns artifact object

property metadata: *mlrun.projects.project.ProjectMetadata*

property mountdir: str
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

property name: str
Project name, this is a property of the project metadata

property notifiers

property params: str
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

pull(*branch=None*, *remote=None*)
pull/update sources from git or tar into the context dir

Parameters

- **branch** – git branch, if not the current one
- **remote** – git remote, if other than origin

push(*branch*, *message=None*, *update=True*, *remote=None*, *add: Optional[list] = None*)
update spec and push updates to remote git repo

Parameters

- **branch** – target git branch
- **message** – git commit message
- **update** – update files (git add update=True)
- **remote** – git remote, default to origin
- **add** – list of files to add

register_artifacts()
register the artifacts in the MLRun DB (under this project)

reload(*sync=False*, *context=None*) → *mlrun.projects.project.MlrunProject*
reload the project and function objects from the project yaml/specs

Parameters

- **sync** – set to True to load functions objects
- **context** – context directory (where the yaml and code exist)

Returns project object

remove_function(*name*)
remove a function from a project

Parameters **name** – name of the function (under the project)

run(*name: Optional[str] = None*, *workflow_path: Optional[str] = None*, *arguments: Optional[Dict[str, Any]] = None*, *artifact_path: Optional[str] = None*, *workflow_handler: Optional[Union[str, Callable]] = None*, *namespace: Optional[str] = None*, *sync: bool = False*, *watch: bool = False*, *dirty: bool = False*, *ttl: Optional[int] = None*, *engine: Optional[str] = None*, *local: Optional[bool] = None*, *schedule: Optional[Union[str, mlrun.api.schemas.schedule.ScheduleCronTrigger, bool]] = None*, *timeout: Optional[int] = None*, *overwrite: bool = False*) → mlrun.projects.pipelines._PipelineRunStatus
run a workflow using kubeflow pipelines

Parameters

- **name** – name of the workflow
- **workflow_path** – url to a workflow file, if not a project workflow
- **arguments** – kubeflow pipelines arguments (parameters)
- **artifact_path** – target path/url for workflow artifacts, the string '{{workflow.uid}}' will be replaced by workflow id
- **workflow_handler** – workflow function handler (for running workflow function directly)
- **namespace** – kubernetes namespace if other than default
- **sync** – force functions sync before run
- **watch** – wait for pipeline completion
- **dirty** – allow running the workflow when the git repo is dirty
- **ttl** – pipeline ttl in secs (after that the pods will be removed)
- **engine** – workflow engine running the workflow. supported values are 'kfp' (default), 'local' or 'remote'. for setting engine for remote running use 'remote:local' or 'remote:kfp'.

- **local** – run local pipeline with local functions (set local=True in function.run())
- **schedule** – ScheduleCronTrigger class instance or a standard crontab expression string (which will be converted to the class using its *from_crontab* constructor), see this link for help: https://apscheduler.readthedocs.io/en/3.x/modules/triggers/cron.html#module-apscheduler.triggers.cron for using the pre-defined workflow's schedule, set *schedule=True*
- **timeout** – timeout in seconds to wait for pipeline completion (used when watch=True)
- **overwrite** – replacing the schedule of the same workflow (under the same name) if exists with the new one.

Returns run id

run_function(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *handler: Optional[str] = None*, *name: str = ''*, *params: Optional[dict] = None*, *hyperparams: Optional[dict] = None*, *hyper_param_options: Optional[mlrun.model.HyperParamOptions] = None*, *inputs: Optional[dict] = None*, *outputs: Optional[List[str]] = None*, *workdir: str = ''*, *labels: Optional[dict] = None*, *base_task: Optional[mlrun.model.RunTemplate] = None*, *watch: bool = True*, *local: Optional[bool] = None*, *verbose: Optional[bool] = None*, *selector: Optional[str] = None*, *auto_build: Optional[bool] = None*, *schedule: Optional[Union[str, mlrun.api.schemas.schedule.ScheduleCronTrigger]] = None*, *artifact_path: Optional[str] = None*) → Union[*mlrun.model.RunObject*, kfp.dsl._container_op.ContainerOp]

Run a local or remote task as part of a local/kubeflow pipeline

example (use with project):

```
# create a project with two functions (local and from marketplace)
project = mlrun.new_project(project_name, "./proj")
project.set_function("mycode.py", "myfunc", image="mlrun/mlrun")
project.set_function("hub://sklearn_classifier", "train")

# run functions (refer to them by name)
run1 = project.run_function("myfunc", params={"x": 7})
run2 = project.run_function("train", params={"data": run1.outputs["data"]})
```

Parameters

- **function** – name of the function (in the project) or function object
- **handler** – name of the function handler
- **name** – execution name
- **params** – input parameters (dict)
- **hyperparams** – hyper parameters
- **selector** – selection criteria for hyper params e.g. "max.accuracy"
- **hyper_param_options** – hyper param options (selector, early stop, strategy, ..) see: *HyperParamOptions*
- **inputs** – input objects (dict of key: path)
- **outputs** – list of outputs which can pass in the workflow
- **workdir** – default input artifacts path
- **labels** – labels to tag the job/run with ({key:val, ..})
- **base_task** – task object to use as base

- **watch** – watch/follow run log, True by default
- **local** – run the function locally vs on the runtime/cluster
- **verbose** – add verbose prints/logs
- **auto_build** – when set to True and the function require build it will be built on the first function run, use only if you dont plan on changing the build config between runs
- **schedule** – ScheduleCronTrigger class instance or a standard crontab expression string (which will be converted to the class using its *from_crontab* constructor), see this link for help: https://apscheduler.readthedocs.io/en/v3.6.3/modules/triggers/cron.html#module-apscheduler.triggers.cron
- **artifact_path** – path to store artifacts, when running in a workflow this will be set automatically

Returns MLRun RunObject or KubeFlow containerOp

save(*filepath=None*, *store=True*)
export project to yaml file and save project in database

Store if True, allow updating in case project already exists

save_to_db(*store=True*)
save project to database

Store if True, allow updating in case project already exists

save_workflow(*name*, *target*, *artifact_path=None*, *ttl=None*)
create and save a workflow as a yaml or archive file

Parameters

- **name** – workflow name
- **target** – target file path (can end with .yaml or .zip)
- **artifact_path** – target path/url for workflow artifacts, the string '{{workflow.uid}}' will be replaced by workflow id
- **ttl** – pipeline ttl (time to live) in secs (after that the pods will be removed)

set_artifact(*key*, *artifact: Optional[Union[str, dict, mlrun.artifacts.base.Artifact]] = None*, *target_path: Optional[str] = None*, *tag: Optional[str] = None*)
add/set an artifact in the project spec (will be registered on load)

example:

```
# register a simple file artifact
project.set_artifact('data', target_path=data_url)
# register a model artifact
project.set_artifact('model', ModelArtifact(model_file="model.pkl"), target_
↪path=model_dir_url)

# register a path to artifact package (will be imported on project load)
# to generate such package use `artifact.export(target_path)`
project.set_artifact('model', 'https://mystuff.com/models/mymodel.zip')
```

Parameters

- **key** – artifact key/name

- **artifact** – mlrun Artifact object/dict (or its subclasses) or path to artifact file to import (yaml/json/zip), relative paths are relative to the context path
- **target_path** – absolute target path url (point to the artifact content location)
- **tag** – artifact tag

set_function(*func: Optional[Union[str, mlrun.runtimes.base.BaseRuntime]] = None*, *name: str = ''*, *kind: str = ''*, *image: Optional[str] = None*, *handler=None*, *with_repo: Optional[bool] = None*, *tag: Optional[str] = None*, *requirements: Optional[Union[str, List[str]]] = None*) → *mlrun.runtimes.base.BaseRuntime*

update or add a function object to the project

function can be provided as an object (func) or a .py/.ipynb/.yaml url support url prefixes:

```
object (s3://, v3io://, ..)
MLRun DB e.g. db://project/func:ver
functions hub/market: e.g. hub://sklearn_classifier:master
```

examples:

```
proj.set_function(func_object)
proj.set_function('./src/mycode.py', 'ingest',
                  image='myrepo/ing:latest', with_repo=True)
proj.set_function('http://.../mynb.ipynb', 'train')
proj.set_function('./func.yaml')
proj.set_function('hub://get_toy_data', 'getdata')
```

Parameters

- **func** – function object or spec/code url, None refers to current Notebook
- **name** – name of the function (under the project)
- **kind** – runtime kind e.g. job, nuclio, spark, dask, mpijob default: job
- **image** – docker image to be used, can also be specified in the function object/yaml
- **handler** – default function handler to invoke (can only be set with .py/.ipynb files)
- **with_repo** – add (clone) the current repo to the build source
- **requirements** – list of python packages or pip requirements file path

Tag function version tag (none for 'latest', can only be set with .py/.ipynb files)

Returns project object

set_model_monitoring_credentials(*access_key: str*)

Set the credentials that will be used by the project's model monitoring infrastructure functions. The supplied credentials must have data access

Parameters **access_key** – Model Monitoring access key for managing user permissions.

set_secrets(*secrets: Optional[dict] = None*, *file_path: Optional[str] = None*, *provider: Optional[Union[str, mlrun.api.schemas.secret.SecretProviderName]] = None*)

set project secrets from dict or secrets env file when using a secrets file it should have lines in the form KEY=VALUE, comment line start with "#" V3IO paths/credentials and MLrun service API address are dropped from the secrets

example secrets file:

```
# this is an env file
AWS_ACCESS_KEY_ID-XXXX
AWS_SECRET_ACCESS_KEY=YYYY
```

usage:

```
# read env vars from dict or file and set as project secrets
project.set_secrets({"SECRET1": "value"})
project.set_secrets(file_path="secrets.env")
```

Parameters

- **secrets** – dict with secrets key/value
- **file_path** – path to secrets file
- **provider** – MLRun secrets provider

set_source(*source*, *pull_at_runtime=False*, *workdir=None*)

set the project source code path(can be git/tar/zip archive)

Parameters

- **source** – valid path to git, zip, or tar file, (or None for current) e.g. git://github.com/mlrun/something.git http://some/url/file.zip
- **pull_at_runtime** – load the archive into the container at job runtime vs on build/deploy
- **workdir** – the relative workdir path (under the context dir)

set_workflow(*name*, *workflow_path: str*, *embed=False*, *engine=None*, *args_schema: Optional[List[mlrun.model.EntrypointParam]] = None*, *handler=None*, *schedule: Optional[Union[str, mlrun.api.schemas.schedule.ScheduleCronTrigger]] = None*, *ttl=None*, ***args*)

add or update a workflow, specify a name and the code path

Parameters

- **name** – name of the workflow
- **workflow_path** – url/path for the workflow file
- **embed** – add the workflow code into the project.yaml
- **engine** – workflow processing engine ("kfp" or "local")
- **args_schema** – list of arg schema definitions (:py:class`~mlrun.model.EntrypointParam`)
- **handler** – workflow function handler
- **schedule** – ScheduleCronTrigger class instance or a standard crontab expression string (which will be converted to the class using its *from_crontab* constructor), see this link for help: https://apscheduler.readthedocs.io/en/3.x/modules/triggers/cron.html#module-apscheduler.triggers.cron
- **ttl** – pipeline ttl in secs (after that the pods will be removed)
- **args** – argument values (key=value, ..)

property source: str
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

property spec: *mlrun.projects.project.ProjectSpec*

property status: *mlrun.projects.project.ProjectStatus*

sync_functions(*names: Optional[list] = None*, *always=True*, *save=False*)
reload function objects from specs and files

with_secrets(*kind*, *source*, *prefix=''*)
register a secrets source (file, env or dict)

read secrets from a source provider to be used in workflows, example:

```
proj.with_secrets('file', 'file.txt')
proj.with_secrets('inline', {'key': 'val'})
proj.with_secrets('env', 'ENV1,ENV2', prefix='PFX_')
```

Vault secret source has several options:

```
proj.with_secrets('vault', {'user': <user name>, 'secrets': ['secret1',
↪'secret2' ...]})
proj.with_secrets('vault', {'project': <proj.name>, 'secrets': ['secret1',
↪'secret2' ...]})
proj.with_secrets('vault', ['secret1', 'secret2' ...])
```

The 2nd option uses the current project name as context. Can also use empty secret list:

```
proj.with_secrets('vault', [])
```

This will enable access to all secrets in vault registered to the current project.

Parameters

- **kind** – secret type (file, inline, env, vault)
- **source** – secret data or link (see example)
- **prefix** – add a prefix to the keys in this source

Returns project object

property workflows: list
This is a property of the spec, look there for documentation leaving here for backwards compatibility with users code that used MlrunProjectLegacy

class mlrun.projects.**ProjectMetadata**(*name=None*, *created=None*, *labels=None*, *annotations=None*)
Bases: mlrun.model.ModelObj

property name: str
Project name

static validate_project_name(*name: str*, *raise_on_failure: bool = True*) → bool

class mlrun.projects.**ProjectSpec**(*description=None*, *params=None*, *functions=None*, *workflows=None*, *artifacts=None*, *artifact_path=None*, *conda=None*, *source=None*, *subpath=None*, *origin_url=None*, *goals=None*, *load_source_on_run=None*, *default_requirements: Optional[Union[str, List[str]]] = None*, *desired_state='online'*, *owner=None*, *disable_auto_mount=None*, *workdir=None*)

Bases: mlrun.model.ModelObj

property artifacts: list
list of artifacts used in this project

property functions: list
list of function object/specs used in this project

get_code_path()
Get the path to the code root/workdir

property mountdir: str
specify to mount the context dir inside the function container use '.' to use the same path as in the client e.g. Jupyter

remove_artifact(*key*)

remove_function(*name*)

remove_workflow(*name*)

set_artifact(*key*, *artifact*)

set_function(*name*, *function_object*, *function_dict*)

set_workflow(*name*, *workflow*)

property source: str
source url or git repo

property workflows: List[dict]
list of workflows specs dicts used in this project

Type returns

class mlrun.projects.**ProjectStatus**(*state=None*)

Bases: mlrun.model.ModelObj

mlrun.projects.**build_function**(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *with_mlrun: Optional[bool] = None*, *skip_deployed: bool = False*, *image=None*, *base_image=None*, *commands: Optional[list] = None*, *secret_name=''*, *requirements: Optional[Union[str, List[str]]] = None*, *mlrun_version_specifier=None*, *builder_env: Optional[dict] = None*, *project_object=None*, *overwrite_build_params: bool = False*) → Union[mlrun.projects.operations.BuildStatus, kfp.dsl._container_op.ContainerOp]

deploy ML function, build container with its dependencies

Parameters

- **function** – name of the function (in the project) or function object
- **with_mlrun** – add the current mlrun package to the container build
- **skip_deployed** – skip the build if we already have an image for the function
- **image** – target image name/path

- **base_image** – base image name/path (commands and source code will be added to it)
- **commands** – list of docker build (RUN) commands e.g. ['pip install pandas']
- **secret_name** – k8s secret for accessing the docker registry
- **requirements** – list of python packages or pip requirements file path, defaults to None
- **mlrun_version_specifier** – which mlrun package version to include (if not current)
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}, does not work yet in KFP
- **project_object** – override the project object to use, will default to the project set in the runtime context.
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}, does not work yet in KFP
- **overwrite_build_params** – overwrite the function build parameters with the provided ones, or attempt to add to existing parameters

mlrun.projects.**deploy_function**(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *dashboard: str = ''*, *models: Optional[list] = None*, *env: Optional[dict] = None*, *tag: Optional[str] = None*, *verbose: Optional[bool] = None*, *builder_env: Optional[dict] = None*, *project_object=None*, *mock: Optional[bool] = None*) → Union[mlrun.projects.operations.DeployStatus, kfp.dsl._container_op.ContainerOp]

deploy real-time (nuclio based) functions

Parameters

- **function** – name of the function (in the project) or function object
- **dashboard** – url of the remote Nuclio dashboard (when not local)
- **models** – list of model items
- **env** – dict of extra environment variables
- **tag** – extra version tag
- **verbose** – add verbose prints/logs
- **builder_env** – env vars dict for source archive config/credentials e.g. builder_env={"GIT_TOKEN": token}
- **mock** – deploy mock server vs a real Nuclio function (for local simulations)
- **project_object** – override the project object to use, will default to the project set in the runtime context.

