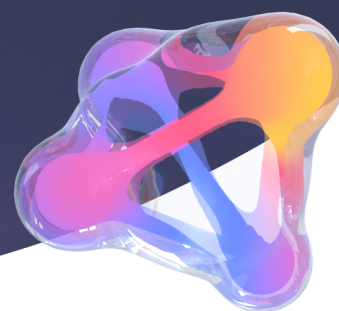


The Complete Guide to Using the Iguazio Feature Store with Azure ML



In this e-book, we will showcase an end-to-end hybrid cloud ML workflow using the Iguazio MLOps Platform & Feature Store combined with Azure ML. The first part gives an overview of the solution and the types of problems it solves, and the following parts are a technical deep dive into each step of the process.

Part 1: Feature Store Motivation

Part 2: Data Ingestion + Transformation via Iguazio's Feature Store

- Detailed overview of Iguazio feature store functionality
- Ingest and transform dataset into feature store
- Retrieve features in batch and real-time

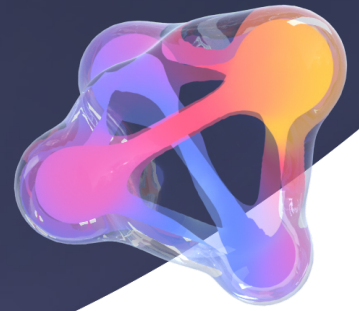
Part 3: Model Training via Azure ML

- Upload/register features from the Iguazio feature store into Azure ML
- Orchestrate Azure ML training job from the Iguazio platform
- Download trained model(s) + metadata from Azure back into the Iguazio platform

Part 4: Hybrid Cloud + On-Premise Model Serving + Model Monitoring

- Deploy models to a real-time HTTP endpoint
- Combine multiple models into a voting ensemble
- Integrate model serving with real-time feature retrieval

Part 1: Feature Store Motivation



The Gaps: Challenges When Operationalizing Data Science

Regardless of the environment, one of the main challenges when operationalizing data science is fostering collaboration between teams, and eliminating tech silos. This is both a technological and organizational challenge, requiring the right processes in place and the right tools to support these processes.

In a typical data science project, there are usually three different teams involved at different points in the pipeline:

- Data Engineer: Ingest and transform raw data from various sources
- Data Scientist: Utilize transformed data to train model
- MLOps Engineer: Containerize and deploy model at scale with monitoring, drift detection, and re-training capabilities



While the pipeline itself looks straightforward, there are more than a few places where things can go wrong - mostly at the handoff points between teams. What happens when a data scientist needs additional or different features? What happens when the MLOps Engineer cannot anticipate a model issue the data scientist knew about? Or a data issue the Data Engineer knew about?

These organizational challenges can be solved, in part, with central artifact/job management and a feature store.

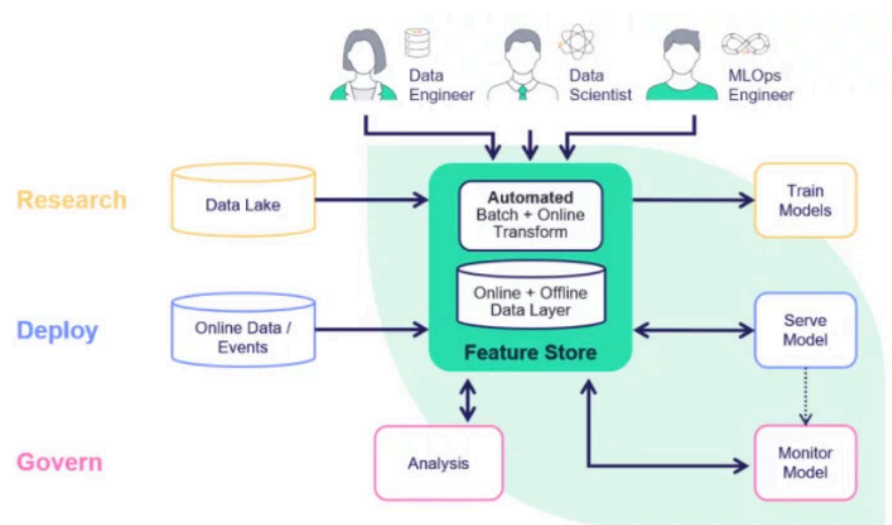
Filling the Gaps with Iguazio's Feature Store

The feature store is a relatively recent concept in the world of MLOps. The purpose and function of a feature store is slightly different depending on who is asked. However, most can agree on the main definition: a central place where teams can store and access features, and share them across projects. You can read more about the necessity of feature stores for scaling data science [here](https://www.iguazio.com/blog/feature-store-for-scaling-data-science/).

For Iguazio's feature store, storing and retrieving features is the bare minimum. Also available out of the box are custom batch/real-time pipelines to transform data upon ingestion, dual storage formats to facilitate batch and real-time workloads, and integration with model monitoring and serving.

This allows the feature store to be used as a central communication plane across projects, jobs, models, artifacts, transformation pipelines, etc. The Iguazio Feature Store also functions as a data transformation service, enabling complex feature calculations such as the customers' mean purchases in the last 24 hours, financial transactions in the last 12 hours, or other sliding window aggregations.

Not only does this standardize the overall ML workflow, it uniquely benefits each team in the pipeline:



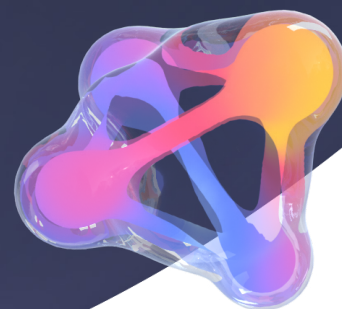
- Data Engineer: Allows for batch/real-time data ingestion and transformation pipelines
- Data Scientist: Allows for easily accessing features and reducing duplicated work, and the harnessing of complex real-time features for their predictions
- MLOps Engineer: Allows for access of features in real-time for model serving and monitoring

Next Steps: Iguazio + Azure

Despite the number of fantastic services available in the Azure ecosystem, a feature store as described above is not one of them. Additionally, due to the inherent nature of cloud services, they typically cannot be used for on-premise workloads or serving. This is where the Iguazio MLOps Platform can fill in some gaps.

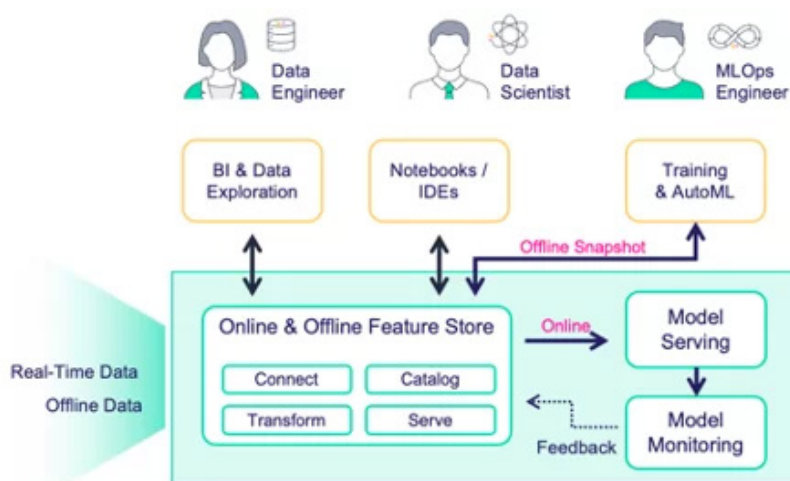
In the next three sections, we will build out a full end-to-end hybrid cloud ML workflow using features from Azure and Iguazio.

Part 2: Data Ingestion + Transformation into the Iguazio Feature Store



Overview of Iguazio's Feature Store

While some feature stores are more focused in their functionality, Iguazio's feature store is designed to facilitate the entire end-to-end workflow:



In part one, we discussed the Iguazio feature store at a high level. The functionalities include (but are not limited to) the following:

- Ingest and transform data sources in batch or real-time
- Easily retrieve features in batch or real-time
- Dual storage formats to facilitate batch and real-time workloads
- Complex real-time feature engineering (eg. Sliding window aggregations)
- Integration with model monitoring
- Integration with model serving

We will be exploring all of this functionality in this e-book.

