

Operationalizing Machine Learning

Overview

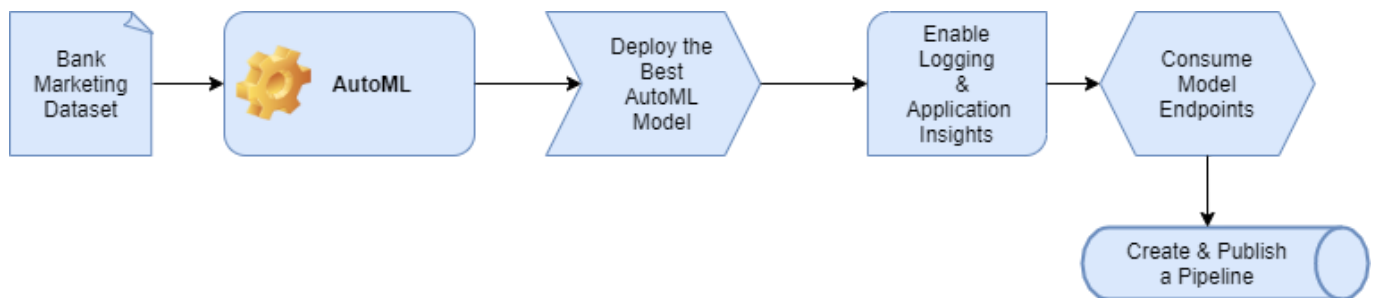
This project is part of the Udacity Azure ML Nanodegree.

In this project, we use Azure to **configure a cloud-based machine learning production model, deploy it, and consume it**. We will also create, publish, and consume a pipeline.

Architectural Diagram

In this project, we explain briefly how to publish your best AutoML model and deploy it as a Web API.

- **Experiment Run:** Using MLStudio we manually upload data, select the compute target and the task we want to accomplish
- **Best Model Selection:** Comparing all the models, finally we choose the one with better primary metric, in this case, weighted AUC, according to the target unbalance.
- **Model Deployment:** The best model is deployed as an endpoint.
- **Application Insights Activation:** We enable this logs monitoring tool.
- **Display Swagger Documentation:** We make use of the *swagger.json* file given by Azure to visualize documentation in a more clear and professional way.
- **Consume Endpoint:** We do a test and a benchmark to check out that the endpoint works and to better know latency times.
- **Create, Publish and Consume a Pipeline using Python SDK:** In the top of Automation we have the pipelines. Processing data and retraining only with an HTTP request is possible.



Key Steps

1. Data Preparation: When AutoML run is created, *Bank-Marketing* data is uploaded and registered so that we can use it in our experiments.

Given the *bank-marketing.csv* file, we upload it to Datasets section so that we can do a quick exploration before using it. In addition to that, in "Consume" tab Azure gives us an example of python code chunk to retrieve data and dive into it using pandas library.

One of the most important keys before thinking about models is to be familiar with the data available for modeling and the distribution of the target variable. In this case, the target feature is called "y" and has a particular characteristic: distribution of yes/no is not uniform, i.e., we have much more no than yes so that we say that we have high unbalancement. There are many ways to solve this problem (resampling, choosing a

right metric, ...) and none of them is better than other. We have to choose accordingly to the task we are accomplishing.

Microsoft Azure Machine Learning

quick-starts-ws-138938 > Datasets > Bank-marketing

Bank-marketing Version 1 (latest)

Profile This is the quick profile generated by the top 10,000 rows. Please generate profile to view the schema and summary statistics for full data.