mlrun.projects.**get_or_create_project**(*name: str*, *context: str = './'*, *url: Optional[str] = None*, *secrets: Optional[dict] = None*, *init_git=False*, *subpath: Optional[str] = None*, *clone: bool = False*, *user_project: bool = False*, *from_template: Optional[str] = None*, *save: bool = True*) → *mlrun.projects.project.MlrunProject*

Load a project from MLRun DB, or create/import if doesnt exist

example:

```
# load project from the DB (if exist) or the source repo
project = get_or_create_project("myproj", "./", "git://github.com/mlrun/demo-xgb-
↪project.git")
```

(continues on next page)

(continued from previous page)

```
project.pull("development")  # pull the latest code from git
project.run("main", arguments={'data': data_url})  # run the workflow "main"
```

Parameters

- **name** – project name
- **context** – project local directory path (Default value = "./")
- **url** – name (in DB) or git or tar.gz or .zip sources archive path e.g.: git://github.com/mlrun/demo-xgb-project.git http://mysite/archived-project.zip
- **secrets** – key:secret dict or SecretsStore used to download sources
- **init_git** – if True, will git init the context dir
- **subpath** – project subpath (within the archive/context)
- **clone** – if True, always clone (delete any existing content)
- **user_project** – add the current user name to the project name (for db:// prefixes)
- **from_template** – path to project YAML file that will be used as from_template (for new projects)
- **save** – whether to save the created project in the DB

Returns project object

mlrun.projects.**load_project**(*context: str = './'*, *url: Optional[str] = None*, *name: Optional[str] = None*, *secrets: Optional[dict] = None*, *init_git: bool = False*, *subpath: Optional[str] = None*, *clone: bool = False*, *user_project: bool = False*, *save: bool = True*) → *mlrun.projects.project.MlrunProject*

Load an MLRun project from git or tar or dir

example:

```
# Load the project and run the 'main' workflow.
# When using git as the url source the context directory must be an empty or
# non-existent folder as the git repo will be cloned there
project = load_project("./demo_proj", "git://github.com/mlrun/project-demo.git")
project.run("main", arguments={'data': data_url})
```

Parameters

- **context** – project local directory path
- **url** – name (in DB) or git or tar.gz or .zip sources archive path e.g.: git://github.com/mlrun/demo-xgb-project.git http://mysite/archived-project.zip <project-name> The git project should include the project yaml file. If the project yaml file is in a sub-directory, must specify the sub-directory.
- **name** – project name
- **secrets** – key:secret dict or SecretsStore used to download sources
- **init_git** – if True, will git init the context dir
- **subpath** – project subpath (within the archive)
- **clone** – if True, always clone (delete any existing content)

- **user_project** – add the current user name to the project name (for db:// prefixes)
- **save** – whether to save the created project and artifact in the DB

Returns project object

mlrun.projects.**new_project**(*name*, *context: str = './'*, *init_git: bool = False*, *user_project: bool = False*, *remote: Optional[str] = None*, *from_template: Optional[str] = None*, *secrets: Optional[dict] = None*, *description: Optional[str] = None*, *subpath: Optional[str] = None*, *save: bool = True*, *overwrite: bool = False*) → *mlrun.projects.project.MlrunProject*

Create a new MLRun project, optionally load it from a yaml/zip/git template

example:

```
# create a project with local and marketplace functions, a workflow, and an artifact
project = mlrun.new_project("myproj", "./", init_git=True, description="my new␣
↪project")
project.set_function('prep_data.py', 'prep-data', image='mlrun/mlrun', handler=
↪'prep_data')
project.set_function('hub://sklearn_classifier', 'train')
project.set_artifact('data', Artifact(target_path=data_url))
project.set_workflow('main', "./myflow.py")
project.save()

# run the "main" workflow (watch=True to wait for run completion)
project.run("main", watch=True)
```

example (load from template):

```
# create a new project from a zip template (can also use yaml/git templates)
# initialize a local git, and register the git remote path
project = mlrun.new_project("myproj", "./", init_git=True,
                            remote="git://github.com/mlrun/project-demo.git",
                            from_template="http://mysite/proj.zip")
project.run("main", watch=True)
```

Parameters

- **name** – project name
- **context** – project local directory path
- **init_git** – if True, will git init the context dir
- **user_project** – add the current user name to the provided project name (making it unique per user)
- **remote** – remote Git url
- **from_template** – path to project YAML/zip file that will be used as a template
- **secrets** – key:secret dict or SecretsStore used to download sources
- **description** – text describing the project
- **subpath** – project subpath (relative to the context dir)
- **save** – whether to save the created project in the DB

- **overwrite** – overwrite project using 'cascade' deletion strategy (deletes project resources) if project with name exists

Returns project object

mlrun.projects.**run_function**(*function: Union[str, mlrun.runtimes.base.BaseRuntime]*, *handler: Optional[str] = None*, *name: str = ''*, *params: Optional[dict] = None*, *hyperparams: Optional[dict] = None*, *hyper_param_options: Optional[mlrun.model.HyperParamOptions] = None*, *inputs: Optional[dict] = None*, *outputs: Optional[List[str]] = None*, *workdir: str = ''*, *labels: Optional[dict] = None*, *base_task: Optional[mlrun.model.RunTemplate] = None*, *watch: bool = True*, *local: Optional[bool] = None*, *verbose: Optional[bool] = None*, *selector: Optional[str] = None*, *project_object=None*, *auto_build: Optional[bool] = None*, *schedule: Optional[Union[str, mlrun.api.schemas.schedule.ScheduleCronTrigger]] = None*, *artifact_path: Optional[str] = None*) → Union[*mlrun.model.RunObject*, kfp.dsl._container_op.ContainerOp]

Run a local or remote task as part of a local/kubeflow pipeline

run_function() allow you to execute a function locally, on a remote cluster, or as part of an automated workflow function can be specified as an object or by name (str), when the function is specified by name it is looked up in the current project eliminating the need to redefine/edit functions.

when functions run as part of a workflow/pipeline (project.run()) some attributes can be set at the run level, e.g. local=True will run all the functions locally, setting artifact_path will direct all outputs to the same path. project runs provide additional notifications/reporting and exception handling. inside a Kubeflow pipeline (KFP) run_function() generates KFP "ContainerOps" which are used to form a DAG some behavior may differ between regular runs and deferred KFP runs.

example (use with function object):

```
function = mlrun.import_function("hub://sklearn_classifier")
run1 = run_function(function, params={"data": url})
```

example (use with project):

```
# create a project with two functions (local and from marketplace)
project = mlrun.new_project(project_name, "./proj)
project.set_function("mycode.py", "myfunc", image="mlrun/mlrun")
project.set_function("hub://sklearn_classifier", "train")

# run functions (refer to them by name)
run1 = run_function("myfunc", params={"x": 7})
run2 = run_function("train", params={"data": run1.outputs["data"]})
```

example (use in pipeline):

```
@dsl.pipeline(name="test pipeline", description="test")
def my_pipe(url=""):
    run1 = run_function("loaddata", params={"url": url})
    run2 = run_function("train", params={"data": run1.outputs["data"]})

project.run(workflow_handler=my_pipe, arguments={"param1": 7})
```

Parameters

- **function** – name of the function (in the project) or function object

- **handler** – name of the function handler
- **name** – execution name
- **params** – input parameters (dict)
- **hyperparams** – hyper parameters
- **selector** – selection criteria for hyper params e.g. "max.accuracy"
- **hyper_param_options** – hyper param options (selector, early stop, strategy, ..) see: *HyperParamOptions*
- **inputs** – input objects (dict of key: path)
- **outputs** – list of outputs which can pass in the workflow
- **workdir** – default input artifacts path
- **labels** – labels to tag the job/run with ({key:val, ..})
- **base_task** – task object to use as base
- **watch** – watch/follow run log, True by default
- **local** – run the function locally vs on the runtime/cluster
- **verbose** – add verbose prints/logs
- **project_object** – override the project object to use, will default to the project set in the runtime context.
- **auto_build** – when set to True and the function require build it will be built on the first function run, use only if you dont plan on changing the build config between runs
- **schedule** – ScheduleCronTrigger class instance or a standard crontab expression string (which will be converted to the class using its *from_crontab* constructor), see this link for help: https://apscheduler.readthedocs.io/en/v3.6.3/modules/triggers/cron.html#module-apscheduler.triggers.cron
- **artifact_path** – path to store artifacts, when running in a workflow this will be set automatically

Returns MLRun RunObject or KubeFlow containerOp

17.12 mlrun.run

class mlrun.run.**ArtifactType**(*value*)

Bases: enum.Enum

Possible artifact types to log using the MLRun *context* decorator.

DATASET = 'dataset'

DEFAULT = 'result'

DIRECTORY = 'directory'

FILE = 'file'

OBJECT = 'object'

PLOT = 'plot'

RESULT = 'result'

class mlrun.run.**ContextHandler**

Bases: object

Private class for handling an MLRun context of a function that is wrapped in MLRun's *handler* decorator.

The context handler have 3 duties:

1. Check if the user used MLRun to run the wrapped function and if so, get the MLRun context.
2. Parse the user's inputs (MLRun *DataItem*) to the function.
3. Log the function's outputs to MLRun.

The context handler use dictionaries to map objects to their logging / parsing function. The maps can be edited using the relevant *update_X* class method. If needed to add additional artifacts types, the *ArtifactType* class can be inherited and replaced as well using the *update_artifact_type_class* class method.

Initialize a context handler.

is_context_available() → bool

Check if a context was found by the method *look_for_context*.

Returns True if a context was found and False otherwise.

log_outputs(*outputs: list*, *logging_instructions: List[Optional[Union[Tuple[str,* mlrun.run.ArtifactType*], Tuple[str, str], Tuple[str,* mlrun.run.ArtifactType*, Dict[str, Any]], Tuple[str, str, Dict[str, Any]], str]]]*)

Log the given outputs as artifacts with the stored context.

Parameters

- **outputs** – List of outputs to log.
- **logging_instructions** – List of logging instructions to use.

look_for_context(*args: tuple*, *kwargs: dict*)

Look for an MLRun context (*mlrun.MLClientCtx*). The handler will look for a context in the given order:

1. The given arguments.
2. The given keyword arguments.
3. If an MLRun RunTime was used the context will be located via the *mlrun.get_or_create_ctx* method.

Parameters

- **args** – The arguments tuple passed to the function.
- **kwargs** – The keyword arguments dictionary passed to the function.

parse_inputs(*args: tuple*, *kwargs: dict*, *expected_arguments_types: collections.OrderedDict*) → tuple

Parse the given arguments and keyword arguments data items to the expected types.

Parameters

- **args** – The arguments tuple passed to the function.
- **kwargs** – The keyword arguments dictionary passed to the function.
- **expected_arguments_types** – An ordered dictionary of the expected types of arguments.

Returns The parsed args (kwargs are parsed inplace).

set_labels(*labels: Dict[str, str]*)
Set the given labels with the stored context.

Parameters **labels** – The labels to set.

classmethod update_artifact_type_class(*artifact_type_class: Type[*mlrun.run.ArtifactType*]*)
Update the artifact type enum class that the handler will use to specify new artifact types to log and parse.

Parameters **artifact_type_class** – An enum inheriting from the *ArtifactType* enum.

classmethod update_default_objects_artifact_types_map(*updates: Dict[type,* mlrun.run.ArtifactType*]*)
Enrich the default objects artifact types map with new objects types to support.

Parameters **updates** – New objects types to artifact types to support.

classmethod update_inputs_parsing_map(*updates: Dict[type, Callable[[*mlrun.datastore.base.DataItem*], Any]]*)
Enrich the inputs parsing map with new objects to support. The inputs parsing map is a dictionary of object types as key, and a function that will handle the given input. The function must accept 1 keyword argument (data_item: *mlrun.DataItem*) and return the relevant parsed object.

Parameters **updates** – New object types to support - a dictionary of artifact type enum as key, and a function that will handle the given input to update the current map.

classmethod update_outputs_logging_map(*updates: Dict[*mlrun.run.ArtifactType*, Callable[[*mlrun.execution.MLClientCtx*, Any, str, dict], None]]*)
Enrich the outputs logging map with new artifact types to support. The outputs logging map is a dictionary of artifact type enum as key, and a function that will handle the given output. The function must accept 4 keyword arguments

- ctx: *mlrun.MLClientCtx* - The MLRun context to log with.
- obj: *Any* - The value / object to log.
- key: *str* - The key of the artifact.
- logging_kwargs: *dict* - Keyword arguments the user can pass in the instructions tuple.

Parameters **updates** – New artifact types to support - a dictionary of artifact type enum as key, and a function that will handle the given output to update the current map.

class mlrun.run.**InputsParser**
Bases: object

A static class to hold all the common parsing functions - functions for parsing MLRun DataItem to the user desired type.

static parse_dict(*data_item:* mlrun.datastore.base.DataItem) → dict
Parse an MLRun *DataItem* to a *dict*.

Parameters **data_item** – The *DataItem* to parse.

Returns The *DataItem* as a *dict*.

static parse_list(*data_item:* mlrun.datastore.base.DataItem) → list
Parse an MLRun *DataItem* to a *list*.

Parameters **data_item** – The *DataItem* to parse.

Returns The *DataItem* as a *list*.

static parse_numpy_array(*data_item:* mlrun.datastore.base.DataItem) → numpy.ndarray
Parse an MLRun *DataItem* to a *numpy.ndarray*.

Parameters **data_item** – The *DataItem* to parse.

Returns The *DataItem* as a *numpy.ndarray*.

static parse_object(*data_item:* mlrun.datastore.base.DataItem) → object
Parse an MLRun *DataItem* to its unpickled object. The pickle file will be downloaded to a local temp directory and then loaded.

Parameters **data_item** – The *DataItem* to parse.

Returns The *DataItem* as the original object that was pickled once it was logged.

static parse_pandas_dataframe(*data_item:* mlrun.datastore.base.DataItem) → pandas.core.frame.DataFrame
Parse an MLRun *DataItem* to a *pandas.DataFrame*.

Parameters **data_item** – The *DataItem* to parse.

Returns The *DataItem* as a *pandas.DataFrame*.

class mlrun.run.**OutputsLogger**
Bases: object

A static class to hold all the common logging functions - functions for logging different objects by artifact type to MLRun.

static log_dataset(*ctx:* mlrun.execution.MLClientCtx, *obj: Union[pandas.core.frame.DataFrame, numpy.ndarray, pandas.core.series.Series, dict, list]*, *key: str*, *logging_kwargs: dict*)
Log an object as a dataset. The dataset wil lbe cast to a *pandas.DataFrame*. Supporting casting from *pandas.Series*, *numpy.ndarray*, *dict* and *list*.

Parameters

- **ctx** – The MLRun context to log with.
- **obj** – The data to log.
- **key** – The key of the artifact.
- **logging_kwargs** – Additional keyword arguments to pass to the *context.log_dataset*

Raises **MLRunInvalidArgumentError** – If the type is not supported for being cast to *pandas.DataFrame*.

static log_directory(*ctx:* mlrun.execution.MLClientCtx, *obj: Union[str, pathlib.Path]*, *key: str*, *logging_kwargs: dict*)
Log a directory as a zip file. The zip file will be created at the current working directory. Once logged, it will be deleted.

Parameters

- **ctx** – The MLRun context to log with.
- **obj** – The directory to zip path.
- **key** – The key of the artifact.
- **logging_kwargs** – Additional keyword arguments to pass to the *context.log_artifact* method.

Raises **MLRunInvalidArgumentError** – In case the given path is not of a directory or do not exist.

static log_file(*ctx:* mlrun.execution.MLClientCtx, *obj: Union[str, pathlib.Path]*, *key: str*, *logging_kwargs: dict*)

Log a file to MLRun.

Parameters

- **ctx** – The MLRun context to log with.
- **obj** – The path of the file to log.
- **key** – The key of the artifact.
- **logging_kwargs** – Additional keyword arguments to pass to the *context.log_artifact* method.

Raises **MLRunInvalidArgumentError** – In case the given path is not of a file or do not exist.

static log_object(*ctx:* mlrun.execution.MLClientCtx, *obj*, *key: str*, *logging_kwargs: dict*)

Log an object as a pickle.