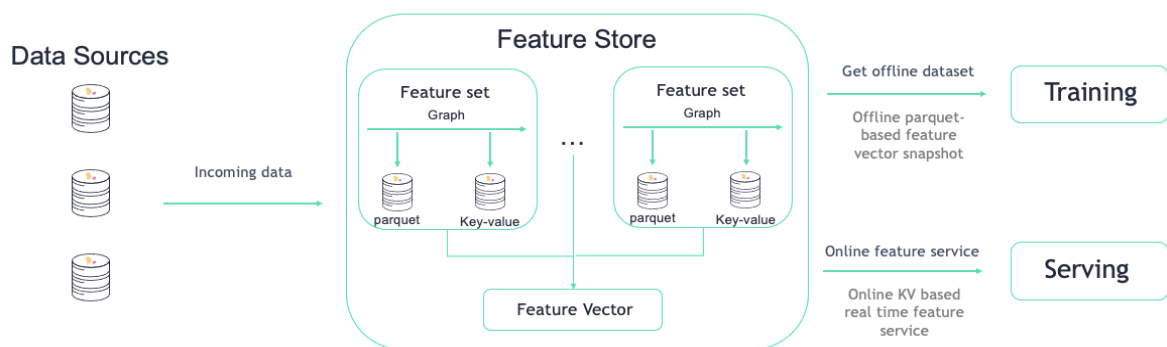
Iguazio Terminology: FeatureSet and FeatureVector

The Iguazio feature store introduces two new terms:

- **FeatureSet**: group of features from one data source (file, data frame, table, etc.)
- **FeatureVector**: group of features from one or more **FeatureSets** (i.e. a few columns from here, a few columns from there)

These act as the building blocks of the Iguazio feature store and will be used heavily throughout this blog series.

They fit into the overall picture as described in this diagram:



Each data source will be ingested as a **FeatureSet** with an optional transformation graph. Once ingested, the data will be stored as parquet for offline usage and KV for online usage.

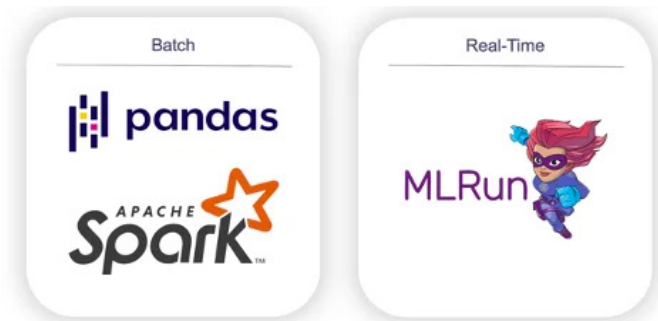
From there, a **FeatureVector** is created as a combination of multiple **FeatureSet**'s. This allows for simple joining of multiple **FeatureSet**'s during retrieval using either the offline or online file format.

You can read more about the Feature Store, **FeatureSet** and **FeatureVector** in the [MLRun documentation](#).

Feature Store Transformation Engines

Not only is the Iguazio feature store useful for ingesting and cataloging features, it can also be used to perform data transformations in batch and real time. These transformations will be added to the **FeatureSet** in the form of a transformation graph, where each step is a Python class. The Python class itself can be a pre-built class or a custom implementation.

What exactly the Python class does depends on the desired transformation engine. Currently, the feature store supports the following:



- **Pandas**
 - Batch transformations that fit into memory
 - Input Pandas data frame → transform → output pandas dataframe
- **Spark**
 - Batch transformations that to not fit into memory
 - Input Spark data frame → transform → output Spark data frame
- **MLRun**
 - Real-time transformations (one record at a time)
 - Input OrderedDict → transform → output OrderedDict

Implementing each of the different engines is quite simple and explained fully in the Feature Set transformation documentation.

Dataset Overview

For this demo, we will use the Heart Disease Dataset from the UCI Machine Learning Repository to train a model that detects whether a given patient has heart disease.

For the purposes of this demo, we have generated a unique ID per record called ``patient_id``, separated the features into a file titled ``features.csv``, and separated the target into a file titled ``target.csv``. We will be using these CSV files as the inputs for our feature store ingestion.

Feature Store Resources to Create

For this project, we will be creating the following resources in the feature store:

- FeatureSet #1 - ``heart_disease_features``
 - Ingest data from local ``features.csv`` file
 - Perform transformations in real-time to incoming data
- FeatureSet #2 - ``heart_disease_target``
 - Ingest data from local ``target.csv`` file
 - No transformations
- FeatureVector ``heart_disease_vec``
 - Combine all features from ``heart_disease_features`` and target variable from ``heart_disease_target`` into singular dataset for training and serving

Ingest and Transform Feature Sets

Let's get into the actual code, starting with imports and creating the project:

```
# MLRun imports

import mlrun.feature_store as fstore

from mlrun.feature_store.steps import MapValues, OneHotEncoder

from mlrun.datastore.sources import CSVSource

from mlrun import get_or_create_project

# Create MLRun project

project = get_or_create_project(name="azure-fs-demo", context=".")
```

FeatureSet #2 Ingestion - `heart_disease_target`

The `FeatureSet` ingestion itself is fairly straightforward. We will start with `heart_disease_target` as there is no transformation component.

The raw `target.csv` dataset looks like this:

patient_id	target
e4d3544b-609e-476c-9873-e2d8113bae0	0
822763d6-76a6-4412-8aed-99472302d680	0
13c4b48e-a040-440e-8ab6-90c8c63792c4	0
f0e6cd22-7aed-4817-a671-e1362719c8d8	0
2d0b3bca-4841-4618-8a6c-ca302019c009	0

First, create desired `FeatureSet`:

```
# Create feature set with name and entity (primary key)

heart_disease_target = fstore.FeatureSet(name="heart_disease_target",
                                         entities=[fstore.Entity("patient_id")])

# Set default parquet/KV targets

heart_disease_target.set_targets()
```

To ingest, we simply pass in our newly created `FeatureSet` and desired source:

```
# CSV source - customizable
source = CSVSource("mycsv", path="target.csv")

# Ingest into feature store - returns transformed dataframe
resp = fstore.ingest(featureset=heart_disease_target, source=source)

# Print top 5 rows of newly ingested data
resp.head()
```

	target
patient_id	
e443544b-8d9e-4f6c-9623-e24b6139aae0	0
8227d3df-16ab-4452-8ea5-99472362d982	0
10c4b4ba-ab40-44de-8aba-6bdb062192c4	0
f0acdc22-7ee6-4817-a671-e136211bc0a6	0
2d6b3bca-4841-4618-9a8c-ca902010b009	0

FeatureSet #1 Ingestion/Transformation - `heart_disease_features`

Next, we will ingest `heart_disease_features` and specify a transformation graph using the real-time engine. This will allow us to transform records as we ingest (in addition to re-using this graph later in the project during model serving).

The raw `features.csv` dataset looks like this:

patient_id	age	sex	cp	resttpe	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
e443544b-8d9e-4f6c-9623-e24b6139aae0	53	male	typical_angina	125	212	False	1	168	no	1.0	downsloping	2.0	normal
8227d3df-16ab-4452-8ea5-99472362d982	69	male	typical_angina	140	203	True	0	159	yes	3.1	uprising	0.0	normal
10c4b4ba-ab40-44de-8aba-6bdb062192c4	70	male	typical_angina	145	176	False	1	175	yes	2.6	uprising	0.0	normal
f0acdc22-7ee6-4817-a671-e136211bc0a6	61	male	typical_angina	148	203	False	1	161	no	0.0	downsloping	1.0	normal
2d6b3bca-4841-4618-9a8c-ca902010b009	62	female	typical_angina	158	204	True	1	196	no	1.0	flat	3.0	reversible_defect

First, create desired `FeatureSet` with transformation graph:

```
# Create feature set with name and entity (primary key)
heart_disease_features = fstore.FeatureSet(name="heart_disease_features",
                                           entities=[fstore.Entity("patient_id")])

# Set default parquet/KV targets
heart_disease_features.set_targets()
```



```

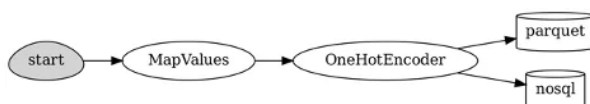
# Transformation mapping - specific to MapValues component
age_mapping = {'age': {'ranges': [{ 'range': [0, 3], "value": "toddler"},
                                   { 'range': [3, 18], "value": "child"},
                                   { 'range': [18, 65], "value": "adult"},
                                   { 'range': [65, 120], "value": "elder"}]}}

# Transformation mapping - specific to OneHotEncoder component
one_hot_encoder_mapping = {'age_mapped': ['toddler', 'child', 'adult', 'elder'],
                           'sex': ['male', 'female'],
                           'cp': ['typical_angina', 'atypical_angina', 'non_anginal_pain',
                                  'asymptomatic'],
                           'exang': ['no', 'yes'],
                           'fbs': [False, True],
                           'slope': ['downsloping', 'upsloping', 'flat'],
                           'thal': ['normal', 'reversable_defect', 'fixed_defect']}

# Add Python classes to transformation graph (with configuration)
heart_disease_features.graph.to(MapValues(mapping=age_mapping, with_original_features=True))\
                              .to(OneHotEncoder(mapping=one_hot_encoder_mapping))

# Print out transformation graph
heart_disease_features.plot(rankdir='LR', with_targets=True)

```



Like before, to ingest we simply pass in our newly created `FeatureSet` and desired source:

```

# CSV source - customizable
source = CSVSource("mycsv", path="features.csv")

# Ingest into feature store - returns transformed dataframe
resp = fstore.ingest(featureset=heart_disease_features, source=source)

```

```
# Print top 5 rows of newly ingested data
```

```
resp.head()
```

	age_mapped_toddler	age_mapped_child	age_mapped_adult	age_mapped_elder	age	sex_male	sex_female	cp_typical_angina	cp_atypical_s
patient_id									
e643544b-8d9a-489c-9623-e24b6199ae0	0	0	1	0	52	1	0	1	0
8227d3d1-18ab-4482-8a6f-904723d2d982	0	0	1	0	53	1	0	1	0
10c4b4ba-ab40-44da-8aba-9b49062192e4	0	0	0	1	70	1	0	1	0
9ba0d222-7ee6-4817-a671-e136211bc0a8	0	0	1	0	61	1	0	1	0
2d6b3bca-4841-4818-9d8c-ca902010c009	0	0	1	0	62	0	1	1	0

5 rows x 27 columns

Create FeatureVector

Now that we have two `FeatureSet`'s, we will combine them into a singular dataset called a `FeatureVector`. The backend API will handle joining multiple data sources for us.