Details Consume Explore Models

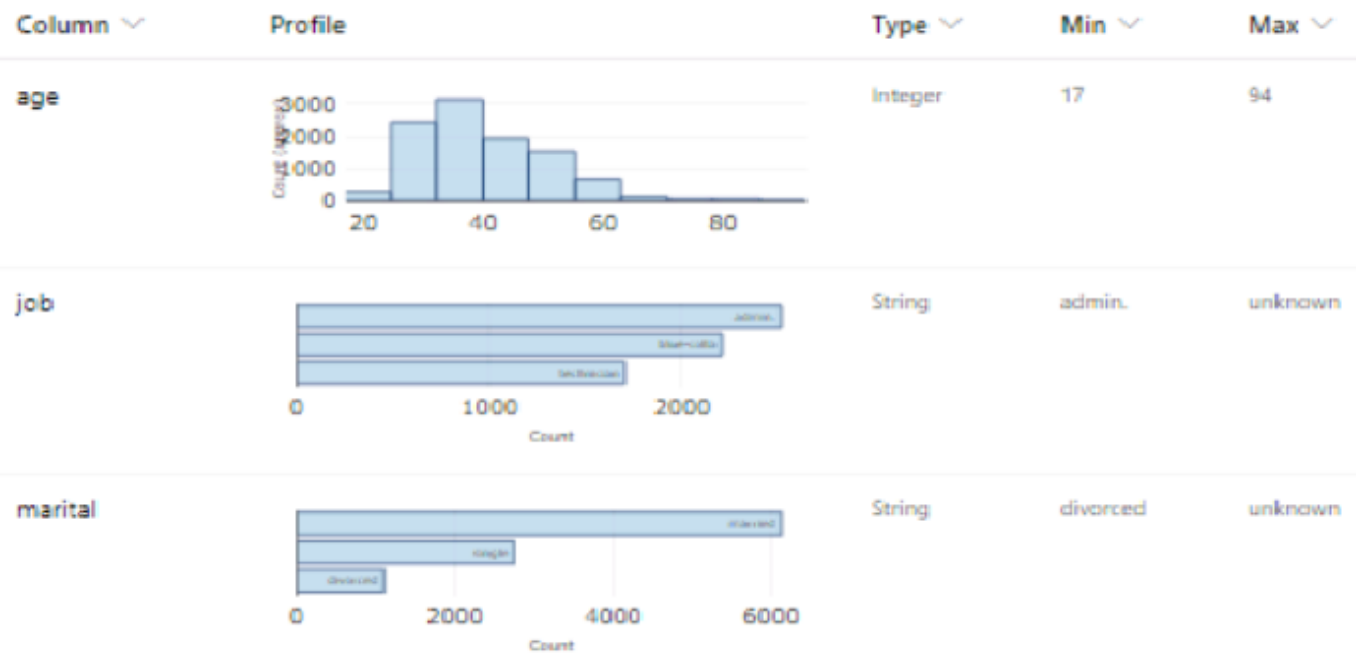
Refresh Generate profile Unregister New version

Preview Profile

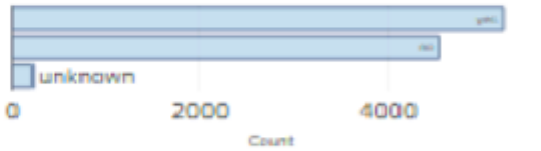

Number of columns: 21 Number of rows: 50 (of 10000)

| id | age | job | marital | education | default | housing | loan | contact | month | day_of_week | duration | campaign |
|----|-----|-------------|---------|---------------------|---------|---------|------|-----------|-------|-------------|----------|----------|
| 1 | 57 | technician | married | high.school | no | no | yes | cellular | may | mon | 371 | 1 |
| 2 | 55 | unknown | married | unknown | unknown | yes | no | telephone | may | thu | 285 | 2 |
| 3 | 33 | blue-collar | married | basic.9y | no | no | no | cellular | may | fri | 52 | 1 |
| 4 | 36 | admin. | married | high.school | no | no | no | telephone | jun | fri | 355 | 4 |
| 5 | 27 | housemaid | married | high.school | no | yes | no | cellular | jul | fri | 189 | 2 |
| 6 | 58 | retired | married | professional.course | no | yes | yes | cellular | jul | fri | 605 | 1 |
| 7 | 48 | services | married | high.school | unknown | yes | no | telephone | may | wed | 243 | 1 |

Personal information: Occupation, age, marital status and education.