Parameters

- **ctx** – The MLRun context to log with.
- **obj** – The object to log.
- **key** – The key of the artifact.
- **logging_kwargs** – Additional keyword arguments to pass to the *context.log_artifact* method.

static log_plot(*ctx:* mlrun.execution.MLClientCtx, *obj*, *key: str*, *logging_kwargs: dict*)

Log an object as a plot. Currently, supporting plots produced by one the following modules: *matplotlib*, *seaborn*, *plotly* and *bokeh*.

Parameters

- **ctx** – The MLRun context to log with.
- **obj** – The plot to log.
- **key** – The key of the artifact.
- **logging_kwargs** – Additional keyword arguments to pass to the *context.log_artifact*.

Raises **MLRunInvalidArgumentError** – If the object type is not supported (meaning the plot was not produced by one of the supported modules).

static log_result(*ctx:* mlrun.execution.MLClientCtx, *obj: Union[int, float, str, list, tuple, dict, numpy.ndarray]*, *key: str*, *logging_kwargs: dict*)

Log an object as a result. The objects value will be cast to a serializable version of itself. Supporting: int, float, str, list, tuple, dict, numpy.ndarray

Parameters

- **ctx** – The MLRun context to log with.
- **obj** – The value to log.
- **key** – The key of the artifact.
- **logging_kwargs** – Additional keyword arguments to pass to the *context.log_result* method.

class mlrun.run.**RunStatuses**

Bases: object

static all()

error = 'Error'

failed = 'Failed'

running = 'Running'

skipped = 'Skipped'

static stable_statuses()

succeeded = 'Succeeded'

static transient_statuses()

mlrun.run.**code_to_function**(*name: str = ''*, *project: str = ''*, *tag: str = ''*, *filename: str = ''*, *handler: str = ''*, *kind: str = ''*, *image: Optional[str] = None*, *code_output: str = ''*, *embed_code: bool = True*, *description: str = ''*, *requirements: Optional[Union[str, List[str]]] = None*, *categories: Optional[List[str]] = None*, *labels: Optional[Dict[str, str]] = None*, *with_doc: bool = True*, *ignored_tags=None*) → Union[mlrun.runtimes.mpijob.v1alpha1.MpiRuntimeV1Alpha1, mlrun.runtimes.mpijob.v1.MpiRuntimeV1, *mlrun.runtimes.function.RemoteRuntime*, *mlrun.runtimes.serving.ServingRuntime*, *mlrun.runtimes.daskjob.DaskCluster*, *mlrun.runtimes.kubejob.KubejobRuntime*, *mlrun.runtimes.local.LocalRuntime*, mlrun.runtimes.sparkjob.spark2job.Spark2Runtime, mlrun.runtimes.sparkjob.spark3job.Spark3Runtime, *mlrun.runtimes.remotesparkjob.RemoteSparkRuntime*]

Convenience function to insert code and configure an mlrun runtime.

Easiest way to construct a runtime type object. Provides the most often used configuration options for all runtimes as parameters.

Instantiated runtimes are considered 'functions' in mlrun, but they are anything from nuclio functions to generic kubernetes pods to spark jobs. Functions are meant to be focused, and as such limited in scope and size. Typically a function can be expressed in a single python module with added support from custom docker images and commands for the environment. The returned runtime object can be further configured if more customization is required.

One of the most important parameters is 'kind'. This is what is used to specify the chosen runtimes. The options are:

- local: execute a local python or shell script
- job: insert the code into a Kubernetes pod and execute it
- nuclio: insert the code into a real-time serverless nuclio function
- serving: insert code into orchestrated nuclio function(s) forming a DAG
- dask: run the specified python code / script as Dask Distributed job
- mpijob: run distributed Horovod jobs over the MPI job operator
- spark: run distributed Spark job using Spark Kubernetes Operator
- remote-spark: run distributed Spark job on remote Spark service

Learn more about function runtimes here: https://docs.mlrun.org/en/latest/runtimes/functions.html#function-runtimes

Parameters

- **name** – function name, typically best to use hyphen-case

- **project** – project used to namespace the function, defaults to 'default'
- **tag** – function tag to track multiple versions of the same function, defaults to 'latest'
- **filename** – path to .py/.ipynb file, defaults to current jupyter notebook
- **handler** – The default function handler to call for the job or nuclio function, in batch functions (job, mpijob, ..) the handler can also be specified in the *.run()* command, when not specified the entire file will be executed (as main). for nuclio functions the handler is in the form of module:function, defaults to 'main:handler'
- **kind** – function runtime type string - nuclio, job, etc. (see docstring for all options)
- **image** – base docker image to use for building the function container, defaults to None
- **code_output** – specify '.' to generate python module from the current jupyter notebook
- **embed_code** – indicates whether or not to inject the code directly into the function runtime spec, defaults to True
- **description** – short function description, defaults to ''
- **requirements** – list of python packages or pip requirements file path, defaults to None
- **categories** – list of categories for mlrun Function Hub, defaults to None
- **labels** – immutable name/value pairs to tag the function with useful metadata, defaults to None
- **with_doc** – indicates whether to document the function parameters, defaults to True
- **ignored_tags** – notebook cells to ignore when converting notebooks to py code (separated by ';')

Returns pre-configured function object from a mlrun runtime class

example:

```
import mlrun

# create job function object from notebook code and add doc/metadata
fn = mlrun.code_to_function("file_utils", kind="job",
                            handler="open_archive", image="mlrun/mlrun",
                            description = "this function opens a zip archive into a␣
↪local/mounted folder",
                            categories = ["fileutils"],
                            labels = {"author": "me"})
```

example:

```
import mlrun
from pathlib import Path

# create file
Path("mover.py").touch()

# create nuclio function object from python module call mover.py
fn = mlrun.code_to_function("nuclio-mover", kind="nuclio",
                            filename="mover.py", image="python:3.7",
                            description = "this function moves files from one␣
↪system to another",
```

(continues on next page)

(continued from previous page)

```
                                requirements = ["pandas"],
                                labels = {"author": "me"})
```

mlrun.run.**download_object**(*url*, *target*, *secrets=None*)
download mlrun dataitem (from path/url to target path)

mlrun.run.**function_to_module**(*code=''*, *workdir=None*, *secrets=None*, *silent=False*)
Load code, notebook or mlrun function as .py module this function can import a local/remote py file or notebook or load an mlrun function object as a module, you can use this from your code, notebook, or another function (for common libs)

Note: the function may have package requirements which must be satisfied

example:

```
mod = mlrun.function_to_module('./examples/training.py')
task = mlrun.new_task(inputs={'infile.txt': '../examples/infile.txt'})
context = mlrun.get_or_create_ctx('myfunc', spec=task)
mod.my_job(context, p1=1, p2='x')
print(context.to_yaml())

fn = mlrun.import_function('hub://open_archive')
mod = mlrun.function_to_module(fn)
data = mlrun.run.get_dataitem("https://fpsignals-public.s3.amazonaws.com/catsndogs.
↪tar.gz")
context = mlrun.get_or_create_ctx('myfunc')
mod.open_archive(context, archive_url=data)
print(context.to_yaml())
```

Parameters

- **code** – path/url to function (.py or .ipynb or .yaml) OR function object
- **workdir** – code workdir
- **secrets** – secrets needed to access the URL (e.g.s3, v3io, ..)
- **silent** – do not raise on errors

Returns python module

mlrun.run.**get_dataitem**(*url*, *secrets=None*, *db=None*) → *mlrun.datastore.base.DataItem*
get mlrun dataitem object (from path/url)

mlrun.run.**get_object**(*url*, *secrets=None*, *size=None*, *offset=0*, *db=None*)
get mlrun dataitem body (from path/url)

mlrun.run.**get_or_create_ctx**(*name: str*, *event=None*, *spec=None*, *with_env: bool = True*, *rundb: str = ''*, *project: str = ''*, *upload_artifacts=False*)
called from within the user program to obtain a run context

the run context is an interface for receiving parameters, data and logging run results, the run context is read from the event, spec, or environment (in that order), user can also work without a context (local defaults mode)

all results are automatically stored in the "rundb" or artifact store, the path to the rundb can be specified in the call or obtained from env.

Parameters

- **name** – run name (will be overridden by context)
- **event** – function (nuclio Event object)
- **spec** – dictionary holding run spec
- **with_env** – look for context in environment vars, default True
- **rundb** – path/url to the metadata and artifact database
- **project** – project to initiate the context in (by default mlrun.mlctx.default_project)
- **upload_artifacts** – when using local context (not as part of a job/run), upload artifacts to the system default artifact path location

Returns execution context

Examples:

```
# load MLRUN runtime context (will be set by the runtime framework e.g. KubeFlow)
context = get_or_create_ctx('train')

# get parameters from the runtime context (or use defaults)
p1 = context.get_param('p1', 1)
p2 = context.get_param('p2', 'a-string')

# access input metadata, values, files, and secrets (passwords)
print(f'Run: {context.name} (uid={context.uid})')
print(f'Params: p1={p1}, p2={p2}')
print(f'accesskey = {context.get_secret("ACCESS_KEY")}')
input_str = context.get_input('infile.txt').get()
print(f'file: {input_str}')

# RUN some useful code e.g. ML training, data prep, etc.

# log scalar result values (job result metrics)
context.log_result('accuracy', p1 * 2)
context.log_result('loss', p1 * 3)
context.set_label('framework', 'sklearn')

# log various types of artifacts (file, web page, table), will be versioned and␣
↪visible in the UI
context.log_artifact('model.txt', body=b'abc is 123', labels={'framework': 'xgboost
↪'})
context.log_artifact('results.html', body=b'<b> Some HTML <b>', viewer='web-app')
```

mlrun.run.**get_pipeline**(*run_id*, *namespace=None*, *format_: Union[str, mlrun.api.schemas.pipeline.PipelinesFormat] = PipelinesFormat.summary*, *project: Optional[str] = None*, *remote: bool = True*)

Get Pipeline status

Parameters

- **run_id** – id of pipelines run
- **namespace** – k8s namespace if not default
- **format** – Format of the results. Possible values are: - `summary` (default value) - Return summary of the object data. - `full` - Return full pipeline object.
- **project** – the project of the pipeline run

- **remote** – read kfp data from mlrun service (default=True)

Returns kfp run dict

mlrun.run.**handler**(*labels: Optional[Dict[str, str]] = None*, *outputs: Optional[List[Optional[Union[Tuple[str, mlrun.run.ArtifactType], Tuple[str, str], Tuple[str, mlrun.run.ArtifactType, Dict[str, Any]], Tuple[str, str, Dict[str, Any]], str]]]] = None*, *inputs: Union[bool, Dict[str, Type]] = True*)

MLRun's handler is a decorator to wrap a function and enable setting labels, automatic *mlrun.DataItem* parsing and outputs logging.

Parameters

- **labels** – Labels to add to the run. Expecting a dictionary with the labels names as keys. Default: None.
- **outputs** – Logging configurations for the function's returned values. Expecting a list of tuples and None values:
  - **str - A string in the format of '{key}:{artifact_type}'. If a string was given without ':' it will** indicate the key and the artifact type will be according to the returned value type.
  - tuple - A tuple of:
    - [0]: str - The key (name) of the artifact to use for the logged output.
    - [1]: Union[*ArtifactType*, str] = "result" - An *ArtifactType* enum or an equivalent string, that indicates how to log the returned value. The artifact types can be one of:
      - DATASET = "dataset"
      - DIRECTORY = "directory"
      - FILE = "file"
      - OBJECT = "object"
      - PLOT = "plot"
      - RESULT = "result".
    - [2]: Optional[Dict[str, Any]] - A keyword arguments dictionary with the properties to pass to the relevant logging function (one of *context.log_artifact*, *context.log_result*, *context.log_dataset*).
  - None - Do not log the output.

  The list length must be equal to the total amount of returned values from the function. Default is None - meaning no outputs will be logged.
- **inputs** – Parsing configurations for the arguments passed as inputs via the *run* method of an MLRun function. Can be passed as a boolean value or a dictionary:
  - **True - Parse all found inputs to the assigned type hint in the function's signature. If there is no** type hint assigned, the value will remain an *mlrun.DataItem*.
  - False - Do not parse inputs, leaving the inputs as *mlrun.DataItem*.
  - **Dict[str, Type] - A dictionary with argument name as key and the expected type to parse the** *mlrun.DataItem* to.

  Default: True.

Example:

```
import mlrun

@mlrun.handler(outputs=["my_array", None, "my_multiplier"])
def my_handler(array: np.ndarray, m: int):
    array = array * m
    m += 1
    return array, "I won't be logged", m

>>> mlrun_function = mlrun.code_to_function("my_code.py", kind="job")
>>> run_object = mlrun_function.run(
...     handler="my_handler",
...     inputs={"array": "store://my_array_Artifact"},
...     params={"m": 2}
... )
>>> run_object.outputs
{'my_multiplier': 3, 'my_array': 'store://...'}
```

mlrun.run.**import_function**(*url=''*, *secrets=None*, *db=''*, *project=None*, *new_name=None*)

Create function object from DB or local/remote YAML file

Functions can be imported from function repositories (mlrun Function Hub (formerly Marketplace) or local db), or be read from a remote URL (http(s), s3, git, v3io, ..) containing the function YAML

special URLs:

```
function hub: hub://{name}[:{tag}]
local mlrun db:       db://{project-name}/{name}[:{tag}]
```

examples:

```
function = mlrun.import_function("hub://sklearn_classifier")
function = mlrun.import_function("./func.yaml")
function = mlrun.import_function("https://raw.githubusercontent.com/org/repo/func.
↪yaml")
```

Parameters

- **url** – path/url to Function Hub, db or function YAML file
- **secrets** – optional, credentials dict for DB or URL (s3, v3io, ...)
- **db** – optional, mlrun api/db path
- **project** – optional, target project for the function
- **new_name** – optional, override the imported function name

Returns function object

mlrun.run.**import_function_to_dict**(*url*, *secrets=None*)

Load function spec from local/remote YAML file

mlrun.run.**list_pipelines**(*full=False*, *page_token=''*, *page_size=None*, *sort_by=''*, *filter_=''*, *namespace=None*, *project='*'*, *format_: mlrun.api.schemas.pipeline.PipelinesFormat = PipelinesFormat.metadata_only*) → Tuple[int, Optional[int], List[dict]]

List pipelines

Parameters

- **full** – Deprecated, use *format_* instead. if True will set *format_* to full, otherwise *format_* will be used
- **page_token** – A page token to request the next page of results. The token is acquired from the nextPageToken field of the response from the previous call or can be omitted when fetching the first page.
- **page_size** – The number of pipelines to be listed per page. If there are more pipelines than this number, the response message will contain a nextPageToken field you can use to fetch the next page.
- **sort_by** – Can be format of "field_name", "field_name asc" or "field_name desc" (Example, "name asc" or "id desc"). Ascending by default.
- **filter** – A url-encoded, JSON-serialized Filter protocol buffer, see: [filter.proto](https://github.com/kubeflow/pipelines/ blob/master/backend/api/filter.proto).
- **namespace** – Kubernetes namespace if other than default
- **project** – Can be used to retrieve only specific project pipelines. "*" for all projects. Note that filtering by project can't be used together with pagination, sorting, or custom filter.
- **format** – Control what will be returned (full/metadata_only/name_only)

mlrun.run.**load_func_code**(*command=''*, *workdir=None*, *secrets=None*, *name='name'*)

mlrun.run.**new_function**(*name: str = ''*, *project: str = ''*, *tag: str = ''*, *kind: str = ''*, *command: str = ''*, *image: str = ''*, *args: Optional[list] = None*, *runtime=None*, *mode=None*, *handler: Optional[str] = None*, *source: Optional[str] = None*, *requirements: Optional[Union[str, List[str]]] = None*, *kfp=None*)

Create a new ML function from base properties

example:

```
# define a container based function (the `training.py` must exist in the container␣
↪workdir)
f = new_function(command='training.py -x {x}', image='myrepo/image:latest', kind=
↪'job')
f.run(params={"x": 5})

# define a container based function which reads its source from a git archive
f = new_function(command='training.py -x {x}', image='myrepo/image:latest', kind=
↪'job',
                 source='git://github.com/mlrun/something.git')
f.run(params={"x": 5})

# define a local handler function (execute a local function handler)
f = new_function().run(task, handler=myfunction)
```

Parameters

- **name** – function name
- **project** – function project (none for 'default')
- **tag** – function version tag (none for 'latest')
- **kind** – runtime type (local, job, nuclio, spark, mpijob, dask, ..)