Creating the `FeatureVector` is quite simple as well:

```
# Create feature vector with list of features + desired label
# Specify in format of FEATURE_SET.FEATURE (or * for all)
vector = fstore.FeatureVector(
    name="heart_disease_vec",
    features=["heart_disease_features.*"],
    label_feature="heart_disease_target.target",
    with_indexes=True
)
```

That's it.

Retrieve Offline Features

To retrieve the `FeatureVector` using the offline (parquet) format, use the following:

```
# Retrieve feature vector as pandas dataframe
# Specify in format of PROJECT/FEATURE_VECTOR
fstore.get_offline_features("azure-fs-demo/heart_disease_vec").to_dataframe().head()
```

	age_mapped_todtder	age_mapped_chi	age_mapped_adult	age_mapped_older	age	sex_male	sex_female	cp_typical_angina	cp_atypical_a	p_non_anginal_pain	exang_yes	oldpeak	slope_downsloping	slope_upsloping	slope_flat	ca	thal_normal	thal_reversible_defect	thal_fixed_defect	target
patient_id																				
e4c354b-859a-43b0-9623-c24b6139aae0	0	0	1	0	52	1	0	1	0	...	0	1.0	1	0	0	2.0	1	0	0	0
f22743df-15ab-4452-baed-9947236f0992	0	0	1	0	53	1	0	1	0	...	1	3.1	0	1	0	0.0	1	0	0	0
10c4b4ba-a040-44de-8ada-6cc0062192d4	0	0	0	1	70	1	0	1	0	...	1	2.6	0	1	0	0.0	1	0	0	0
fba5dc25-7ee6-4817-a071-e136211bc036	0	0	1	0	61	1	0	1	0	...	0	0.0	1	0	0	1.0	1	0	0	0
2a093eca-4041-4019-9af6-ca020101000	0	0	1	0	62	0	1	1	0	...	0	1.9	0	0	1	3.0	0	1	0	0

5 rows x 26 columns

Retrieve Online Features

```
# Retrieve online feature service
# Specify in format of PROJECT/FEATURE_VECTOR
feature_service = fstore.get_online_feature_service("azure-fs-demo/heart disease vec")
```

```
feature_service.vector.get_stats_table().head()
```

Or retrieve individual records with real-time speed:

```
[{'patient_id': 'e443544b-8d9e-4f6c-9623-e24b6139aae0',
 'age_mapped_toddler': 0,
 'age_mapped_child': 0,
 'age_mapped_adult': 1,
 'age_mapped_elder': 0,
 'age': 52,
 'sex_male': 1,
 'sex_female': 0,
 'cp_typical_angina': 1,
 'cp_atypical_angina': 0,
 'cp_non_anginal_pain': 0,
 'cp_asymptomatic': 0,
 'trestbps': 125,
 'chol': 212,
 'fbs_False': 1,
 'fbs_True': 0,
 'restecg': 1,
 'thalach': 168,
 'exang_no': 1,
 'exang_yes': 0,
 'oldpeak': 1.0,
 'slope_downsloping': 1,
 'slope_upsloping': 0,
 'slope_flat': 0,
 'ca': 2.0,
 'thal_normal': 1,
 'thal_reversible_defect': 0,
 'thal_fixed_defect': 0}]
```

FIND OUT MORE ON www.iquazio.com

Feature Store UI

Finally, we are able to view everything we've done in the Iguazio project UI:

The first screenshot shows the 'heart_disease_features' configuration page. It displays a flow diagram with 'Source' pointing to 'MapValues', which then points to 'OneHotEncoder'. The 'Configuration' panel on the right shows the 'Function template' and 'Source' sections, with 'Steps' set to 'Task'.

The second screenshot shows the 'heart_disease_target' preview page. It displays a table with two columns: 'patient_id' and 'target'. The 'target' column contains binary values (0 or 1) for each patient ID.

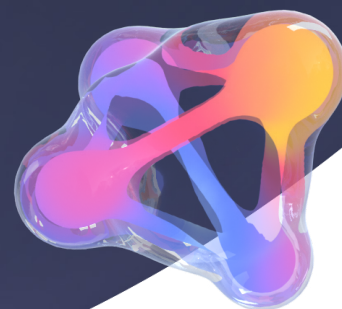
The third screenshot shows the 'heart_disease_vec' statistics page. It displays a table with various statistical metrics for different features. The 'Statistics' panel on the right shows histograms for each feature.

Name	Count	Mean	Std	Min	Max	Unique	Typ	Freq	Histogram
age_mapped_tot...	968	0	0.00000000	0	0	1	0	0	
age_mapped_chi...	968	0	0.00000000	0	0	1	0	0	
age_mapped_ad...	968	0.873	0.33089987	0	1	2	0	0	
age_mapped_abb...	968	0.125	0.33089987	0	1	2	0	0	
age	968	54.672247...	9.12789399	29	77	1	0	0	
sex_male	968	0.7159690...	0.45121346	0	1	2	0	0	
sex_female	968	0.2840309...	0.45121346	0	1	2	0	0	
cp_typical_angi...	968	0.4731484...	0.49953914	0	1	2	0	0	

Next Steps

In the next two blogs, we will build upon the features we just ingested to build out a full end-to-end hybrid cloud ML workflow using features from Azure and from Iguazio.

Part 3: Model Training with Azure ML



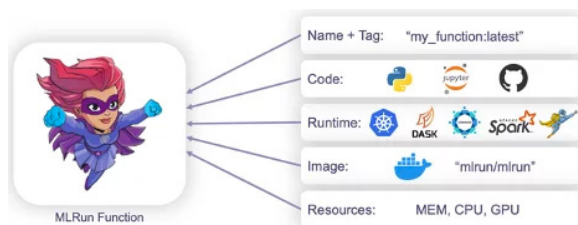
In this part, we will:

- Upload data from Iguazio into Azure and register dataset in Azure ML
- Train several models in Azure using Auto ML
- Retrieve trained models from Azure back to Iguazio
- Log trained models with experiment tracking and metrics

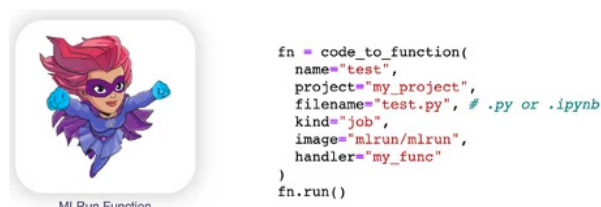
MLRun Functions: Overview

Before running any code, we need to take a moment to discuss one of the core tenants of the Iguazio platform - the MLRun function.

This abstraction allows for simple containerization and deployment of code. Users are able to execute workloads on Kubernetes by specifying code and configuration options via high level Python syntax. An MLRun function will look something like this:



The function will have its own name/tag, code, docker image, resources, and runtime. The syntax for creating an MLRun function looks like the following:



From there, the code can be executed locally in Jupyter or on the cluster using several runtime engines including Job, Spark, Dask, Horovod, and Nuclio real-time functions.

Create Azure MLRun Function from Python File

Now that we have the background on what an MLRun function is, we are going to create one with our Azure code. We have written a Python file called `azure_automl.py` that performs several tasks such as:

- Upload the dataset from the feature store
- Register the dataset in AzureML
- Execute a training job using Azure AutoML
- Download the trained models back to Iguazio
- Log models with experiment tracking metadata such as labels and metrics

First we will configure our code to use the same project that our features reside in:

```
import mlrun

from mlrun import get_or_create_project, code_to_function, build_function, run_function

project = get_or_create_project(name="azure-fs-demo", context=".")
```

Next, we will use `code_to_function` from the MLRun library to convert our Python file into an MLRun function:

```
azure_automl = code_to_function(
    name="azure",                # Name for function in project
    filename="azure_utils.py",   # Python file where code resides
    kind="job",                  # Kubernetes Job
)

azure_automl.save()              # Save function in project
```

This will create our MLRun function and store it in the project. Now, we can execute our code on top of the Kubernetes cluster with ease.