- **command** – command/url + args (e.g.: training.py –verbose)
- **image** – container image (start with '.' for default registry)
- **args** – command line arguments (override the ones in command)
- **runtime** – runtime (job, nuclio, spark, dask ..) object/dict store runtime specific details and preferences
- **mode** –

  runtime mode, "args" mode will push params into command template, example:
  command=`mycode.py –x {xparam}` will substitute the *{xparam}* with the value of the xparam param

  "pass" mode will run the command as is in the container (not wrapped by mlrun), the command can use *{}* for parameters like in the "args" mode
- **handler** – The default function handler to call for the job or nuclio function, in batch functions (job, mpijob, ..) the handler can also be specified in the *.run()* command, when not specified the entire file will be executed (as main). for nuclio functions the handler is in the form of module:function, defaults to "main:handler"
- **source** – valid path to git, zip, or tar file, e.g. *git://github.com/mlrun/something.git*, *http://some/url/file.zip*
- **requirements** – list of python packages or pip requirements file path, defaults to None
- **kfp** – reserved, flag indicating running within kubeflow pipeline

Returns function object

mlrun.run.**run_local**(*task=None*, *command=''*, *name: str = ''*, *args: Optional[list] = None*, *workdir=None*, *project: str = ''*, *tag: str = ''*, *secrets=None*, *handler=None*, *params: Optional[dict] = None*, *inputs: Optional[dict] = None*, *artifact_path: str = ''*, *mode: Optional[str] = None*, *allow_empty_resources=None*)

Run a task on function/code (.py, .ipynb or .yaml) locally,

example:

```
# define a task
task = new_task(params={'p1': 8}, out_path=out_path)
# run
run = run_local(spec, command='src/training.py', workdir='src')
```

or specify base task parameters (handler, params, ..) in the call:

```
run = run_local(handler=my_function, params={'x': 5})
```

Parameters

- **task** – task template object or dict (see RunTemplate)
- **command** – command/url/function
- **name** – ad hook function name
- **args** – command line arguments (override the ones in command)
- **workdir** – working dir to exec in
- **project** – function project (none for 'default')

- **tag** – function version tag (none for 'latest')
- **secrets** – secrets dict if the function source is remote (s3, v3io, ..)
- **handler** – pointer or name of a function handler
- **params** – input parameters (dict)
- **inputs** – input objects (dict of key: path)
- **artifact_path** – default artifact output path

Returns run object

mlrun.run.**run_pipeline**(*pipeline*, *arguments=None*, *project=None*, *experiment=None*, *run=None*, *namespace=None*, *artifact_path=None*, *ops=None*, *url=None*, *ttl=None*, *remote: bool = True*)

remote KubeFlow pipeline execution

Submit a workflow task to KFP via mlrun API service

Parameters

- **pipeline** – KFP pipeline function or path to .yaml/.zip pipeline file
- **arguments** – pipeline arguments
- **project** – name of project
- **experiment** – experiment name
- **run** – optional, run name
- **namespace** – Kubernetes namespace (if not using default)
- **url** – optional, url to mlrun API service
- **artifact_path** – target location/url for mlrun artifacts
- **ops** – additional operators (.apply() to all pipeline functions)
- **ttl** – pipeline ttl in secs (after that the pods will be removed)
- **remote** – read kfp data from mlrun service (default=True)

Returns kubeflow pipeline id

mlrun.run.**wait_for_pipeline_completion**(*run_id*, *timeout=3600*, *expected_statuses: Optional[List[str]] = None*, *namespace=None*, *remote=True*, *project: Optional[str] = None*)

Wait for Pipeline status, timeout in sec

Parameters

- **run_id** – id of pipelines run
- **timeout** – wait timeout in sec
- **expected_statuses** – list of expected statuses, one of [Succeeded | Failed | Skipped | Error], by default [Succeeded]
- **namespace** – k8s namespace if not default
- **remote** – read kfp data from mlrun service (default=True)
- **project** – the project of the pipeline

Returns kfp run dict

mlrun.run.**wait_for_runs_completion**(*runs: list*, *sleep=3*, *timeout=0*, *silent=False*)
wait for multiple runs to complete

Note: need to use *watch=False* in *.run()* so the run will not wait for completion

example:

```
# run two training functions in parallel and wait for the results
inputs = {'dataset': cleaned_data}
run1 = train.run(name='train_lr', inputs=inputs, watch=False,
                 params={'model_pkg_class': 'sklearn.linear_model.LogisticRegression
↪',
                         'label_column': 'label'})
run2 = train.run(name='train_lr', inputs=inputs, watch=False,
                 params={'model_pkg_class': 'sklearn.ensemble.RandomForestClassifier
↪',
                         'label_column': 'label'})
completed = wait_for_runs_completion([run1, run2])
```

Parameters

- **runs** – list of run objects (the returned values of function.run())
- **sleep** – time to sleep between checks (in seconds)
- **timeout** – maximum time to wait in seconds (0 for unlimited)
- **silent** – set to True for silent exit on timeout

Returns list of completed runs

17.13 mlrun.runtimes

class mlrun.runtimes.**BaseRuntime**(*metadata=None*, *spec=None*)
Bases: mlrun.model.ModelObj

as_step(*runspec: Optional[mlrun.model.RunObject] = None*, *handler=None*, *name: str = ''*, *project: str = ''*, *params: Optional[dict] = None*, *hyperparams=None*, *selector=''*, *hyper_param_options: Optional[mlrun.model.HyperParamOptions] = None*, *inputs: Optional[dict] = None*, *outputs: Optional[dict] = None*, *workdir: str = ''*, *artifact_path: str = ''*, *image: str = ''*, *labels: Optional[dict] = None*, *use_db=True*, *verbose=None*, *scrape_metrics=False*)
Run a local or remote task.

Parameters

- **runspec** – run template object or dict (see RunTemplate)
- **handler** – name of the function handler
- **name** – execution name
- **project** – project name
- **params** – input parameters (dict)
- **hyperparams** – hyper parameters
- **selector** – selection criteria for hyper params

- **hyper_param_options** – hyper param options (selector, early stop, strategy, ..) see: *HyperParamOptions*
- **inputs** – input objects (dict of key: path)
- **outputs** – list of outputs which can pass in the workflow
- **artifact_path** – default artifact output path (replace out_path)
- **workdir** – default input artifacts path
- **image** – container image to use
- **labels** – labels to tag the job/run with ({key:val, ..})
- **use_db** – save function spec in the db (vs the workflow file)
- **verbose** – add verbose prints/logs
- **scrape_metrics** – whether to add the *mlrun/scrape-metrics* label to this run's resources

Returns KubeFlow containerOp

clean_build_params()

doc()

export(*target=''*, *format='.yaml'*, *secrets=None*, *strip=True*)
save function spec to a local/remote path (default to./function.yaml)

Parameters

- **target** – target path/url
- **format** – *.yaml* (default) or *.json*
- **secrets** – optional secrets dict/object for target path (e.g. s3)
- **strip** – strip status data

Returns self

fill_credentials()

full_image_path(*image=None*, *client_version: Optional[str] = None*)

is_deployed()

kind = 'base'

property metadata: mlrun.model.BaseMetadata

run(*runspec: Optional[*mlrun.model.RunObject*] = None*, *handler=None*, *name: str = ''*, *project: str = ''*, *params: Optional[dict] = None*, *inputs: Optional[Dict[str, str]] = None*, *out_path: str = ''*, *workdir: str = ''*, *artifact_path: str = ''*, *watch: bool = True*, *schedule: Optional[Union[str, mlrun.api.schemas.schedule.ScheduleCronTrigger]] = None*, *hyperparams: Optional[Dict[str, list]] = None*, *hyper_param_options: Optional[*mlrun.model.HyperParamOptions*] = None*, *verbose=None*, *scrape_metrics: Optional[bool] = None*, *local=False*, *local_code_path=None*, *auto_build=None*, *param_file_secrets: Optional[Dict[str, str]] = None*) → *mlrun.model.RunObject*
Run a local or remote task.

Parameters

- **runspec** – run template object or dict (see RunTemplate)
- **handler** – pointer or name of a function handler

- **name** – execution name
- **project** – project name
- **params** – input parameters (dict)
- **inputs** – input objects (dict of key: path)
- **out_path** – default artifact output path
- **artifact_path** – default artifact output path (will replace out_path)
- **workdir** – default input artifacts path
- **watch** – watch/follow run log
- **schedule** – ScheduleCronTrigger class instance or a standard crontab expression string (which will be converted to the class using its *from_crontab* constructor), see this link for help: https://apscheduler.readthedocs.io/en/v3.6.3/modules/triggers/cron.html#module-apscheduler.triggers.cron
- **hyperparams** – dict of param name and list of values to be enumerated e.g. {"p1": [1,2,3]} the default strategy is grid search, can specify strategy (grid, list, random) and other options in the hyper_param_options parameter
- **hyper_param_options** – dict or *HyperParamOptions* struct of hyper parameter options
- **verbose** – add verbose prints/logs
- **scrape_metrics** – whether to add the *mlrun/scrape-metrics* label to this run's resources
- **local** – run the function locally vs on the runtime/cluster
- **local_code_path** – path of the code for local runs & debug
- **auto_build** – when set to True and the function require build it will be built on the first function run, use only if you dont plan on changing the build config between runs
- **param_file_secrets** – dictionary of secrets to be used only for accessing the hyperparam parameter file. These secrets are only used locally and will not be stored anywhere

Returns run context object (RunObject) with run metadata, results and status

save(*tag=''*, *versioned=False*, *refresh=False*) → str

set_db_connection(*conn*, *is_api=False*)

set_label(*key*, *value*)

property spec: mlrun.runtimes.base.FunctionSpec

property status: mlrun.runtimes.base.FunctionStatus

store_run(*runobj:* mlrun.model.RunObject)

to_dict(*fields=None*, *exclude=None*, *strip=False*)
convert the object to a python dictionary

try_auto_mount_based_on_config()

property uri

validate_and_enrich_service_account(*allowed_service_account*, *default_service_account*)

verify_base_image()

with_code(*from_file=''*, *body=None*, *with_doc=True*)
Update the function code This function eliminates the need to build container images every time we edit the code

Parameters

- **from_file** – blank for current notebook, or path to .py/.ipynb file
- **body** – will use the body as the function code
- **with_doc** – update the document of the function parameters

Returns function object

with_commands(*commands: List[str]*, *overwrite: bool = False*, *verify_base_image: bool = True*)
add commands to build spec.

Parameters **commands** – list of commands to run during build

Returns function object

with_requirements(*requirements: Union[str, List[str]]*, *overwrite: bool = False*, *verify_base_image: bool = True*)
add package requirements from file or list to build spec.

Parameters

- **requirements** – python requirements file path or list of packages
- **overwrite** – overwrite existing requirements
- **verify_base_image** – verify that the base image is configured

Returns function object

class mlrun.runtimes.**DaskCluster**(*spec=None*, *metadata=None*)
Bases: *mlrun.runtimes.kubejob.KubejobRuntime*

property client

close(*running=True*)

cluster()

deploy(*watch=True*, *with_mlrun=None*, *skip_deployed=False*, *is_kfp=False*, *mlrun_version_specifier=None*, *show_on_failure: bool = False*)
deploy function, build container with dependencies

Parameters

- **watch** – wait for the deploy to complete (and print build logs)
- **with_mlrun** – add the current mlrun package to the container build
- **skip_deployed** – skip the build if we already have an image for the function
- **mlrun_version_specifier** – which mlrun package version to include (if not current)
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}
- **show_on_failure** – show logs only in case of build failure

:return True if the function is ready (deployed)

get_status()

gpus(*gpus*, *gpu_type='nvidia.com/gpu'*)

property initialized

is_deployed()
check if the function is deployed (have a valid container)

kind = 'dask'

property spec: mlrun.runtimes.daskjob.DaskSpec

property status: mlrun.runtimes.daskjob.DaskStatus

with_limits(*mem=None*, *cpu=None*, *gpus=None*, *gpu_type='nvidia.com/gpu'*)
set pod cpu/memory/gpu limits by default it overrides the whole limits section, if you wish to patch specific resources use *patch=True*.

with_requests(*mem=None*, *cpu=None*)
set requested (desired) pod cpu/memory resources by default it overrides the whole requests section, if you wish to patch specific resources use *patch=True*.

with_scheduler_limits(*mem: Optional[str] = None*, *cpu: Optional[str] = None*, *gpus: Optional[int] = None*, *gpu_type: str = 'nvidia.com/gpu'*, *patch: bool = False*)
set scheduler pod resources limits by default it overrides the whole limits section, if you wish to patch specific resources use *patch=True*.

with_scheduler_requests(*mem: Optional[str] = None*, *cpu: Optional[str] = None*, *patch: bool = False*)
set scheduler pod resources requests by default it overrides the whole requests section, if you wish to patch specific resources use *patch=True*.

with_worker_limits(*mem: Optional[str] = None*, *cpu: Optional[str] = None*, *gpus: Optional[int] = None*, *gpu_type: str = 'nvidia.com/gpu'*, *patch: bool = False*)
set worker pod resources limits by default it overrides the whole limits section, if you wish to patch specific resources use *patch=True*.

with_worker_requests(*mem: Optional[str] = None*, *cpu: Optional[str] = None*, *patch: bool = False*)
set worker pod resources requests by default it overrides the whole requests section, if you wish to patch specific resources use *patch=True*.

class mlrun.runtimes.**HandlerRuntime**(*metadata=None*, *spec=None*)
Bases: *mlrun.runtimes.base.BaseRuntime*, mlrun.runtimes.local.ParallelRunner

kind = 'handler'

class mlrun.runtimes.**KubejobRuntime**(*spec=None*, *metadata=None*)
Bases: mlrun.runtimes.pod.KubeResource

build_config(*image=''*, *base_image=None*, *commands: Optional[list] = None*, *secret=None*, *source=None*, *extra=None*, *load_source_on_run=None*, *with_mlrun=None*, *auto_build=None*, *requirements=None*, *overwrite=False*, *verify_base_image=True*)
specify builder configuration for the deploy operation

Parameters

- **image** – target image name/path
- **base_image** – base image name/path
- **commands** – list of docker build (RUN) commands e.g. ['pip install pandas']
- **secret** – k8s secret for accessing the docker registry
- **source** – source git/tar archive to load code from in to the context/workdir e.g. git://github.com/mlrun/something.git#development

- **extra** – extra Dockerfile lines
- **load_source_on_run** – load the archive code into the container at runtime vs at build time
- **with_mlrun** – add the current mlrun package to the container build
- **auto_build** – when set to True and the function require build it will be built on the first function run, use only if you dont plan on changing the build config between runs
- **requirements** – requirements.txt file to install or list of packages to install
- **overwrite** – overwrite existing build configuration

- when False we merge the new params with the existing (currently merge is applied to requirements and commands)
- when True we replace the existing params with the new ones

Parameters **verify_base_image** – verify the base image is set

builder_status(*watch=True*, *logs=True*)

deploy(*watch=True*, *with_mlrun=None*, *skip_deployed=False*, *is_kfp=False*, *mlrun_version_specifier=None*, *builder_env: Optional[dict] = None*, *show_on_failure: bool = False*) → bool

deploy function, build container with dependencies

Parameters

- **watch** – wait for the deploy to complete (and print build logs)
- **with_mlrun** – add the current mlrun package to the container build
- **skip_deployed** – skip the build if we already have an image for the function
- **mlrun_version_specifier** – which mlrun package version to include (if not current)
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}
- **show_on_failure** – show logs only in case of build failure

:return True if the function is ready (deployed)

deploy_step(*image=None*, *base_image=None*, *commands: Optional[list] = None*, *secret_name=''*, *with_mlrun=True*, *skip_deployed=False*)

is_deployed()

check if the function is deployed (have a valid container)

kind = 'job'

with_source_archive(*source*, *workdir=None*, *handler=None*, *pull_at_runtime=True*)

load the code from git/tar/zip archive at runtime or build

Parameters

- **source** – valid path to git, zip, or tar file, e.g. git://github.com/mlrun/something.git http://some/url/file.zip
- **handler** – default function handler
- **workdir** – working dir relative to the archive root or absolute (e.g. './subdir')