Build Docker Image

However, the code requires some Python packages for Azure services that are not included in the default `mlrun/mlrun` Docker image. We can easily build our docker image and update our function with the following:

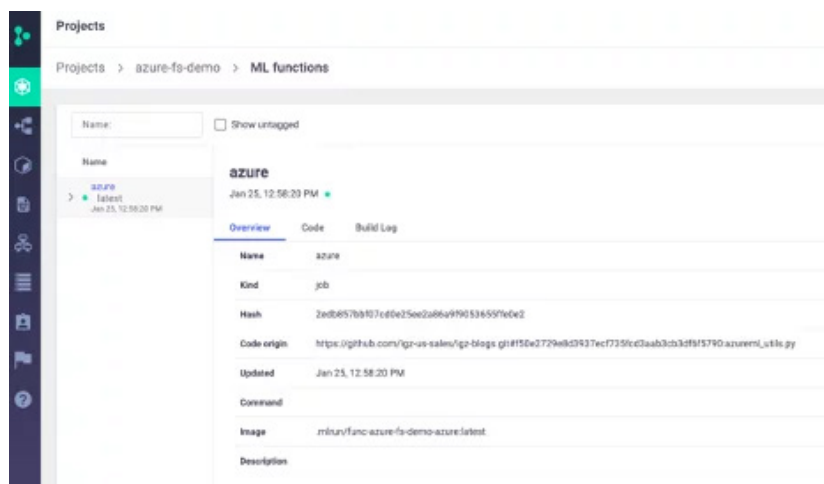
```
build_function(
    function="azure",            # Name of MLRun function we created
    skip_deployed=False,        # Force rebuild
    with_mlrun=False,           # MLRun already installed - no need to reinstall
    base_image="mlrun/mlrun:0.8.0", # Base Docker image
    requirements="requirements.txt" # Required packages
)
```

Because of how we configured our MLRun function, this one line of code will do the following:

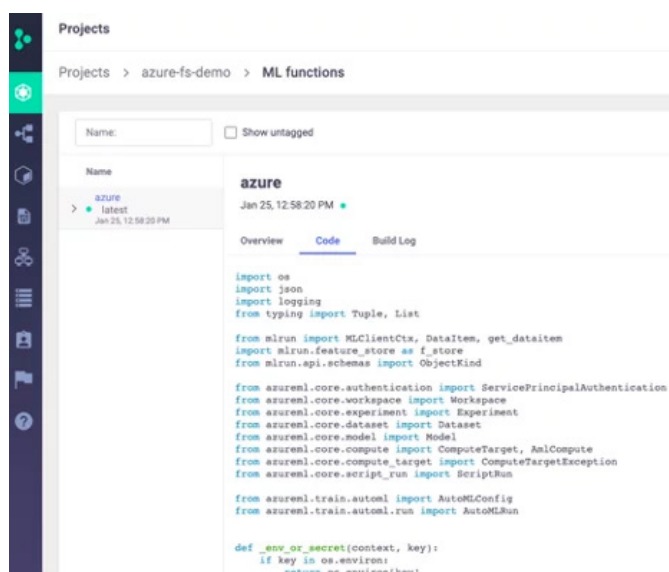
- Fetch the function titled `azure` from our project
- Start building an image with the base image `mlrun/mlrun:0.8.0`
- Install packages specified in `requirements.txt`
- Update the `azure` function in the project with the newly built image

View MLRun Function in Project

Now that we have created our MLRun function and built the required Docker image, we can see what we built in the MLRun UI:



Here we can see some of the high level information including the name, code origin, runtime, and image. Additionally, we can see the code itself:



Note that these MLRun functions can be versioned. This is not a replacement for version control software like Git, but is a useful tool to track which models are tied to which version and how your code has changed over time.

Add Kubernetes Secrets

An additional step before executing our code is to create some project secrets. This will allow us to securely retrieve the Azure credentials without storing sensitive information in the code itself. We can easily do this like so:

```
mlrun.get_run_db().create_project_secrets(  
    project.name,  
    provider=mlrun.api.schemas.SecretProviderName.kubernetes,  
    secrets={  
        "AZURE_TENANT_ID": "XXXXXXXX",  
        "AZURE_SERVICE_PRINCIPAL_ID": "XXXXXXXX",  
        "AZURE_SERVICE_PRINCIPAL_PASSWORD": "XXXXXXXX",  
        "AZURE_SUBSCRIPTION_ID": "XXXXXXXX",  
        "AZURE_RESOURCE_GROUP": "XXXXXXXX",  
        "AZURE_WORKSPACE_NAME": "XXXXXXXX",  
        "AZURE_STORAGE_CONNECTION_STRING": "XXXXXXXX"  
    }  
)
```

This only ever needs to be run once to store the credentials in a Kubernetes secret. **Do not commit these secrets to version control as they will allow anyone to spin up Azure resources.**

To insert our secrets into the MLRun function, we will create a MLRun task like so:

```
secrets_spec = mlrun.new_task().with_secrets(  
    kind='kubernetes',  
    source=[  
        'AZURE_TENANT_ID',  
        'AZURE_SERVICE_PRINCIPAL_ID',  
        'AZURE_SERVICE_PRINCIPAL_PASSWORD',  
        'AZURE_SUBSCRIPTION_ID',  
        'AZURE_RESOURCE_GROUP',  
        'AZURE_WORKSPACE_NAME',  
        'AZURE_STORAGE_CONNECTION_STRING'  
    ]  
)
```

We will use this when executing the code itself. Notice that we are only specifying the names of the secrets we want to retrieve - the values are stored securely as a Kubernetes secret.

Configure AutoML Job

Before executing our job, we need to specify what we want to run. Because we have wrapped all the functionality in our MLRun function, we only need to pass in some configuration parameters to our function.

There are many configuration parameters in this MLRun function including:

- Which `FeatureVector` or dataset to use
- Registered dataset name and description in Azure
- Azure experiment and compute settings
- Registered model name and AutoML settings in Azure
- How many models to save to Iguazio

The most important settings for configuring the job itself are specified here:

```
# Azure ML settings for model training
automl_settings = {
    "task": 'classification',
    "enable_early_stopping" : False,
    "allowed_models": ['LogisticRegression', 'SGD', 'SVM'],
    "iterations" : 5,
    "n_cross_validations": 5,
    "primary_metric": 'accuracy',
    "featurization": 'off',
    "model_explainability": False,
    "enable_voting_ensemble": False,
    "enable_stack_ensemble": False
}

# MLRun input for dataset - FeatureVector that we previously created
inputs = {
    "dataset" : "store://feature-vectors/azure-fs-demo/heart_disease_vec:latest"
}

# MLRun parameters for job
params = {
    "experiment_name" : "azure-iguazio-blog",
    "cpu_cluster_name" : "azureml-cpu",
    "dataset_name" : "iris",
    "dataset_description" : "iris training data",
```

```

    "register_model_name": "iris-model",
    "label_column_name" : "target",
    "save_n_models" : 3,
    "automl_settings" : automl_settings
}

```

From this configuration, we can see:

- We are doing a classification task
- We are using the `FeatureVector` from the previous blog
- The label for the training set is titled `target`
- We will train 5 models total without early stopping
- The models will be of type `LogisticRegression`, `SGD`, or `SVM`
- We will save the top 3 models based on accuracy back to Iguazio

There are many options that we can specify in `automl_settings` - an exhaustive list can be found in the Azure documentation.

Run Azure Auto ML

Last but certainly not least, we can run our AutoML job on Azure. The culmination of all our work so far can be expressed in a single line of code:

```

run_function(
    function="azure",          # MLRun function in project
    handler="train",          # Entrypoint Python function
    inputs=inputs,            # Feature Vector input
    params=params,            # Configuration parameters
    base_task=secrets_spec    # Insert secrets
)

```

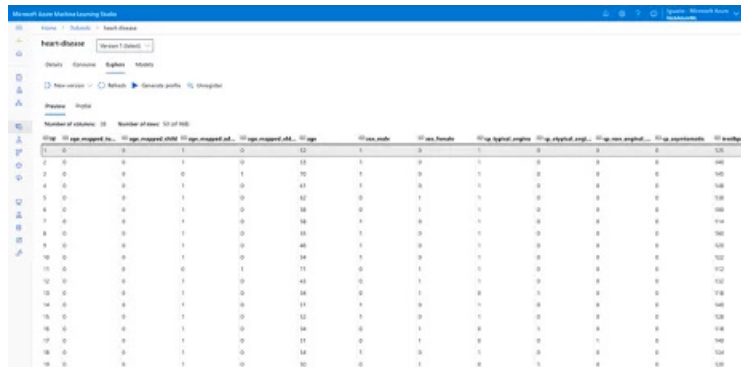
As explained above, this will:

- Upload the dataset from the feature store
- Register the dataset in AzureML
- Execute a training job using Azure AutoML
- Download the trained models back to Iguazio
- Log models with experiment tracking metadata such as labels and metrics

View Azure ML Training Job

Once our Azure job kicks off, we can use the `Microsoft Azure Machine Learning Studio UI` to keep track of everything.