- **pull_at_runtime** – load the archive into the container at job runtime vs on build/deploy

class mlrun.runtimes.**LocalRuntime**(*metadata=None*, *spec=None*)
Bases: *mlrun.runtimes.base.BaseRuntime*, mlrun.runtimes.local.ParallelRunner

is_deployed()

kind = 'local'

property spec: mlrun.runtimes.local.LocalFunctionSpec

to_job(*image=''*)

with_source_archive(*source*, *workdir=None*, *handler=None*, *target_dir=None*)
load the code from git/tar/zip archive at runtime or build

Parameters

- **source** – valid path to git, zip, or tar file, e.g. git://github.com/mlrun/something.git http://some/url/file.zip
- **handler** – default function handler
- **workdir** – working dir relative to the archive root or absolute (e.g. './subdir')
- **target_dir** – local target dir for repo clone (by default its <current-dir>/code)

class mlrun.runtimes.**RemoteRuntime**(*spec=None*, *metadata=None*)
Bases: mlrun.runtimes.pod.KubeResource

add_secrets_config_to_spec()

add_trigger(*name*, *spec*)
add a nuclio trigger object/dict

Parameters

- **name** – trigger name
- **spec** – trigger object or dict

add_v3io_stream_trigger(*stream_path*, *name='stream'*, *group='serving'*, *seek_to='earliest'*, *shards=1*, *extra_attributes=None*, *ack_window_size=None*, ***kwargs*)
add v3io stream trigger to the function

Parameters

- **stream_path** – v3io stream path (e.g. 'v3io:///projects/myproj/stream1')
- **name** – trigger name
- **group** – consumer group
- **seek_to** – start seek from: "earliest", "latest", "time", "sequence"
- **shards** – number of shards (used to set number of replicas)
- **extra_attributes** – key/value dict with extra trigger attributes
- **ack_window_size** – stream ack window size (the consumer group will be updated with the event id - ack_window_size, on failure the events in the window will be re-transmitted)
- **kwargs** – extra V3IOStreamTrigger class attributes

add_volume(*local*, *remote*, *name='fs'*, *access_key=''*, *user=''*)

deploy(*dashboard=''*, *project=''*, *tag=''*, *verbose=False*, *auth_info: Optional[mlrun.api.schemas.auth.AuthInfo] = None*, *builder_env: Optional[dict] = None*)
Deploy the nuclio function to the cluster

Parameters

- **dashboard** – address of the nuclio dashboard service (keep blank for current cluster)
- **project** – project name
- **tag** – function tag
- **verbose** – set True for verbose logging
- **auth_info** – service AuthInfo
- **builder_env** – env vars dict for source archive config/credentials e.g. builder_env={"GIT_TOKEN": token}

deploy_step(*dashboard=''*, *project=''*, *models=None*, *env=None*, *tag=None*, *verbose=None*, *use_function_from_db=None*)
return as a Kubeflow pipeline step (ContainerOp), recommended to use mlrun.deploy_function() instead

from_image(*image*)

invoke(*path: str*, *body: Optional[Union[str, bytes, dict]] = None*, *method: Optional[str] = None*, *headers: Optional[dict] = None*, *dashboard: str = ''*, *force_external_address: bool = False*, *auth_info: Optional[mlrun.api.schemas.auth.AuthInfo] = None*, *mock: Optional[bool] = None*)
Invoke the remote (live) function and return the results

example:

```
function.invoke("/api", body={"inputs": x})
```

Parameters

- **path** – request sub path (e.g. /images)
- **body** – request body (str, bytes or a dict for json requests)
- **method** – HTTP method (GET, PUT, ..)
- **headers** – key/value dict with http headers
- **dashboard** – nuclio dashboard address
- **force_external_address** – use the external ingress URL
- **auth_info** – service AuthInfo
- **mock** – use mock server vs a real Nuclio function (for local simulations)

kind = 'remote'

set_config(*key*, *value*)

property spec: mlrun.runtimes.function.NuclioSpec

property status: mlrun.runtimes.function.NuclioStatus

with_annotations(*annotations: dict*)
set a key/value annotations for function

with_http(*workers=8*, *port=0*, *host=None*, *paths=None*, *canary=None*, *secret=None*, *worker_timeout: Optional[int] = None*, *gateway_timeout: Optional[int] = None*, *trigger_name=None*, *annotations=None*, *extra_attributes=None*)

update/add nuclio HTTP trigger settings

Note: gateway timeout is the maximum request time before an error is returned, while the worker timeout if the max time a request will wait for until it will start processing, gateway_timeout must be greater than the worker_timeout.

Parameters

- **workers** – number of worker processes (default=8)
- **port** – TCP port
- **host** – hostname
- **paths** – list of sub paths
- **canary** – k8s ingress canary (% traffic value between 0 to 100)
- **secret** – k8s secret name for SSL certificate
- **worker_timeout** – worker wait timeout in sec (how long a message should wait in the worker queue before an error is returned)
- **gateway_timeout** – nginx ingress timeout in sec (request timeout, when will the gateway return an error)
- **trigger_name** – alternative nuclio trigger name
- **annotations** – key/value dict of ingress annotations
- **extra_attributes** – key/value dict of extra nuclio trigger attributes

Returns function object (self)

with_node_selection(***kwargs*)

Enables to control on which k8s node the job will run

Parameters

- **node_name** – The name of the k8s node
- **node_selector** – Label selector, only nodes with matching labels will be eligible to be picked
- **affinity** – Expands the types of constraints you can express - see https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity for details
- **tolerations** – Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints - see https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration for details

with_preemption_mode(***kwargs*)

Preemption mode controls whether pods can be scheduled on preemptible nodes. Tolerations, node selector, and affinity are populated on preemptible nodes corresponding to the function spec.

The supported modes are:

- **allow** - The function can be scheduled on preemptible nodes
- **constrain** - The function can only run on preemptible nodes
- **prevent** - The function cannot be scheduled on preemptible nodes

- **none** - No preemptible configuration will be applied on the function

The default preemption mode is configurable in mlrun.mlconf.function_defaults.preemption_mode, by default it's set to **prevent**

Parameters **mode** – allow | constrain | prevent | none defined in `PreemptionModes`

with_priority_class(***kwargs*)
Enables to control the priority of the pod If not passed - will default to mlrun.mlconf.default_function_priority_class_name

Parameters **name** – The name of the priority class

with_source_archive(*source*, *workdir=None*, *handler=None*, *runtime=''*)
Load nuclio function from remote source

Note: remote source may require credentials, those can be stored in the project secrets or passed in the function.deploy() using the builder_env dict, see the required credentials per source:

- v3io - "V3IO_ACCESS_KEY".
- git - "GIT_USERNAME", "GIT_PASSWORD".
- AWS S3 - "AWS_ACCESS_KEY_ID", "AWS_SECRET_ACCESS_KEY" or "AWS_SESSION_TOKEN".

Parameters

- **source** – a full path to the nuclio function source (code entry) to load the function from
- **handler** – a path to the function's handler, including path inside archive/git repo
- **workdir** – working dir relative to the archive root (e.g. 'subdir')
- **runtime** – (optional) the runtime of the function (defaults to python:3.7)

Examples git:

```
fn.with_source_archive("git://github.com/org/repo#my-branch",
        handler="main:handler",
        workdir="path/inside/repo")
```

s3:

```
fn.spec.nuclio_runtime = "golang"
fn.with_source_archive("s3://my-bucket/path/in/bucket/my-functions-
↪archive",
    handler="my_func:Handler",
    workdir="path/inside/functions/archive",
    runtime="golang")
```

with_v3io(*local=''*, *remote=''*)
Add v3io volume to the function

Parameters

- **local** – local path (mount path inside the function container)
- **remote** – v3io path

class mlrun.runtimes.**RemoteSparkRuntime**(*spec=None*, *metadata=None*)
Bases: *mlrun.runtimes.kubejob.KubejobRuntime*

default_image = '.remote-spark-default-image'

deploy(*watch=True*, *with_mlrun=None*, *skip_deployed=False*, *is_kfp=False*, *mlrun_version_specifier=None*, *show_on_failure: bool = False*)
deploy function, build container with dependencies

Parameters

- **watch** – wait for the deploy to complete (and print build logs)
- **with_mlrun** – add the current mlrun package to the container build
- **skip_deployed** – skip the build if we already have an image for the function
- **mlrun_version_specifier** – which mlrun package version to include (if not current)
- **builder_env** – Kaniko builder pod env vars dict (for config/credentials) e.g. builder_env={"GIT_TOKEN": token}
- **show_on_failure** – show logs only in case of build failure

:return True if the function is ready (deployed)

classmethod deploy_default_image()

is_deployed()
check if the function is deployed (have a valid container)

kind = 'remote-spark'

property spec: mlrun.runtimes.remotesparkjob.RemoteSparkSpec

with_security_context(*security_context: kubernetes.client.models.v1_security_context.V1SecurityContext*)
With security context is not supported for spark runtime. Driver / Executor processes run with uid / gid 1000 as long as security context is not defined. If in the future we want to support setting security context it will work only from spark version 3.2 onwards.

with_spark_service(*spark_service*, *provider='iguazio'*)
Attach spark service to function

class mlrun.runtimes.**ServingRuntime**(*spec=None*, *metadata=None*)
Bases: *mlrun.runtimes.function.RemoteRuntime*

MLRun Serving Runtime

add_child_function(*name*, *url=None*, *image=None*, *requirements=None*, *kind=None*)
in a multi-function pipeline add child function

example:

```
fn.add_child_function('enrich', './enrich.ipynb', 'mlrun/mlrun')
```

Parameters

- **name** – child function name
- **url** – function/code url, support .py, .ipynb, .yaml extensions
- **image** – base docker image for the function
- **requirements** – py package requirements file path OR list of packages
- **kind** – mlrun function/runtime kind

:return function object

add_model(*key: str*, *model_path: Optional[str] = None*, *class_name: Optional[str] = None*, *model_url: Optional[str] = None*, *handler: Optional[str] = None*, *router_step: Optional[str] = None*, *child_function: Optional[str] = None*, ***class_args*)

add ml model and/or route to the function.

Example, create a function (from the notebook), add a model class, and deploy:

```
fn = code_to_function(kind='serving')
fn.add_model('boost', model_path, model_class='MyClass', my_arg=5)
fn.deploy()
```

only works with router topology, for nested topologies (model under router under flow) need to add router to flow and use router.add_route()

Parameters

- **key** – model api key (or name:version), will determine the relative url/path
- **model_path** – path to mlrun model artifact or model directory file/object path
- **class_name** – V2 Model python class name or a model class instance (can also module.submodule.class and it will be imported automatically)
- **model_url** – url of a remote model serving endpoint (cannot be used with model_path)
- **handler** – for advanced users!, override default class handler name (do_event)
- **router_step** – router step name (to determine which router we add the model to in graphs with multiple router steps)
- **child_function** – child function name, when the model runs in a child function
- **class_args** – extra kwargs to pass to the model serving class __init__ (can be read in the model using .get_param(key) method)

add_secrets_config_to_spec()

deploy(*dashboard=''*, *project=''*, *tag=''*, *verbose=False*, *auth_info: Optional[mlrun.api.schemas.auth.AuthInfo] = None*, *builder_env: Optional[dict] = None*)

deploy model serving function to a local/remote cluster

Parameters

- **dashboard** – remote nuclio dashboard url (blank for local or auto detection)
- **project** – optional, override function specified project name
- **tag** – specify unique function tag (a different function service is created for every tag)
- **verbose** – verbose logging
- **auth_info** – The auth info to use to communicate with the Nuclio dashboard, required only when providing dashboard
- **builder_env** – env vars dict for source archive config/credentials e.g. builder_env={"GIT_TOKEN": token}

kind = 'serving'

plot(*filename=None*, *format=None*, *source=None*, ***kw*)

plot/save graph using graphviz

example:

```
serving_fn = mlrun.new_function("serving", image="mlrun/mlrun", kind="serving")
serving_fn.add_model('my-classifier',model_path=model_path,
                     class_name='mlrun.frameworks.sklearn.SklearnModelServer')
serving_fn.plot(rankdir="LR")
```

Parameters

- **filename** – target filepath for the image (None for the notebook)
- **format** – The output format used for rendering (`'pdf'`, `'png'`, etc.)
- **source** – source step to add to the graph
- **kw** – kwargs passed to graphviz, e.g. rankdir="LR" (see: https://graphviz.org/doc/info/attrs.html)

Returns graphviz graph object

remove_states(*keys: list*)

remove one, multiple, or all states/models from the spec (blank list for all)

set_topology(*topology=None*, *class_name=None*, *engine=None*, *exist_ok=False*, ***class_args*) → Union[mlrun.serving.states.RootFlowStep, *mlrun.serving.states.RouterStep*]

set the serving graph topology (router/flow) and root class or params

examples:

```
# simple model router topology
graph = fn.set_topology("router")
fn.add_model(name, class_name="ClassifierModel", model_path=model_uri)

# async flow topology
graph = fn.set_topology("flow", engine="async")
graph.to("MyClass").to(name="to_json", handler="json.dumps").respond()
```

topology options are:

```
router - root router + multiple child route states/models
         route is usually determined by the path (route key/name)
         can specify special router class and router arguments

flow   - workflow (DAG) with a chain of states
         flow support "sync" and "async" engines, branches are not allowed in␣
↪sync mode
         when using async mode calling state.respond() will mark the state as␣
↪the
         one which generates the (REST) call response
```

Parameters

- **topology** –
  - graph topology, router or flow
- **class_name** –
  - optional for router, router class name/path or router object
- **engine** –

- optional for flow, sync or async engine (default to async)

- **exist_ok** –
  - allow overriding existing topology
- **class_args** –
  - optional, router/flow class init args

:return graph object (fn.spec.graph)

set_tracking(*stream_path: Optional[str] = None*, *batch: Optional[int] = None*, *sample: Optional[int] = None*, *stream_args: Optional[dict] = None*, *tracking_policy: Optional[Union[mlrun.utils.model_monitoring.TrackingPolicy, dict]] = None*)

set tracking parameters:

Parameters

- **stream_path** – Path/url of the tracking stream e.g. v3io:///users/mike/mystream you can use the "dummy://" path for test/simulation.
- **batch** – Micro batch size (send micro batches of N records at a time).
- **sample** – Sample size (send only one of N records).
- **stream_args** – Stream initialization parameters, e.g. shards, retention_in_hours, ..
- **tracking_policy** – Tracking policy object or a dictionary that will be converted into a tracking policy object. By using TrackingPolicy, the user can apply his model monitoring requirements, such as setting the scheduling policy of the model monitoring batch job or changing the image of the model monitoring stream.

  example:

```
# initialize a new serving function
serving_fn = mlrun.import_function("hub://v2_model_server", new_
↪name="serving")
# apply model monitoring and set monitoring batch job to run␣
↪every 3 hours
tracking_policy = {'default_batch_intervals':"0 */3 * * *"}
serving_fn.set_tracking(tracking_policy=tracking_policy)
```

property spec: mlrun.runtimes.serving.ServingSpec

to_mock_server(*namespace=None*, *current_function='*'*, *track_models=False*, *workdir=None*, ***kwargs*) → *mlrun.serving.server.GraphServer*

create mock server object for local testing/emulation

Parameters

- **namespace** – one or list of namespaces/modules to search the steps classes/functions in
- **log_level** – log level (error | info | debug)
- **current_function** – specify if you want to simulate a child function, * for all functions
- **track_models** – allow model tracking (disabled by default in the mock server)
- **workdir** – working directory to locate the source code (if not the current one)

with_secrets(*kind*, *source*)

register a secrets source (file, env or dict)

read secrets from a source provider to be used in workflows, example:

```
task.with_secrets('file', 'file.txt')
task.with_secrets('inline', {'key': 'val'})
task.with_secrets('env', 'ENV1,ENV2')
task.with_secrets('vault', ['secret1', 'secret2'...])

# If using an empty secrets list [] then all accessible secrets will be␣
↪available.
task.with_secrets('vault', [])

# To use with Azure key vault, a k8s secret must be created with the following␣
↪keys:
# kubectl -n <namespace> create secret generic azure-key-vault-secret \
#     --from-literal=tenant_id=<service principal tenant ID> \
#     --from-literal=client_id=<service principal client ID> \
#     --from-literal=secret=<service principal secret key>

task.with_secrets('azure_vault', {
    'name': 'my-vault-name',
    'k8s_secret': 'azure-key-vault-secret',
    # An empty secrets list may be passed ('secrets': []) to access all vault␣
↪secrets.
    'secrets': ['secret1', 'secret2'...]
})
```