This includes the registered dataset:



id	age	sex	cp	trestbps	chol	fbs	restecg	exang	oldpeak	st_slope
1	39	M	1	120	120	0	1	0	0	0
2	41	M	1	115	115	0	1	0	0	0
3	43	M	1	135	135	0	1	0	0	0
4	45	M	1	140	140	0	1	0	0	0
5	47	M	1	145	145	0	1	0	0	0
6	49	M	1	150	150	0	1	0	0	0
7	51	M	1	155	155	0	1	0	0	0
8	53	M	1	160	160	0	1	0	0	0
9	55	M	1	165	165	0	1	0	0	0
10	57	M	1	170	170	0	1	0	0	0

Also the completed job output:

Microsoft Azure Machine Learning Studio

Home

Experiments

azure-iguazio-blog

green_rod_sl05nf9

Refresh

Edit and submit

Cancel

Delete

Details

Data guardrails

Models

Outputs + logs

Child runs

Snapshot

Properties

Status

Completed

Script name

--

Created by

Service Principal

Created

Jan 25, 2022 12:41 PM

Started

Jan 25, 2022 12:41 PM

Duration

3m 6.540s

Compute duration

3m 6.540s

Compute target

azureml-cpu

Run ID

AutoML_065a2542-77aa-4494-a23e-1ff7c58b8a1a

Tags

No tags

Description

Click add rows to add a description

Best model summary

Algorithm name

StandardScale/Wrapper, SVM

Hyperparameters

See View hyperparameters

Accuracy

0.9989

See View all other metrics

Sampling

100.00%

Registered models

iris-model:135

Deploy status

No deployment yet

Run summary

Task type

Classification

See View configuration settings

Featurization

Auto

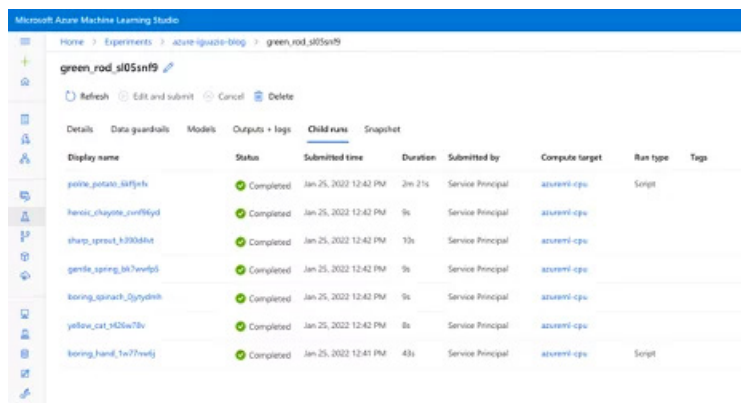
Primary metric

Accuracy

Experiment name

azure-iguazio-blog

This page displays a lot of useful information including the time taken, the best model, evaluation metrics, and more. We can also view the individual child runs for the model training:



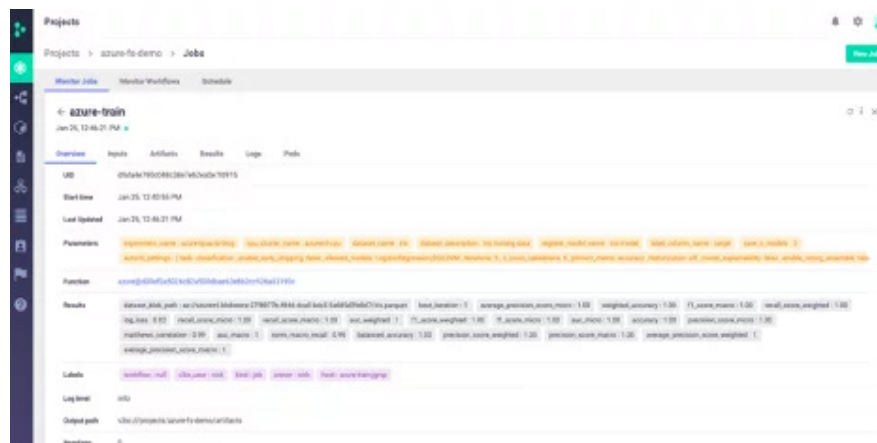
Display name	Status	Submitted time	Duration	Submitted by	Compute target	Run type	Tags
peixe_potato_sl05snf9	Completed	Jan 25, 2022 12:42 PM	2m 21s	Service Principal	azureml-cpu	Script	
terracotta_potato_sl05snf9	Completed	Jan 25, 2022 12:42 PM	9s	Service Principal	azureml-cpu		
sharp_potato_sl05snf9	Completed	Jan 25, 2022 12:42 PM	13s	Service Principal	azureml-cpu		
gentle_potato_sl05snf9	Completed	Jan 25, 2022 12:42 PM	9s	Service Principal	azureml-cpu		
boiling_potato_sl05snf9	Completed	Jan 25, 2022 12:42 PM	9s	Service Principal	azureml-cpu		
yellow_potato_sl05snf9	Completed	Jan 25, 2022 12:42 PM	8s	Service Principal	azureml-cpu		
boiling_potato_sl05snf9	Completed	Jan 25, 2022 12:41 PM	43s	Service Principal	azureml-cpu	Script	

We can see there are 6 jobs total - 1 setup job and the 5 training jobs we specified. Each of these child runs has additional information on time taken, evaluation metrics, and more.

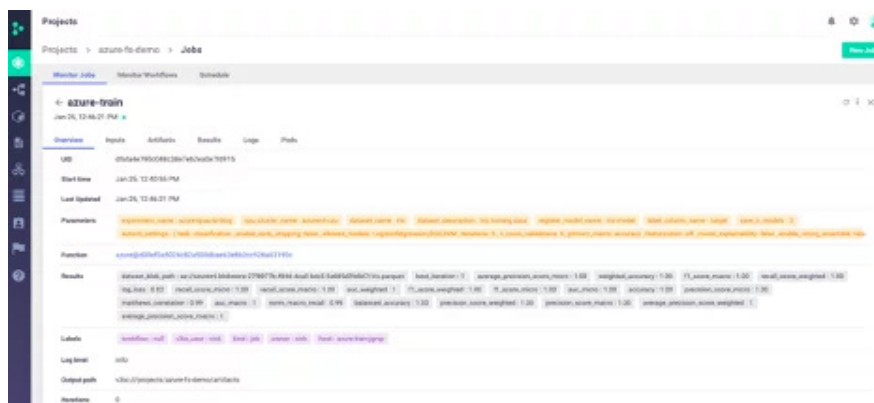
View MLRun Job

Although Azure ML is incredibly powerful and robust, the goal of this blog series is to integrate with Iguazio and bring the models back into the platform. Within the Iguazio UI, we can also view information about the job that ran.

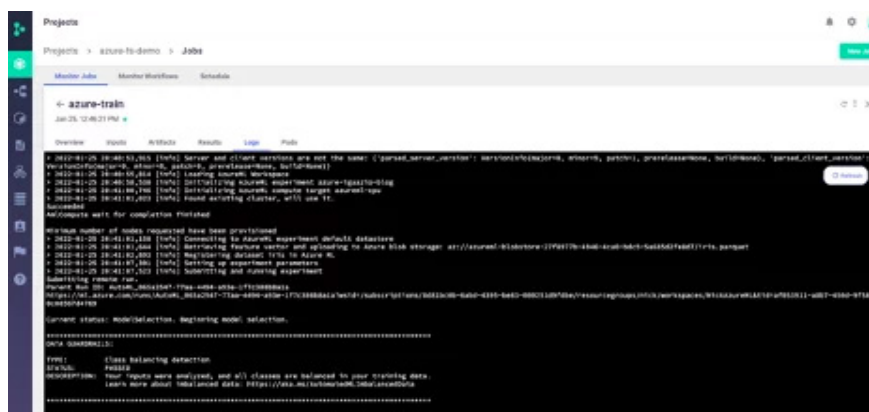
This includes an overview of the job with parameters, results, and even a link to the function code itself:



Additional tabs include the model artifacts:



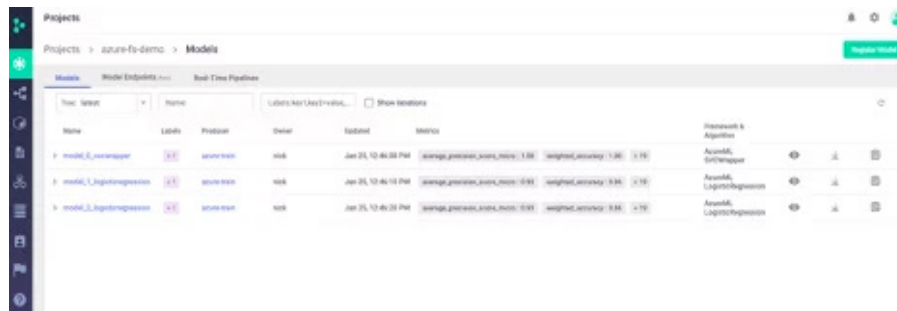
And Function logs:



All of this information is stored per run and is available to retrieve via the UI as seen here or programmatically through a Python SDK.