Parameters

- **kind** – secret type (file, inline, env)
- **source** – secret data or link (see example)

Returns The Runtime (function) object

17.14 mlrun.serving

class mlrun.serving.**GraphContext**(*level='info'*, *logger=None*, *server=None*, *nuclio_context=None*)

Bases: object

Graph context object

get_param(*key: str*, *default=None*)

get_remote_endpoint(*name*, *external=True*)

return the remote nuclio/serving function http(s) endpoint given its name

Parameters

- **name** – the function name/uri in the form [project/]function-name[:tag]
- **external** – return the external url (returns the external url by default)

get_secret(*key: str*)

property project
current project name (for the current function)

push_error(*event*, *message*, *source=None*, ***kwargs*)

property server

class mlrun.serving.**GraphServer**(*graph=None*, *parameters=None*, *load_mode=None*, *function_uri=None*, *verbose=False*, *version=None*, *functions=None*, *graph_initializer=None*, *error_stream=None*, *track_models=None*, *tracking_policy=None*, *secret_sources=None*, *default_content_type=None*)

Bases: mlrun.model.ModelObj

property graph: Union[mlrun.serving.states.RootFlowStep, *mlrun.serving.states.RouterStep*]

init_object(*namespace*)

init_states(*context*, *namespace*, *resource_cache: Optional[mlrun.datastore.store_resources.ResourceCache] = None*, *logger=None*, *is_mock=False*)
for internal use, initialize all steps (recursively)

kind = 'server'

run(*event*, *context=None*, *get_body=False*, *extra_args=None*)

set_current_function(*function*)
set which child function this server is currently running on

set_error_stream(*error_stream*)
set/initialize the error notification stream

test(*path: str = '/'*, *body: Optional[Union[str, bytes, dict]] = None*, *method: str = ''*, *headers: Optional[str] = None*, *content_type: Optional[str] = None*, *silent: bool = False*, *get_body: bool = True*, *event_id: Optional[str] = None*, *trigger: Optional[mlrun.serving.server.MockTrigger] = None*, *offset=None*, *time=None*)
invoke a test event into the server to simulate/test server behavior

example:

```
server = create_graph_server()
server.add_model("my", class_name=MyModelClass, model_path="{path}", z=100)
print(server.test("my/infer", testdata))
```

Parameters

- **path** – api path, e.g. (/{router.url_prefix}/{model-name}/..) path
- **body** – message body (dict or json str/bytes)
- **method** – optional, GET, POST, ..
- **headers** – optional, request headers, ..
- **content_type** – optional, http mime type
- **silent** – don't raise on error responses (when not 20X)
- **get_body** – return the body as py object (vs serialize response into json)
- **event_id** – specify the unique event ID (by default a random value will be generated)

- **trigger** – nuclio trigger info or mlrun.serving.server.MockTrigger class (holds kind and name)
- **offset** – trigger offset (for streams)
- **time** – event time Datetime or str, default to now()

wait_for_completion()
wait for async operation to complete

class mlrun.serving.**QueueStep**(*name: Optional[str] = None*, *path: Optional[str] = None*, *after: Optional[list] = None*, *shards: Optional[int] = None*, *retention_in_hours: Optional[int] = None*, *trigger_args: Optional[dict] = None*, ***options*)
Bases: mlrun.serving.states.BaseStep

queue step, implement an async queue or represent a stream

property async_object

default_shape = 'cds'

init_object(*context*, *namespace*, *mode='sync'*, *reset=False*, ***extra_kwargs*)
init the step class

kind = 'queue'

run(*event*, **args*, ***kwargs*)

class mlrun.serving.**RouterStep**(*class_name: Optional[Union[str, type]] = None*, *class_args: Optional[dict] = None*, *handler: Optional[str] = None*, *routes: Optional[list] = None*, *name: Optional[str] = None*, *function: Optional[str] = None*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*)
Bases: *mlrun.serving.states.TaskStep*

router step, implement routing logic for running child routes

add_route(*key*, *route=None*, *class_name=None*, *handler=None*, *function=None*, ***class_args*)
add child route step or class to the router

Parameters

- **key** – unique name (and route path) for the child step
- **route** – child step object (Task, ..)
- **class_name** – class name to build the route step from (when route is not provided)
- **class_args** – class init arguments
- **handler** – class handler to invoke on run/event
- **function** – function this step should run in

clear_children(*routes: list*)
clear child steps (routes)

default_shape = 'doubleoctagon'

get_children()
get child steps (routes)

init_object(*context*, *namespace*, *mode='sync'*, *reset=False*, ***extra_kwargs*)
init the step class

kind = 'router'

plot(*filename=None*, *format=None*, *source=None*, ***kw*)
plot/save graph using graphviz

Parameters

- **filename** – target filepath for the image (None for the notebook)
- **format** – The output format used for rendering (`'pdf'`, `'png'`, etc.)
- **source** – source step to add to the graph
- **kw** – kwargs passed to graphviz, e.g. rankdir="LR" (see: https://graphviz.org/doc/info/attrs.html)

Returns graphviz graph object

property routes
child routes/steps, traffic is routed to routes based on router logic

class mlrun.serving.**TaskStep**(*class_name: Optional[Union[str, type]] = None*, *class_args: Optional[dict] = None*, *handler: Optional[str] = None*, *name: Optional[str] = None*, *after: Optional[list] = None*, *full_event: Optional[bool] = None*, *function: Optional[str] = None*, *responder: Optional[bool] = None*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*)
Bases: mlrun.serving.states.BaseStep

task execution step, runs a class or handler

property async_object
return the sync or async (storey) class instance

clear_object()

init_object(*context*, *namespace*, *mode='sync'*, *reset=False*, ***extra_kwargs*)
init the step class

kind = 'task'

respond()
mark this step as the responder.

step output will be returned as the flow result, no other step can follow

run(*event*, **args*, ***kwargs*)
run this step, in async flows the run is done through storey

class mlrun.serving.**V2ModelServer**(*context=None*, *name: Optional[str] = None*, *model_path: Optional[str] = None*, *model=None*, *protocol=None*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*, ***kwargs*)
Bases: mlrun.serving.utils.StepToDict

base model serving class (v2), using similar API to KFServing v2 and Triton

base model serving class (v2), using similar API to KFServing v2 and Triton

The class is initialized automatically by the model server and can run locally as part of a nuclio serverless function, or as part of a real-time pipeline default model url is: /v2/models/<model>[/versions/<ver>]/operation

You need to implement two mandatory methods: load() - download the model file(s) and load the model into memory predict() - accept request payload and return prediction/inference results

you can override additional methods : preprocess, validate, postprocess, explain you can add custom api endpoint by adding method op_xx(event), will be invoked by calling the <model-url>/xx (operation = xx)

model server classes are subclassed (subclass implements the *load()* and *predict()* methods) the subclass can be added to a serving graph or to a model router

defining a sub class:

```
class MyClass(V2ModelServer):
    def load(self):
        # load and initialize the model and/or other elements
        model_file, extra_data = self.get_model(suffix='.pkl')
        self.model = load(open(model_file, "rb"))

    def predict(self, request):
        events = np.array(request['inputs'])
        dmatrix = xgb.DMatrix(events)
        result: xgb.DMatrix = self.model.predict(dmatrix)
        return {"outputs": result.tolist()}
```

usage example:

```
# adding a model to a serving graph using the subclass MyClass
# MyClass will be initialized with the name "my", the model_path, and an arg called␣
↪my_param
graph = fn.set_topology("router")
fn.add_model("my", class_name="MyClass", model_path="<model-uri>>", my_param=5)
```

Parameters

- **context** – for internal use (passed in init)
- **name** – step name
- **model_path** – model file/dir or artifact path
- **model** – model object (for local testing)
- **protocol** – serving API protocol (default "v2")
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {"data": {"a": 5, "b": 7}}, input_path="data.b" means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {"x": 5} , result_path="resp" means the returned response will be written to event["y"] resulting in {"x": 5, "resp": <result>}
- **kwargs** – extra arguments (can be accessed using self.get_param(key))

do_event(*event*, **args*, ***kwargs*)
main model event handler method

explain(*request: Dict*) → Dict
model explain operation

get_model(*suffix=''*)
get the model file(s) and metadata from model store

the method returns a path to the model file and the extra data (dict of dataitem objects) it also loads the model metadata into the self.model_spec attribute, allowing direct access to all the model metadata attributes.

get_model is usually used in the model .load() method to init the model .. rubric:: Examples

```
def load(self):
    model_file, extra_data = self.get_model(suffix='.pkl')
    self.model = load(open(model_file, "rb"))
    categories = extra_data['categories'].as_df()
```

Parameters **suffix** (*str*) – optional, model file suffix (when the model_path is a directory)

Returns

- *str* – (local) model file
- *dict* – extra dataitems dictionary

get_param(*key: str*, *default=None*)
get param by key (specified in the model or the function)

load()
model loading function, see also .get_model() method

logged_results(*request: dict*, *response: dict*, *op: str*)
hook for controlling which results are tracked by the model monitoring

this hook allows controlling which input/output data is logged by the model monitoring allow filtering out columns or adding custom values, can also be used to monitor derived metrics for example in image classification calculate and track the RGB values vs the image bitmap

the request["inputs"] holds a list of input values/arrays, the response["outputs"] holds a list of corresponding output values/arrays (the schema of the input/output fields is stored in the model object), this method should return lists of alternative inputs and outputs which will be monitored

Parameters

- **request** – predict/explain request, see model serving docs for details
- **response** – result from the model predict/explain (after postprocess())
- **op** – operation (predict/infer or explain)

Returns the input and output lists to track

post_init(*mode='sync'*)
sync/async model loading, for internal use

postprocess(*request: Dict*) → Dict
postprocess, before returning response

predict(*request: Dict*) → Dict
model prediction operation

preprocess(*request: Dict*, *operation*) → Dict
preprocess the event body before validate and action

set_metric(*name: str*, *value*)
set real time metric (for model monitoring)

validate(*request*, *operation*)
validate the event body (after preprocess)

class mlrun.serving.**VotingEnsemble**(*context=None*, *name: Optional[str] = None*, *routes=None*, *protocol: Optional[str] = None*, *url_prefix: Optional[str] = None*, *health_prefix: Optional[str] = None*, *vote_type=None*, *executor_type=None*, *prediction_col_name=None*, ***kwargs*)

Bases: mlrun.serving.routers.BaseModelRouter

Voting Ensemble

The *VotingEnsemble* class enables you to apply prediction logic on top of the different added models.

You can use it by calling:

- ***<prefix>/<model>[/versions/<ver>]/operation*** Sends the event to the specific <model>[/versions/<ver>]
- ***<prefix>/operation*** Sends the event to all models and applies *vote(self, event)*

The *VotingEnsemble* applies the following logic: Incoming Event -> Router Preprocessing -> Send to model/s -> Apply all model/s logic (Preprocessing -> Prediction -> Postprocessing) -> Router Voting logic -> Router Postprocessing -> Response

This enables you to do the general preprocessing and postprocessing steps once on the router level, with only model-specific adjustments at the model level.

> When enabling model tracking via *set_tracking()* the ensemble logic predictions will appear with model name as the given VotingEnsemble name or "VotingEnsemble" by default.

Example:

```
# Define a serving function
# Note: You can point the function to a file containing you own Router or␣
↪Classifier Model class
#       this basic class supports sklearn based models (with `<model>.predict()` api)
fn = mlrun.code_to_function(name='ensemble',
                            kind='serving',
                            filename='model-server.py'
                            image='mlrun/ml-models')

# Set the router class
# You can set your own classes by simply changing the `class_name`
fn.set_topology(class_name='mlrun.serving.routers.VotingEnsemble')

# Add models
fn.add_model(<model_name>, <model_path>, <model_class_name>)
fn.add_model(<model_name>, <model_path>, <model_class_name>)
```

How to extend the VotingEnsemble:

The VotingEnsemble applies its logic using the *logic(predictions)* function. The *logic()* function receives an array of (# samples, # predictors) which you can then use to apply whatever logic you may need.

If we use this *VotingEnsemble* as an example, the *logic()* function tries to figure out whether you are trying to do a **classification** or a **regression** prediction by the prediction type or by the given *vote_type* parameter. Then we apply the appropriate *max_vote()* or *mean_vote()* which calculates the actual prediction result and returns it as the VotingEnsemble's prediction.

Parameters

- **context** – for internal use (passed in init)
- **name** – step name
- **routes** – for internal use (routes passed in init)

- **protocol** – serving API protocol (default "v2")
- **url_prefix** – url prefix for the router (default /v2/models)
- **health_prefix** – health api url prefix (default /v2/health)
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {"data": {"a": 5, "b": 7}}, input_path="data.b" means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {"x": 5} , result_path="resp" means the returned response will be written to event["y"] resulting in {"x": 5, "resp": <result>}
- **vote_type** – Voting type to be used (from *VotingTypes*). by default will try to self-deduct upon the first event: - float prediction type: regression - int prediction type: classification
- **executor_type** – Parallelism mechanism, out of *ParallelRunnerModes*, by default *threads*
- **prediction_col_name** – The dict key for the predictions column in the model's responses output. Example: If the model returns *{id: <id>, model_name: <name>, outputs: {..., prediction: [<predictions>], ...}}* the prediction_col_name should be *prediction*. by default, *prediction*
- **kwargs** – extra arguments

do_event(*event*, **args*, ***kwargs*)

Handles incoming requests.

Parameters **event** (*nuclio.Event*) – Incoming request as a nuclio.Event.

Returns Event response after running the requested logic

Return type Response

extract_results_from_response(*response*)

Extracts the prediction from the model response. This function is used to allow multiple model return types. and allow for easy extension to the user's ensemble and models best practices.

Parameters **response** (*Union[List, Dict]*) – The model response's *output* field.

Returns The model's predictions

Return type List

logic(*predictions*)

post_init(*mode='sync'*)

validate(*request: dict*, *method: str*)

Validate the event body (after preprocessing)

Parameters

- **request** – Event body.
- **method** – Event method.

Returns The given Event body (request).