View Logged Models

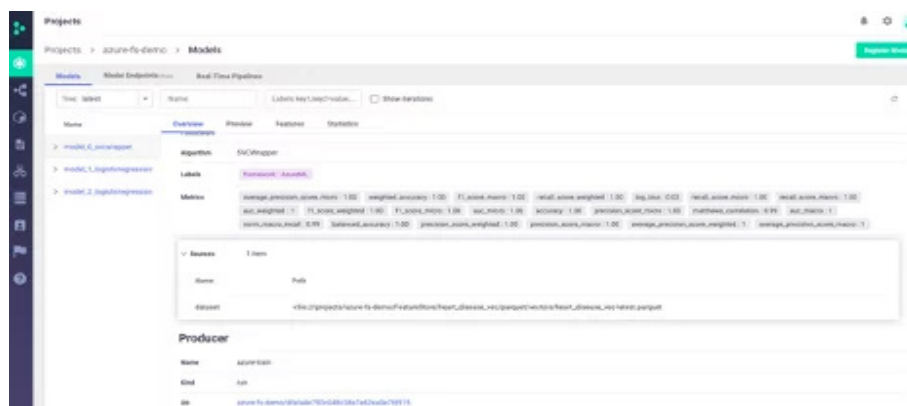
Finally, we can view the models that were logged as part of this training run. We are storing the model itself plus metadata on the algorithm, evaluation metrics, and custom labels:



The screenshot shows the 'Models' tab in the Iguazio UI. It displays a table with columns: Name, Labels, Protocol, Status, Started, Expired, Metrics, and Framework & Algorithm. Three models are listed:

Name	Labels	Protocol	Status	Started	Expired	Metrics	Framework & Algorithm
model_0_xgboost	0.5	secure-train	tick	Jan 25, 10:46:33 PM		average_precision_score_macro: 1.00 weighted_accuracy: 1.00 > 10	Amazon SageMaker XGBoost
model_1_logisticregression	0.5	secure-train	tick	Jan 25, 10:46:10 PM		average_precision_score_macro: 0.99 weighted_accuracy: 0.99 > 10	Amazon SageMaker LogisticRegression
model_2_logisticregression	0.5	secure-train	tick	Jan 25, 10:46:33 PM		average_precision_score_macro: 0.99 weighted_accuracy: 0.99 > 10	Amazon SageMaker LogisticRegression

We can also view more detailed information per model:



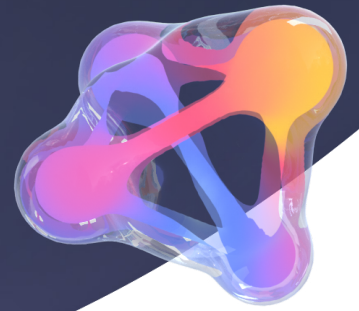
The screenshot shows the 'Overview' tab for a specific model. It displays detailed information about the model, including the algorithm, labels, metrics, and producer.

Algorithm	Labels	Metrics	Producer
XGBoost	0.5	average_precision_score_macro: 1.00 weighted_accuracy: 1.00 F1_score_macro: 1.00 recall_score_macro: 1.00 precision_score_macro: 1.00 accuracy: 1.00 roc_auc_score: 1.00 MatthewsCorrelationCoefficient: 1.00 JaccardIndex: 1.00	secure-train

This allows us to relate metrics, models, runs, code, and custom metadata all within the experiment tracking UI.

Overall, this combination allows for powerful integration between Iguazio's feature store and Azure's highly customizable AutoML training jobs - something that no single tool can do on its own.

Part 4: Hybrid Cloud + On-Premises Model Serving + Model Monitoring



In this final part, we will:

- Discuss the benefits of a hybrid cloud architecture
- Define model load and predict behavior
- Create a model ensemble using our top 3 trained models
- Enable real-time enrichment via the feature store during inferencing
- Deploy our model ensemble in a Jupyter notebook and on a Kubernetes cluster
- Enable model monitoring and drift detection
- View model/feature drift in specialized dashboards

Hybrid Cloud Benefits and Motivation

Hybrid clouds are all the rage. While cloud computing has given many organizations access to near infinite compute power, the reality is that on-premise infrastructure is not going away. From data privacy concerns, to latency requirements, to simply owning hardware, there are many legitimate reasons for having an on-premise footprint.

However, with this increased flexibility comes increased complexity – the right tools are needed for the job. With the multitude of end-to-end platforms, SaaS services, and cloud offerings, it is increasingly difficult to find those correct tools—especially those that work across cloud and on-premise environments.

In the last blog of this series, we will combine the feature store and on-premise model serving of the Iguazio MLOps Platform with the models trained via Azure's AutoML to create a full end-to-end hybrid cloud ML workflow.

Define Model Behavior

Now that we understand the motivation behind the on-premise deployment, the first step to deploying our models is to define the inferencing behavior. Essentially, how do we load our model and use it for predictions? MLRun Serving Graphs allow users to define a high level Python class for these behaviors, much like KFServing and Seldon Core.

Since all of the models we downloaded from AzureML are `'pickle'` files and have the same prediction syntax, this is a very straightforward class. The inferencing class resides in the `'azure_serving.py'` file and includes the following:

```
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()

```

Note that your class can be as complex as you need it to be with built-in hooks for ``preprocessing``, ``postprocessing``, ``validation``, and more. For more information on MLRun Serving Graphs, check out the [documentation](#).

Define Serving Function

After writing our simple Python class, the MLRun framework will take care of the building, provisioning, routing, enriching, aggregating, etc.

First we will configure our code to use the same project that our features and models reside in:

```

import os

from mlrun import get_or_create_project, code_to_function

project = get_or_create_project(name="azure-fs-demo", context=".")

```

Like in the previous blog, we will be using MLRun's ``code_to_function`` capabilities to containerize and deploy our function. However, unlike last time, we will be using a different runtime engine. Instead of a batch ``job``, we will be deploying a model ``serving`` function:

```

serving_fn = code_to_function(
    name='model-serving',          # Name for function in project
    filename="azure_serving.py",   # Python file where code resides
    kind='serving',                # MLRun Serving Graph
    image="mlrun/mlrun",           # Base Docker image
    requirements="requirements.txt" # Required packages
)

```

Configure Model Ensemble with Feature Store Enrichment

In addition to the initial `code_to_function` configuration, we will be adding additional options to the serving function. Behind the scenes, we are really constructing a graph with different topologies, preprocessing steps, models, etc.

In our case, we are looking for a model router that can:

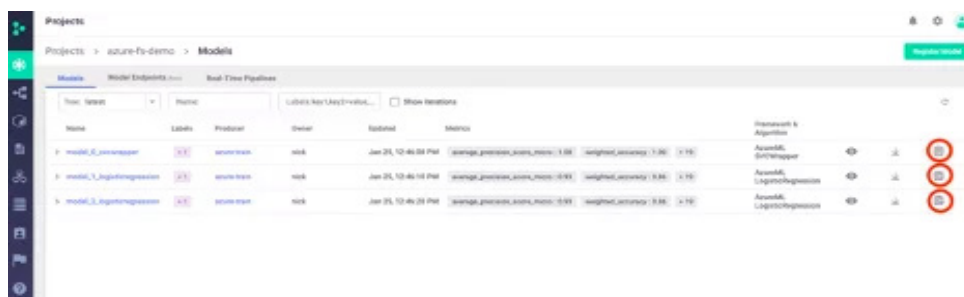
1. Enrich incoming data using the feature store
2. Impute any missing values using statistics from the feature store
3. Utilize multiple models in a voting ensemble
4. Return a single result from the ensemble

MLRun has such a router that comes out of the box called the `EnrichmentVotingEnsemble`. The configuration looks like this:

```
serving_fn.set_topology(
    topology='router',
    class_name='mlrun.serving.routers.EnrichmentVotingEnsemble',
    name='VotingEnsemble',
    feature_vector_uri="heart_disease_vec",
    impute_policy={"*": "$mean"}
)
```

Notice that we are configuring the desired `FeatureVector` to enrich with as well as the imputation method for missing values.