Raises **Exception** – If validation failed.

mlrun.serving.**create_graph_server**(*parameters={}*, *load_mode=None*, *graph=None*, *verbose=False*, *current_function=None*, ***kwargs*) → *mlrun.serving.server.GraphServer*

create graph server host/emulator for local or test runs

Usage example:

```
server = create_graph_server(graph=RouterStep(), parameters={})
server.init(None, globals())
server.graph.add_route("my", class_name=MyModelClass, model_path="{path}", z=100)
print(server.test("/v2/models/my/infer", testdata))
```

class mlrun.serving.remote.**BatchHttpRequests**(*url: Optional[str] = None*, *subpath: Optional[str] = None*, *method: Optional[str] = None*, *headers: Optional[dict] = None*, *url_expression: Optional[str] = None*, *body_expression: Optional[str] = None*, *return_json: bool = True*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*, *retries=None*, *backoff_factor=None*, *timeout=None*, ***kwargs*)

class for calling remote endpoints in parallel

class for calling remote endpoints in parallel

sync and async graph step implementation for request/resp to remote service (class shortcut = "$remote") url can be an http(s) url (e.g. "https://myservice/path") or an mlrun function uri ([project/]name). alternatively the url_expression can be specified to build the url from the event (e.g. "event['url']").

example pipeline:

```
function = mlrun.new_function("myfunc", kind="serving")
flow = function.set_topology("flow", engine="async")
flow.to(
    BatchHttpRequests(
        url_expression="event['url']",
        body_expression="event['data']",
        method="POST",
        input_path="req",
        result_path="resp",
    )
).respond()

server = function.to_mock_server()
# request contains a list of elements, each with url and data
request = [{"url": f"{base_url}/{i}", "data": i} for i in range(2)]
resp = server.test(body={"req": request})
```

Parameters

- **url** – http(s) url or function [project/]name to call
- **subpath** – path (which follows the url)
- **method** – HTTP method (GET, POST, ..), default to POST
- **headers** – dictionary with http header values
- **url_expression** – an expression for getting the url from the event, e.g. "event['url']"

- **body_expression** – an expression for getting the request body from the event, e.g. “event[‘data’]”
- **return_json** – indicate the returned value is json, and convert it to a py object
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {“data”: {“a”: 5, “b”: 7}}, input_path=”data.b” means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {“x”: 5} , result_path=”resp” means the returned response will be written to event[“y”] resulting in {“x”: 5, “resp”: <result>}
- **retries** – number of retries (in exponential backoff)
- **backoff_factor** – A backoff factor in seconds to apply between attempts after the second try
- **timeout** – How long to wait for the server to send data before giving up, float in seconds

__init__(*url: Optional[str] = None*, *subpath: Optional[str] = None*, *method: Optional[str] = None*, *headers: Optional[dict] = None*, *url_expression: Optional[str] = None*, *body_expression: Optional[str] = None*, *return_json: bool = True*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*, *retries=None*, *backoff_factor=None*, *timeout=None*, ***kwargs*)

class for calling remote endpoints in parallel

sync and async graph step implementation for request/resp to remote service (class shortcut = “$remote”) url can be an http(s) url (e.g. “https://myservice/path”) or an mlrun function uri ([project/]name). alternatively the url_expression can be specified to build the url from the event (e.g. “event[‘url’]”).

example pipeline:

```
function = mlrun.new_function("myfunc", kind="serving")
flow = function.set_topology("flow", engine="async")
flow.to(
    BatchHttpRequests(
        url_expression="event['url']",
        body_expression="event['data']",
        method="POST",
        input_path="req",
        result_path="resp",
    )
).respond()

server = function.to_mock_server()
# request contains a list of elements, each with url and data
request = [{"url": f"{base_url}/{i}", "data": i} for i in range(2)]
resp = server.test(body={"req": request})
```

Parameters

- **url** – http(s) url or function [project/]name to call
- **subpath** – path (which follows the url)
- **method** – HTTP method (GET, POST, ..), default to POST
- **headers** – dictionary with http header values
- **url_expression** – an expression for getting the url from the event, e.g. “event[‘url’]”

- **body_expression** – an expression for getting the request body from the event, e.g. “event[‘data’]”
- **return_json** – indicate the returned value is json, and convert it to a py object
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {“data”: {“a”: 5, “b”: 7}}, input_path=”data.b” means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {“x”: 5} , result_path=”resp” means the returned response will be written to event[“y”] resulting in {“x”: 5, “resp”: <result>}
- **retries** – number of retries (in exponential backoff)
- **backoff_factor** – A backoff factor in seconds to apply between attempts after the second try
- **timeout** – How long to wait for the server to send data before giving up, float in seconds

class mlrun.serving.remote.**RemoteStep**(*url: str*, *subpath: Optional[str] = None*, *method: Optional[str] = None*, *headers: Optional[dict] = None*, *url_expression: Optional[str] = None*, *body_expression: Optional[str] = None*, *return_json: bool = True*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*, *max_in_flight=None*, *retries=None*, *backoff_factor=None*, *timeout=None*, ***kwargs*)

class for calling remote endpoints

class for calling remote endpoints

sync and async graph step implementation for request/resp to remote service (class shortcut = “$remote”) url can be an http(s) url (e.g. “https://myservice/path”) or an mlrun function uri ([project/]name). alternatively the url_expression can be specified to build the url from the event (e.g. “event[‘url’]”).

example pipeline:

```
flow = function.set_topology("flow", engine="async")
flow.to(name="step1", handler="func1")
    .to(RemoteStep(name="remote_echo", url="https://myservice/path", method="POST"))
    .to(name="laststep", handler="func2").respond()
```

Parameters

- **url** – http(s) url or function [project/]name to call
- **subpath** – path (which follows the url), use *$path* to use the event.path
- **method** – HTTP method (GET, POST, ..), default to POST
- **headers** – dictionary with http header values
- **url_expression** – an expression for getting the url from the event, e.g. “event[‘url’]”
- **body_expression** – an expression for getting the request body from the event, e.g. “event[‘data’]”
- **return_json** – indicate the returned value is json, and convert it to a py object

- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {"data": {"a": 5, "b": 7}}, input_path="data.b" means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {"x": 5} , result_path="resp" means the returned response will be written to event["y"] resulting in {"x": 5, "resp": <result>}
- **retries** – number of retries (in exponential backoff)
- **backoff_factor** – A backoff factor in seconds to apply between attempts after the second try
- **timeout** – How long to wait for the server to send data before giving up, float in seconds

__init__(*url: str*, *subpath: Optional[str] = None*, *method: Optional[str] = None*, *headers: Optional[dict] = None*, *url_expression: Optional[str] = None*, *body_expression: Optional[str] = None*, *return_json: bool = True*, *input_path: Optional[str] = None*, *result_path: Optional[str] = None*, *max_in_flight=None*, *retries=None*, *backoff_factor=None*, *timeout=None*, ***kwargs*)

class for calling remote endpoints

sync and async graph step implementation for request/resp to remote service (class shortcut = "$remote") url can be an http(s) url (e.g. "https://myservice/path") or an mlrun function uri ([project/]name). alternatively the url_expression can be specified to build the url from the event (e.g. "event['url']").

example pipeline:

```
flow = function.set_topology("flow", engine="async")
flow.to(name="step1", handler="func1")
    .to(RemoteStep(name="remote_echo", url="https://myservice/path", method=
↪"POST"))
    .to(name="laststep", handler="func2").respond()
```

Parameters

- **url** – http(s) url or function [project/]name to call
- **subpath** – path (which follows the url), use *$path* to use the event.path
- **method** – HTTP method (GET, POST, ..), default to POST
- **headers** – dictionary with http header values
- **url_expression** – an expression for getting the url from the event, e.g. "event['url']"
- **body_expression** – an expression for getting the request body from the event, e.g. "event['data']"
- **return_json** – indicate the returned value is json, and convert it to a py object
- **input_path** – when specified selects the key/path in the event to use as body this require that the event body will behave like a dict, example: event: {"data": {"a": 5, "b": 7}}, input_path="data.b" means request body will be 7
- **result_path** – selects the key/path in the event to write the results to this require that the event body will behave like a dict, example: event: {"x": 5} , result_path="resp" means the returned response will be written to event["y"] resulting in {"x": 5, "resp": <result>}
- **retries** – number of retries (in exponential backoff)

- **backoff_factor** – A backoff factor in seconds to apply between attempts after the second try
- **timeout** – How long to wait for the server to send data before giving up, float in seconds

17.15 storey.transformations - Graph transformations

Graph transformations are contained in the *storey.transformations* module. For convenience, they can also be imported directly from the storey package. Note that the transformation functions are actually encapsulated in classes, so that they can be referenced by name of class from graph step definitions.

class storey.transformations.**AggregateByKey**(*aggregates: Union[List[storey.dtypes.FieldAggregator], List[Dict[str, object]]], table: Union[storey.table.Table, str], key: Optional[Union[str, Callable[[storey.dtypes.Event], object]]] = None, emit_policy: Union[storey.dtypes.EmitPolicy, Dict[str, object]] = <storey.dtypes.EmitEveryEvent object>, augmentation_fn: Optional[Callable[[storey.dtypes.Event, Dict[str, object]], storey.dtypes.Event]] = None, enrich_with: Optional[List[str]] = None, aliases: Optional[Dict[str, str]] = None, use_windows_from_schema: bool = False, **kwargs*)

Aggregates the data into the table object provided for later persistence, and outputs an event enriched with the requested aggregation features.

Persistence is done via the *NoSqlTarget* step and based on the Cache object persistence settings.

Parameters

- **aggregates** – List of aggregates to apply for each event. accepts either list of FieldAggregators or a dictionary describing FieldAggregators.
- **table** – A Table object or name for persistence of aggregations. If a table name is provided, it will be looked up in the context object passed in kwargs.
- **key** – Key field to aggregate by, accepts either a string representing the key field or a key extracting function. Defaults to the key in the event's metadata. (Optional)
- **emit_policy** – Policy indicating when the data will be emitted. Defaults to EmitEveryEvent
- **augmentation_fn** – Function that augments the features into the event's body. Defaults to updating a dict. (Optional)
- **enrich_with** – List of attributes names from the associated storage object to be fetched and added to every event. (Optional)
- **aliases** – Dictionary specifying aliases for enriched or aggregate columns, of the format *{'col_name': 'new_col_name'}*. (Optional)

class storey.transformations.**Assert**(***kwargs*)

Exposes an API for testing the flow between steps.

class storey.transformations.**Batch**(*max_events: Optional[int] = None*, *flush_after_seconds: Optional[int] = None*, *key: Optional[Union[str, Callable[[storey.dtypes.Event], str]]] = None*, ***kwargs*)

Batches events into lists of up to max_events events. Each emitted list contained max_events events, unless flush_after_seconds seconds have passed since the first event in the batch was received, at which the batch is emitted with potentially fewer than max_events event.

Parameters

- **max_events** – Maximum number of events per emitted batch. Set to None to emit all events in one batch on flow termination.
- **flush_after_seconds** – Maximum number of seconds to wait before a batch is emitted.
- **key** – The key by which events are grouped. By default (None), events are not grouped. Other options may be: Set a '$key' to group events by the Event.key property. set a 'str' key to group events by Event.body[str]. set a Callable[Any, Any] to group events by a a custom key extractor.

class storey.transformations.**Choice**(*choice_array*, *default=None*, ***kwargs*)

Redirects each input element into at most one of multiple downstreams.

Parameters

- **choice_array** (*tuple of (Flow, Function (Event=>boolean))*) – a list of (downstream, condition) tuples, where downstream is a step and condition is a function. The first condition in the list to evaluate as true for an input element causes that element to be redirected to that downstream step.
- **default** (*Flow*) – a default step for events that did not match any condition in choice_array. If not set, elements that don't match any condition will be discarded.
- **name** (*string*) – Name of this step, as it should appear in logs. Defaults to class name (Choice).
- **full_event** (*boolean*) – Whether user functions should receive and/or return Event objects (when True), or only the payload (when False). Defaults to False.

class storey.transformations.**Extend**(*fn*, *long_running=None*, ***kwargs*)

Adds fields to each incoming event.

Parameters

- **fn** (*Function (Event=>Dict)*) – Function to transform each event to a dictionary. The fields in the returned dictionary are then added to the original event.
- **long_running** (*boolean*) – Whether fn is a long-running function. Long-running functions are run in an executor to avoid blocking other concurrent processing. Default is False.
- **name** (*string*) – Name of this step, as it should appear in logs. Defaults to class name (Extend).
- **full_event** (*boolean*) – Whether user functions should receive and/or return Event objects (when True), or only the payload (when False). Defaults to False.

class storey.transformations.**Filter**(*fn*, *long_running=None*, ***kwargs*)

Filters events based on a user-provided function.

Parameters

- **fn** (*Function (Event=>boolean)*) – Function to decide whether to keep each event.

- **long_running** (*boolean*) – Whether fn is a long-running function. Long-running functions are run in an executor to avoid blocking other concurrent processing. Default is False.
- **name** (*string*) – Name of this step, as it should appear in logs. Defaults to class name (Filter).
- **full_event** (*boolean*) – Whether user functions should receive and/or return Event objects (when True), or only the payload (when False). Defaults to False.

class storey.transformations.**FlatMap**(*fn*, *long_running=None*, ***kwargs*)

Maps, or transforms, each incoming event into any number of events.

Parameters

- **fn** (*Function (Event=>list of Event)*) – Function to transform each event to a list of events.
- **long_running** (*boolean*) – Whether fn is a long-running function. Long-running functions are run in an executor to avoid blocking other concurrent processing. Default is False.
- **name** (*string*) – Name of this step, as it should appear in logs. Defaults to class name (FlatMap).
- **full_event** (*boolean*) – Whether user functions should receive and/or return Event objects (when True), or only the payload (when False). Defaults to False.

storey.transformations.**Flatten**(***kwargs*)

Flatten is equivalent to FlatMap(lambda x: x).

class storey.transformations.**ForEach**(*fn*, *long_running=None*, ***kwargs*)

Applies given function on each event in the stream, passes original event downstream.

class storey.transformations.**JoinWithTable**(*table: Union[storey.table.Table, str]*, *key_extractor: Union[str, Callable[[storey.dtypes.Event], str]]*, *attributes: Optional[List[str]] = None*, *inner_join: bool = False*, *join_function: Optional[Callable[[Any, Dict[str, object]], Any]] = None*, ***kwargs*)

Joins each event with data from the given table.

Parameters

- **table** – A Table object or name to join with. If a table name is provided, it will be looked up in the context.
- **key_extractor** – Key's column name or a function for extracting the key, for table access from an event.
- **attributes** – A comma-separated list of attributes to be queried for. Defaults to all attributes.
- **inner_join** – Whether to drop events when the table does not have a matching entry (join_function won't be called in such a case). Defaults to False.
- **join_function** – Joins the original event with relevant data received from the storage. Event is dropped when this function returns None. Defaults to assume the event's body is a dict-like object and updating it.
- **name** – Name of this step, as it should appear in logs. Defaults to class name (JoinWithTable).

- **full_event** – Whether user functions should receive and/or return Event objects (when True), or only the payload (when False). Defaults to False.
- **context** – Context object that holds global configurations and secrets.

class storey.transformations.**Map**(*fn*, *long_running=None*, ***kwargs*)

Maps, or transforms, incoming events using a user-provided function.

Parameters

- **fn** (*Function (Event=>Event)*) – Function to apply to each event
- **long_running** (*boolean*) – Whether fn is a long-running function. Long-running functions are run in an executor to avoid blocking other concurrent processing. Default is False.
- **name** (*string*) – Name of this step, as it should appear in logs. Defaults to class name (Map).
- **full_event** (*boolean*) – Whether user functions should receive and/or return Event objects (when True), or only the payload (when False). Defaults to False.

class storey.transformations.**MapClass**(*long_running=None*, ***kwargs*)

Similar to Map, but instead of a function argument, this class should be extended and its do() method overridden.

class storey.transformations.**MapWithState**(*initial_state*, *fn*, *group_by_key=False*, ***kwargs*)

Maps, or transforms, incoming events using a stateful user-provided function, and an initial state, which may be a database table.

Parameters

- **initial_state** (*dictionary or Table if group_by_key is True. Any object otherwise.*) – Initial state for the computation. If group_by_key is True, this must be a dictionary or a Table object.
- **fn** (*Function ((Event, state)=>(Event, state))*) – A function to run on each event and the current state. Must yield an event and an updated state.
- **group_by_key** (*boolean*) – Whether the state is computed by key. Optional. Default to False.
- **full_event** (*boolean*) – Whether fn will receive and return an Event object or only the body (payload). Optional. Defaults to False (body only).

class storey.transformations.**Partition**(*predicate: Callable[[Any], bool]*, ***kwargs*)

Partitions events by calling a predicate function on each event. Each processed event results in a *Partitioned* namedtuple of (left=Optional[Event], right=Optional[Event]).

For a given event, if the predicate function results in *True*, the event is assigned to *left*. Otherwise, the event is assigned to *right*.

Parameters **predicate** – A predicate function that results in a boolean.

class storey.transformations.**ReifyMetadata**(*mapping: Iterable[str]*, ***kwargs*)

Inserts event metadata into the event body. :param mapping: Dictionary from event attribute name to entry key in the event body (which must be a

dictionary). Alternatively, an iterable of names may be provided, and these will be used as both attribute name and entry key.

Parameters **name** (*string*) – Name of this step, as it should appear in logs. Defaults to class name (ReifyMetadata).

class storey.transformations.**SampleWindow**(*window_size: int*, *emit_period: storey.steps.sample.EmitPeriod = EmitPeriod.FIRST*, *emit_before_termination: bool = False*, *key: Optional[Union[str, Callable[[storey.dtypes.Event], str]]] = None*, ***kwargs*)

Emits a single event in a window of *window_size* events, in accordance with *emit_period* and *emit_before_termination*.

Parameters

- **window_size** – The size of the window we want to sample a single event from.
- **emit_period** – What event should this step emit for each *window_size* (default: EmitPeriod.First).

Available options: 1.1) EmitPeriod.FIRST - will emit the first event in a window *window_size* events. 1.2) EmitPeriod.LAST - will emit the last event in a window of *window_size* events.

Parameters **emit_before_termination** – On termination signal, should the step emit the last event it seen (default: False).

Available options: 2.1) True - The last event seen will be emitted downstream. 2.2) False - The last event seen will NOT be emitted downstream.