Now that the model router is set up like we want, we can add our models. We will be referencing the models using their URI's available via the MLRun UI. The "copy URI" icon is circled in red below:



We can add the models using very simple syntax. This sets the name of the model in the graph, the desired Python class to use for load/predict behavior, as well as the path to the model itself (using the URI we copied). This looks like the following:

```
serving_fn.add_model(
    key="model_0",
    class_name="ClassifierModel",
```

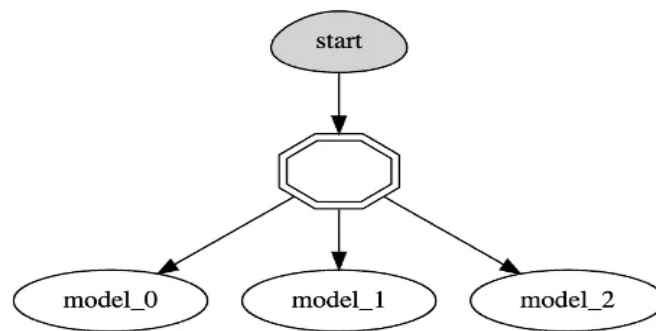


```

        model_path="store://models/azure-fs-demo/model_0_svcwrapper#0:latest"
    )
    serving_fn.add_model(
        key="model_1",
        class_name="ClassifierModel",
        model_path="store://models/azure-fs-demo/model_1_logisticregression#0:latest"
    )
    serving_fn.add_model(
        key="model_2",
        class_name="ClassifierModel",
        model_path="store://models/azure-fs-demo/model_2_logisticregression#0:latest"
    )

```

Finally, we can save our serving function in our project and visualize the computation graph:



```

serving_fn.save()
serving_fn.spec.graph.plot()

```

Before continuing, consider the complexity of what we just set up using a few lines of code:

- Define model load and predict behavior
- Create a model ensemble using our top 3 trained models
- Enable real-time enrichment via the feature store during inferencing
- Enable real-time imputation of missing values using feature store statistics

Test Locally in Jupyter Notebook

Now that our serving function and model ensemble are setup, we should test it out in our local Jupyter environment. MLRun makes this quite simple to do:

```

from azure_serving import ClassifierModel  # Need our serving class available for local
server

local_server = serving_fn.to_mock_server()  # Create local server for testing

```

```
> 2021-12-15 23:08:35,858 [info] model model_0 was loaded
> 2021-12-15 23:08:35,931 [info] model model_1 was loaded
> 2021-12-15 23:08:36,000 [info] model model_2 was loaded
```

From here, we can invoke the model using one of the `patient_ids` from our dataset. This will:

- Retrieve the corresponding record in the feature store
- Impute any missing values
- Send the fully enriched record to our model ensemble
- Aggregate and return the results of the model ensemble

This looks like the following:

```
local_server.test(
    path='/v2/models/infer',
    body={
        'inputs': [
            ["d107db82-fe26-4c02-b264-a3749510ed9b"]
        ]
    }
)

Input -> [[0, 0, 1, 0, 62, 0, 1, 1, 0, 0, 0, 138, 294, 0, 1, 1, 106, 1, 0, 1.9, 0, 0, 1, 3.0,
0, 1, 0]]

Input -> [[0, 0, 1, 0, 62, 0, 1, 1, 0, 0, 0, 138, 294, 0, 1, 1, 106, 1, 0, 1.9, 0, 0, 1, 3.0,
0, 1, 0]]

Input -> [[0, 0, 1, 0, 62, 0, 1, 1, 0, 0, 0, 138, 294, 0, 1, 1, 106, 1, 0, 1.9, 0, 0, 1, 3.0,
0, 1, 0]]

{'id': 'e36dedd8c72c48c48ff01594064c84fa',
 'model_name': 'VotingEnsemble',
 'outputs': [0],
 'model_version': 'v1'}
```

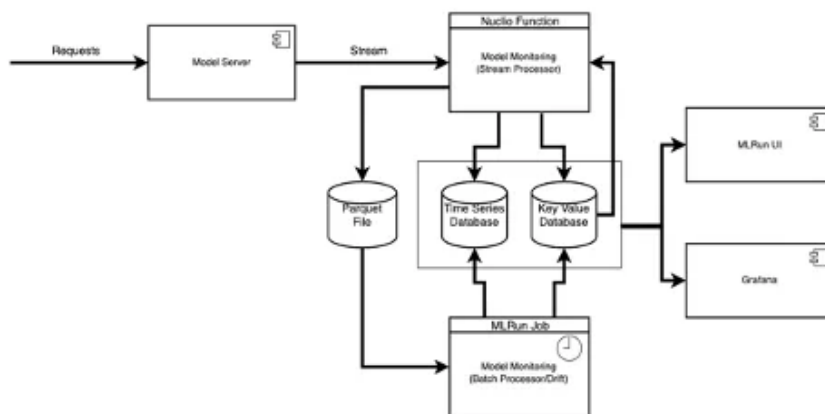
Not only do we receive the desired prediction and model information, we also see that the fully enriched record is printed out three times. This is due to the `print` statement in our `predict` function in `azure_serving.py` being called for each of our three models.

Now that we have verified that our model ensemble is working as expected in our Jupyter environment, we need to deploy it on our Kubernetes cluster.

Enable Model Monitoring

In addition to the deployment itself, we want to enable model monitoring and drift detection on our serving function. There are many excellent resources available on why model monitoring is important as well as how to build a drift-aware ML system. This is not a trivial task, especially for online drift calculations using real-time production data.

Luckily for us, this is available out of the box using the Iguazio platform. Behind the scenes, Iguazio will be deploying a real-time stream processor as well as an hourly scheduled batch job to calculate drift. The architecture looks something like this:



However, for our purposes, we don't need to worry about any of that. We can simply enable model monitoring for our serving function with the following snippet:

```
serving_fn.set_tracking()
project.set_model_monitoring_credentials(os.getenv('V3IO_ACCESS_KEY'))
```

Deploy on Kubernetes Cluster

Finally comes the moment of truth - production deployment on Kubernetes. In many cases, this is where projects go to die. In fact, some estimate that 85% of AI projects fail moving between development and production.

Not for us. Because we are using MLRun for containerization/configuration/deployment, we are almost done with our journey. To do the final deployment, we run this single line:

```
serving_fn.deploy()
```

```

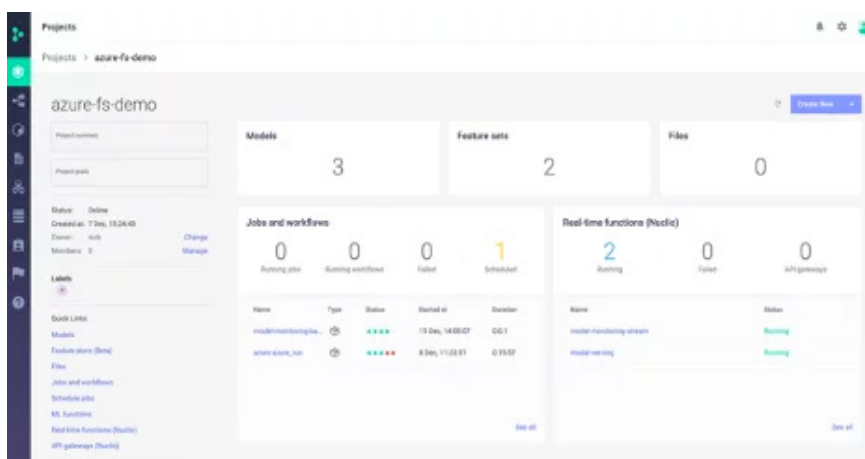
> 2021-12-15 20:33:41,122 [info] Starting remote function deploy
2021-12-15 20:33:42 (info) Deploying function
2021-12-15 20:33:42 (info) Building
2021-12-15 20:33:43 (info) Staging files and preparing base images
2021-12-15 20:33:43 (info) Building processor image
2021-12-15 20:33:44 (info) Build complete
2021-12-15 20:33:51 (info) Function deploy complete

> 2021-12-15 20:33:52,276 [info] successfully deployed function: {'internal_invocation_
urls': ['nuclio-azure-fs-demo-model-serving.default-tenant.svc.cluster.local:8080'], 'external_
invocation_urls': ['azure-fs-demo-model-serving-azure-fs-demo.default-tenant.app.XXXXXXX.XXXXXXX.
com/']}

```

This will build a new Docker image using the base image/requirements we specified, followed by deploying our serving graph to a real-time Nuclio function. Additionally, it will automatically deploy the stream/batch processor for model monitoring purposes.

We can see these changes displayed in the MLRun project UI:



We can test our newly deployed function using the default HTTP trigger like so:

```

serving_fn.invoke(
    path='/v2/models/infer',
    body={
        'inputs': [
            ["d107db82-fe26-4c02-b264-a3749510ed9b"],
            ["4d05f307-b699-4dbe-b51d-f14627233e5a"],
            ["43f23da3-99d0-4630-9831-91d7b54e757e"],

```

```

        ["e031ed66-52f8-4f49-9881-aeff00be2be1"],
        ["31ff724d-b29b-4edb-9f70-f4da66902fe2"]
    ]
}
)

> 2021-12-15 20:33:56,189 [info] invoking function: {'method': 'POST', 'path': '<http://
nuclio-azure-fs-demo-model-serving.default-tenant.svc.cluster.local:8080/v2/models/infer>'}
{'id': 'f178c7da-0dd7-4d1b-8049-5328231229d7',
 'model_name': 'VotingEnsemble',
 'outputs': [0, 1, 1, 0, 1],
 'model_version': 'v1'}

```

The default invocation trigger is via HTTP, but we can also add additional triggers like Cron, Kafka, Kinesis, Azure Event Hub, and more.

Simulate Production Traffic

Before viewing our model monitoring dashboards, we need something to monitor. We are going to simulate production traffic by continuously invoking our function using the `'patient_ids'` from our `'FeatureVector'`.

We can retrieve a list of the `'patient_ids'` like so:

```

import mlrun.feature_store as fstore

records = fstore.get_offline_features(
    feature_vector="azure-fs-demo/heart_disease_vec",
    with_indexes=True
).to_dataframe().index.to_list()

```

Then, we can invoke our serving function with random records and delays like so:

```

from random import choice, uniform
from time import sleep

for _ in range(4000):
    data_point = choice(records)
    try:
        resp = serving_fn.invoke(path='/v2/models/predict', body={'inputs': [[data_point]]})
        print(resp)
        sleep(uniform(0.2, 1.7))
    except:
        pass

```

```

except OSError:
    pass

> 2021-12-15 21:37:40,469 [info] invoking function: {'method': 'POST', 'path': '<http://
nuclio-azure-fs-demo-model-serving.default-tenant.svc.cluster.local:8080/v2/models/predict>'}
    {'id': '65e2c781-31e5-46de-8e1d-2254d7c1b8af', 'model_name': 'VotingEnsemble', 'outputs':
[1], 'model_version': 'v1'}

> 2021-12-15 21:37:41,707 [info] invoking function: {'method': 'POST', 'path': '<http://
nuclio-azure-fs-demo-model-serving.default-tenant.svc.cluster.local:8080/v2/models/predict>'}
    {'id': 'da267484-3b00-4752-87a9-157e2e3ca31c', 'model_name': 'VotingEnsemble', 'outputs':
[1], 'model_version': 'v1'}

> 2021-12-15 21:37:43,411 [info] invoking function: {'method': 'POST', 'path': '<http://
nuclio-azure-fs-demo-model-serving.default-tenant.svc.cluster.local:8080/v2/models/predict>'}
    {'id': '4502196f-b9bb-45ea-a909-22670aa4cc58', 'model_name': 'VotingEnsemble', 'outputs':
[0], 'model_version': 'v1'}
...

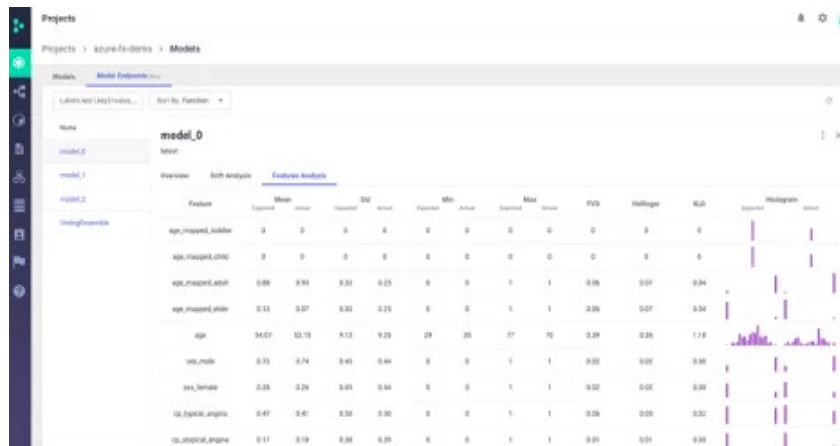
```

View Monitoring Dashboards

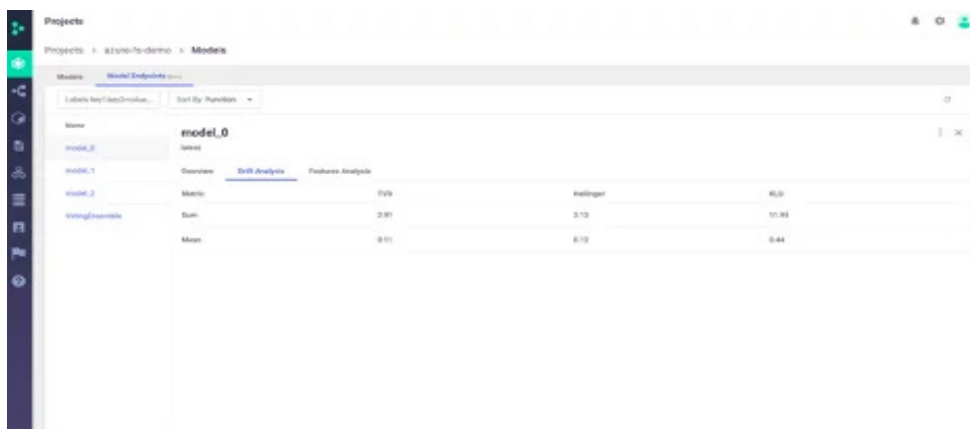
Once our simulated traffic has run for a bit, we can check our model monitoring dashboards.

The first dashboard is high level drift information in the MLRun Project UI.

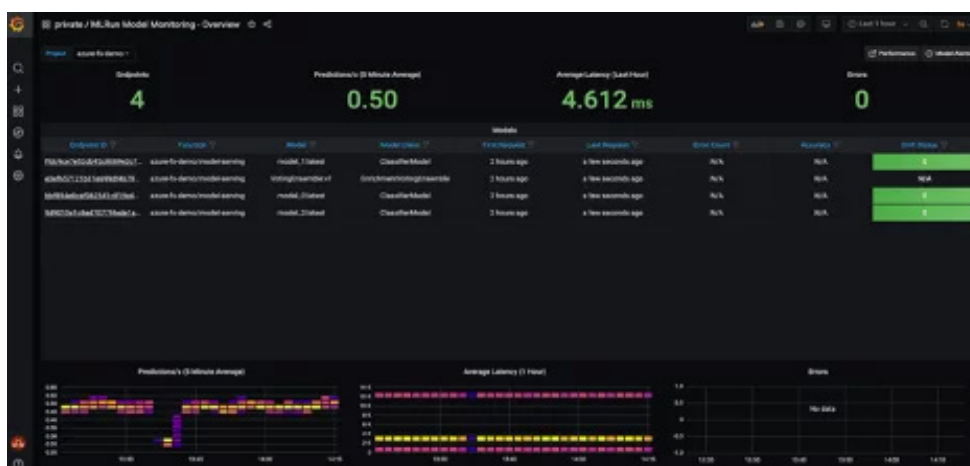
Here we can see statistics, histogram, and drift metrics per feature:



We can also view overall model drift analysis:



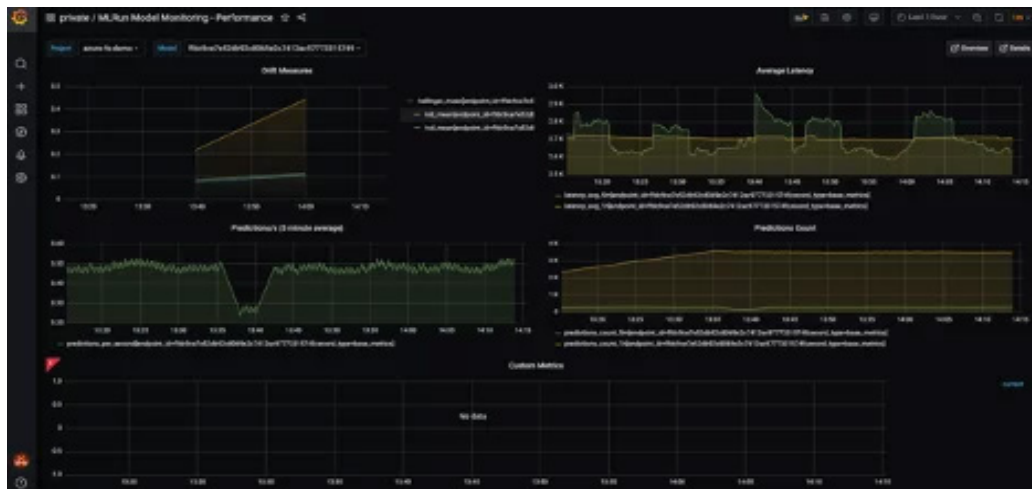
The second dashboard is more in depth drift and inferencing information in a Grafana dashboard. Here we can see drift per model, average number of predictions, and average latency per project:



We can also view a graph of incoming features over time. This is one of my personal favorite dashboards as it lets you visualize how your incoming features change over time:



Finally, you can see model performance, latency, drift metrics over time, and any custom metrics you are logging:



All of these dashboards and visualizations come out of the box with little to no additional configuration required.

If and when model drift is detected, an event with the drift information will be posted to a stream. From there, we can add consumers to the stream to perform tasks like sending notifications or kicking off a re-training pipeline.

Closing Thoughts

In this e-book, we have accomplished quite a lot. Using a combination of the Iguazio MLOps platform and Azure's AutoML capabilities, we have created a full end-to-end training flow that does the following:

- Ingest and transform dataset into feature store
- Upload/register features from Iguazio feature store into Azure ML
- Orchestrate Azure AutoML training job from Iguazio platform
- Download trained model(s) + metadata from Azure back into Iguazio platform
- Deploy top 3 models in model ensemble to real-time HTTP endpoint
- Integrate model serving with real-time enrichment and imputation via feature store
- Integrate model serving with model-monitoring and drift-detection

This whole flow is reproducible and modular in nature - meaning it would be simple to add additional steps, datasets, models, etc.

Want to play around with the code examples in this document?

[Check out the repo here](#)

Want to try out the Iguazio MLOps Platform?

[Start Your 14-day Free Trial](#)