Parameters **key** – The key by which events are sampled. By default (None), events are not sampled by key. Other options may be: Set to '$key' to sample events by the Event.key property. set to 'str' key to sample events by Event.body[str]. set a Callable[[Event], str] to sample events by a custom key extractor.

class storey.transformations.**SendToHttp**(*request_builder*, *join_from_response*, ***kwargs*)

Joins each event with data from any HTTP source. Used for event augmentation.

Parameters

- **request_builder** (*Function (Event=>HttpRequest)*) – Creates an HTTP request from the event. This request is then sent to its destination.
- **join_from_response** (*Function ((Event, HttpResponse)=>Event)*) – Joins the original event with the HTTP response into a new event.
- **name** (*string*) – Name of this step, as it should appear in logs. Defaults to class name (SendToHttp).
- **full_event** (*boolean*) – Whether user functions should receive and/or return Event objects (when True), or only the payload (when False). Defaults to False.

class storey.transformations.**ToDataFrame**(*index: Optional[str] = None*, *columns: Optional[List[str]] = None*, ***kwargs*)

Create pandas data frame from events. Can appear in the middle of the flow, as opposed to ReduceToDataFrame

Parameters

- **index** – Name of the column to be used as index. Optional. If not set, DataFrame will be range indexed.
- **columns** – List of column names to be passed as-is to the DataFrame constructor. Optional.

for additional params, see documentation of `storey.flow.Flow`

See also the *index of all functions and classes*.

CHAPTER

EIGHTEEN

COMMAND-LINE INTERFACE

18.1 CLI commands

Use the following commands of the MLRun command-line interface (CLI) — `mlrun` — to build and run MLRun functions:

- *build*
- *clean*
- *config*
- *get*
- *logs*
- *project*
- *run*
- *version*
- *watch*
- *watch-stream*

Each command supports many flags, some of which are listed in their relevant sections. To view all the flags of a command, run `mlrun <command name> --help`.

18.1.1 build

Use the `build` CLI command to build all the function dependencies from the function specification into a function container (Docker image).

Usage: mlrun build [OPTIONS] FUNC_URL

Example: `mlrun build myfunc.yaml`

Flag	Description
name TEXT	Function name
project TEXT	Project name
tag TEXT	Function tag
-i, image TEXT	Target image path
-s, source TEXT	Path/URL of the function source code. A PY file, or if `-a
-b, base-image TEXT	Base Docker image
-c, command TEXT	Build commands; for example, '-c pip install pandas'
secret-name TEXT	Name of a container-registry secret
-a, archive TEXT	Path/URL of a target function-sources archive directory: as part of the build, the function sources (see `-s
silent	Do not show build logs
with-mlrun	Add the MLRun package ("mlrun")
db TEXT	Save the run results to path or DB url
-r, runtime TEXT	Function spec dict, for pipeline usage
kfp	Running inside Kubeflow Piplines, do not use
skip	Skip if already deployed

> **Note:** For information about using the `-a|--archive` option to create a function-sources archive, see *Using a Sources Archive* later in this tutorial.

18.1.2 clean

Use the `clean` CLI command to clean runtime resources. When run without any flags, it cleans the resources for all runs of all runtimes.

Usage: mlrun clean [OPTIONS] [KIND] [id]

Examples:

- Clean resources for all runs of all runtimes: `mlrun clean`
- Clean resources for all runs of a specific kind (e.g. job): `mlrun clean job`
- Clean resources for specific job (by uid): `mlrun clean mpijob 15d04c19c2194c0a8efb26ea3017254b`

Flag	Description
kind	Clean resources for all runs of a specific kind (e.g. job).
id	Delete the resources of the mlrun object twith this identifier. For most function runtimes, runtime resources are per Run, and the identifier is the Run's UID. For DASK runtime, the runtime resources are per Function, and the identifier is the Function's name.

Options	Description
api	URL of the mlrun-api service.
-ls, label-selector	Delete only runtime resources matching the label selector.
-f, force	Delete the runtime resource even if they're not in terminal state or if the grace period didn't pass.
-gp, grace-period	Grace period, in seconds, given to the runtime resource before they are actually removed, counted from the moment they moved to the terminal state.

18.1.3 config

Use the `config` CLI command to show the mlrun client environment configuration, such as location of artifacts and api.

Example: `mlrun config`

18.1.4 get

Use the `get` CLI command to list one or more objects per kind/class.

Usage: get pods | runs | artifacts | func [name]

Examples:

- `mlrun get runs --project getting-started-admin`
- `mlrun get pods --project getting-started-admin`
- `mlrun get artifacts --project getting-started-admin`
- `mlrun get func prep-data --project getting-started-admin`

Flag	Description
name	Name of object to return
-s, selector	Label selector
-n, namespace	Kubernetes namespace
uid	Object ID
project	Project name to return
-t, tag	Artifact/function tag of object to return
db	db path/url of object to return

18.1.5 logs

Use the `logs` CLI command to get or watch task logs.

Usage: logs [OPTIONS] uid

Example: `mlrun logs ba409c0cb4904d60aa8f8d1c05b40a75 --project getting-started-admin`

Flag	Description
-p, project TEXT	Project name
offset INTEGER	Retrieve partial log, get up to size bytes starting at the offset from beginning of log
db TEXT	API service url
-w, watch	Retrieve logs of a running process, and watch the progress of the execution until it completes. Prints out the logs and continues to periodically poll for, and print, new logs as long as the state of the runtime that generates this log is either `pending` or `running`.

18.1.6 project

Use the `project` CLI command to load and/or run a project.

Usage: mlrun project [OPTIONS] [CONTEXT]

Example: `mlrun project -r workflow.py .`

Flag	Description
-n, name TEXT	Project name
-u, url TEXT	Remote git or archive url of the project
-r, run TEXT	Run workflow name of .py file
-a, arguments TEXT	Kubeflow pipeline arguments name and value tuples (with -r flag), e.g. -a x=6
-p, artifact_path TEXT	Target path/url for workflow artifacts. The string `{{workflow.uid}}` is replaced by workflow id
-x, param TEXT	mlrun project parameter name and value tuples, e.g. -p x=37 -p y='text'
-s, secrets TEXT	Secrets file= or env=ENV_KEY1,...
namespace TEXT	k8s namespace
db TEXT	API service url
init_git	For new projects init git the context dir
-c, clone	Force override/clone into the context dir
sync	Sync functions into db
-w, watch	Wait for pipeline completion (with -r flag)
-d, dirty	Allow run with uncommitted git changes
git_repo TEXT	git repo (org/repo) for git comments
git_issue INTEGER	git issue number for git comments
handler TEXT	Workflow function handler name
engine TEXT	Workflow engine (kfp/local)
local	Try to run workflow functions locally

18.1.7 run

Use the run CLI command to execute a task and inject parameters by using a local or remote function.

Usage: mlrun [OPTIONS] URL [ARGS]...

Examples:

- `mlrun run -f db://getting-started-admin/prep-data --project getting-started-admin`
- `mlrun run -f myfunc.yaml -w -p p1=3`

Flag	Description
-p, param TEXT	Parameter name and value tuples; for example, `-p x=37 -p y='text'`
-i, inputs TEXT	Input artifact; for example, `-i infile.txt=s3://mybucket/infile.txt`
in-path TEXT	Base directory path/URL for storing input artifacts
out-path TEXT	Base directory path/URL for storing output artifacts
-s, secrets TEXT	Secrets, either as `file=<filename>` or `env=<ENVAR>,...`; for example, `-s file=secrets.txt`
name TEXT	Run name
project TEXT	Project name or ID
-f, func-url TEXT	Path/URL of a YAML function-configuration file, or db:///[:tag] for a DB function object
task TEXT	Path/URL of a YAML task-configuration file
handler TEXT	Invoke the function handler inside the code file

18.1.8 version

Use the version CLI command to get the mlrun server version.

18.1.9 The watch Command

Use the watch CLI command to read the current or previous task (pod) logs.

Usage: mlrun watch [OPTIONS] POD

Example: `mlrun watch prep-data-6rf7b`

Flag	Description
-n, namespace	kubernetes namespace
-t, timeout	Timeout in seconds

18.1.10 watch-stream

Use the `watch-stream` CLI command to watch a v3io stream and print data at a recurring interval.

Usage: mlrun watch-stream [OPTIONS] URL

Examples:

- `mlrun watch-stream v3io:///users/my-test-stream`
- `mlrun watch-stream v3io:///users/my-test-stream -s 1`
- `mlrun watch-stream v3io:///users/my-test-stream -s 1 -s 2`
- `mlrun watch-stream v3io:///users/my-test-stream -s 1 -s 2 --seek EARLIEST`

Flag	Description
-s, shard-ids	Shard id to listen on (can be multiple).
–seek TEXT	Where to start/seek (EARLIEST or LATEST)
-i, interval	Interval in seconds. Default = 3
-j, is-json	Indicates that the payload is json (will be deserialized).

18.2 Building and running a function from a Git repository

To build and run a function from a Git repository, start out by adding a YAML function-configuration file in your local environment. This file should describe the function and define its specification. For example, create a **myfunc.yaml** file with the following content in your working directory:

```
kind: job
metadata:
  name: remote-demo1
  project: ''
spec:
  command: 'examples/training.py'
  args: []
  image: .mlrun/func-default-remote-demo-ps-latest
  image_pull_policy: Always
  build:
    base_image: mlrun/mlrun:1.2.0
    source: git://github.com/mlrun/mlrun
```

Then, run the following CLI command and pass the path to your local function-configuration file as an argument to build the function's container image according to the configured requirements. For example, the following command builds the function using the **myfunc.yaml** file from the current directory:

```
mlrun build myfunc.yaml
```

When the build completes, you can use the `run` CLI command to run the function. Set the `-f` option to the path to the local function-configuration file, and pass the relevant parameters. For example:

```
mlrun run -f myfunc.yaml -w -p p1=3
```

You can also try the following function-configuration example, which is based on the MLRun CI demo:

```
kind: job
metadata:
  name: remote-git-test
  project: default
  tag: latest
spec:
  command: 'myfunc.py'
  args: []
  image_pull_policy: Always
  build:
    commands: []
    base_image: mlrun/mlrun:1.2.0
    source: git://github.com/mlrun/ci-demo.git
```

18.3 Using a sources archive

The `-a|--archive` option of the CLI *build* command enables you to define a remote object path for storing TAR archive files with all the required code dependencies. The remote location can be, for example, in an AWS S3 bucket or in a data container in an Iguazio MLOps Platform ("platform") cluster. Alternatively, you can also set the archive path by using the `MLRUN_DEFAULT_ARCHIVE` environment variable. When an archive path is provided, the remote builder archives the configured function sources (see the `-s|-source` *build* option) into a TAR archive file, and then extracts (untars) all of the archive files (i.e., the function sources) into the configured archive location.

To use the archive option, first create a local function-configuration file. For example, you can create a **function.yaml** file in your working directory with the following content; the specification describes the environment to use, defines a Python base image, adds several packages, and defines **examples/training.py** as the application to execute on `run` commands:

```
kind: job
metadata:
  name: remote-demo4
  project: ''
spec:
  command: 'examples/training.py'
  args: []
  image_pull_policy: Always
  build:
    commands: []
    base_image: mlrun/mlrun:1.2.0
```

Next, run the following MLRun CLI command to build the function; replace the `<...>` placeholders to match your configuration:

```
mlrun build <function-configuration file path> -a <archive path/URL> [-s <function-
↪sources path/URL>]
```

For example, the following command uses the **function.yaml** configuration file (`.`), relies on the default function-sources path (`./`), and sets the target archive path to `v3io:///users/$V3IO_USERNAME/tars`. So, for a user named "admin", for example, the function sources from the local working directory will be archived and then extracted into an **admin/tars** directory in the "users" data container of the configured platform cluster (which is accessed via the `v3io` data mount):

```
mlrun build . -a v3io:///users/$V3IO_USERNAME/tars
```

Note:

- `.` is a shorthand for a **function.yaml** configuration file in the local working directory.
- The `-a|--archive` option is used to instruct MLRun to create an archive file from the function-code sources at the location specified by the `-s|--sources` option; the default sources location is the current directory (`./`).

After the function build completes, you can run the function with some parameters. For example:

```
mlrun run -f . -w -p p1=3
```

CHAPTER

NINETEEN

GLOSSARY

19.1 MLRun terms

MLRun terms	Description
Feature set	A group of features that are ingested together and stored in logical group. See Feature sets.
Feature vector	A combination of multiple Features originating from different Feature sets. See Creating and using feature vectors.
HTTPRunDB	API for wrapper to the internal DB in MLRun. See mlrun.db.httpdb.HTTPRunDB.
hub	Used in code to reference the MLRun Function Hub.
MLRun function	An abstraction over the code, extra packages, runtime configuration and desired resources which allow execution in a local environment and on various serverless engines on top of K8s. See MLRun serverless functions and Creating and using functions.
MLRun Function Hub	A collection of pre-built MLRun functions avilable for usage. See MLRun Function Hub.
MLRun project	A logical container for all the work on a particular activity/application that include functions, workflow, artifacts, secrets, and more, and can be assigned to a specific group of users. See Projects.
mpijob	One of the MLRun batch runtimes that runs distributed jobs and Horovod over the MPI job operator, used mainly for deep learning jobs. See MLRun MPIJob and Horovod runtime.
Nuclio function	Subtype of MLRun function that uses the Nuclio runtime for any generic real-time function. See Nuclio real-time functions and Nuclio documentation.
Serving function	Subtype of MLRun function that uses the Nuclio runtime specifically for serving ML models or real-time pipelines. See Real-time serving pipelines (graphs) and Model serving pipelines.
storey	Asynchronous streaming library for real time event processing and feature extraction. Used in Iguazio's feature store and real-time pipelines. See storey.transformations - Graph transformations.

19.2 Iguazio (V3IO) terms

Name	Description
Consumer group	Set of consumers that cooperate to consume data from some topics.
Key Value (KV) store	Type of storage where data is stored by a specific key, allows for real-time lookups.
V3IO	Iguazio real-time data layer, supports several formats including KV, Block, File, Streams, and more.
V3IO shard	Uniquely identified data sets within a V3IO stream. Similar to a Kafka partition.
V3IO stream	Streaming mechanism part of Iguazio's V3IO data layer. Similar to a Kafka stream.

19.3 Standard ML terms

Name	Description
Artifact	A versioned output of a data processing or model training jobs, can be used as input for other jobs or pipelines in the project. There are various types of artifacts (file, model, dataset, chart, etc.) that incorporate useful metadata. See Artifacts.
DAG	Directed acyclic graph, used to describe workflows/pipelines.
Feature engineering	Apply domain knowledge and statistical techniques to raw data to extract more information out of data and improve performance of machine. learning models
EDA	Exploratory data analysis. Used by data scientists to understand dataset via cleaning, visualization, and statistical tests.
ML pipeline	Pipeline of operations for machine learning. It can include loading data, feature engineering, feature selection, model training, hyperparameter tuning, model validation, and model deployment.
Feature	Data field/vector definition and metadata (name, type, stats, etc.). A dataset is a collection of features.
MLOps	Set of practices that reliably and efficiently deploys and maintains machine learning models in production. Combination of Machine Learning and DevOps.
Dataframe	Tabular representation of data, often using tools such as Pandas, Spark, or Dask.

19.4 ML libraries / tools

Name	Description
Dask	Flexible library for parallel computing in Python. Often used for data engineering, data science, and machine learning.
Keras	An open-source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.
Kube-Flow pipeline	Platform for building and deploying portable, scalable machine learning (ML) workflows based on Docker containers.
PyTorch	An open source machine learning framework based on the Torch library, used for applications such as computer vision and natural language. processing
Sklearn	Open source machine learning Python library. Used for modelling, pipelines, data transformations, feature engineering, and more.
Spark	Open source parallel processing framework for running large-scale data analytics applications across clustered computers. Often used for data engineering, data science, and machine learning.
Tensor-Flow	A Google developed open-source software library for machine learning and deep learning.
Tensor-Board	TensorFlow's visualization toolkit, used for tracking metrics like loss and accuracy, visualizing the model graph, viewing histograms of weights, biases, or other tensors as they change over time, etc.
XGBoost	Optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. Implements machine learning algorithms under the Gradient Boosting framework.

Back to top

PYTHON MODULE INDEX

m

s

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W