Financial information: Debts and other data about customer's financial health.

| Column | Profile | Type | Min | Max |
|---------|---|--------|-----|-----|
| default |  | String | no | yes |
| housing |  | String | no | yes |
| loan |  | String | no | yes |

Target: The variable we seek to predict is the one that tells us if a given person is a potential client or not.

| Column | Profile | Type | Min | Max | Count |
|-------------|---|---------|---------|---------|-------|
| euribor3m |  | Decimal | 0.63 | 5.04 | 10000 |
| nr.employed |  | Decimal | 4963.60 | 5228.10 | 10000 |
| y |  | String | no | yes | 10000 |

2. Experiment Run: AutoML experiment correctly run and submitted.

Once we have some sensitivity about data, we are going to train a classifier using an ML model. Modeling is an art but we have some interesting tools that help us to obtain an acceptable ML model without doing much feature engineering. This is the case of AutoML models. These tools work by comparing different candidate algorithms and hyperparameters so that computational cost is high.

Azure AutoML module is optimised to do this tasks in a more efficient way than if a normal person develops by himself. To this fact, we add the possibility of using powerful servers on demand to run this tasks so the more yo need (and pay) the more you have. These servers where AutoML runs are called Computer Instances (or compute targets).

This told, to run an AutoML experiment using Azure Portal first we go to AutoML module in the sidebar and name the **new experiment** as *p2-ml_ops_udacity*. Then, a *Standard_DS12_v2* **compute cluster is configured** with minimum number of nodes equal to 1 and maximum of 6. We call it *p2-azure compute*.

Finally, a *Classification* task is selected. On *Exit Criterion* we reduce the default (3 hours) to 1, because we think it is enough to get a good model for this particular task, and reduce the *Concurrency* from default to 5 (this number should always be less than the number of the compute clusters).



Figure 1: Successful AutoML experiment run on p2-azure-compute cluster

3. Best Model: The best performing model is the one using *VotingEnsemble*.

After the experiment run completes, a summary of all the models and their metrics are shown, including explanations. The Best Model will be shown in the Details tab. In the Models tab, it will come up first (at the top). Make sure you select the best model for deployment.

Deploying the Best Model will allow to interact with the HTTP API service and interact with the model by sending data through POST requests.

| Algorithm name | Explained | Accuracy ↓ | Sampling ⓘ | Run | Created | Duration | Status |
|-------------------------------------|-----------|------------|------------|------------------------|----------------------|----------|-----------|
| VotingEnsemble | | 0.91988 | 100.00 % | Run 82 | Feb 15, 2021 8:27 AM | 1m 9s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91563 | 100.00 % | Run 51 | Feb 15, 2021 8:20 AM | 1m 3s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91502 | 100.00 % | Run 65 | Feb 15, 2021 8:23 AM | 1m 1s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91502 | 100.00 % | Run 62 | Feb 15, 2021 8:23 AM | 57s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91472 | 100.00 % | Run 52 | Feb 15, 2021 8:20 AM | 56s | Completed |
| MaxAbsScaler, LightGBM | | 0.91411 | 100.00 % | Run 74 | Feb 15, 2021 8:25 AM | 1m 14s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91381 | 100.00 % | Run 68 | Feb 15, 2021 8:24 AM | 53s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91351 | 100.00 % | Run 38 | Feb 15, 2021 8:18 AM | 1m 5s | Completed |

Figure 2: List of Models sorted by Primary Metric performance

In Figure 3, best model performance is shown in *Metrics* tab. Although Accuracy was used as a Primary Metric, target unbalancing suggests us to pay moer attention to macro-metrics.

- Micro-averaged: all samples equally contribute to the final averaged metric
- Macro-averaged: all classes equally contribute to the final averaged metric
- Weighted-averaged: each classes's contribution to the average is weighted by its size

You can find an interesting example [here](#).



We choose the best model for deployment and enable "Authentication" while deploying the model using Azure Container Instance (ACI). The executed code in logs.py enables Application Insights. "Application Insights enabled" is disabled before executing logs.py.

4.1. Activate Application Insights:

Before running any python scripts, it is important to create a python virtual environment with `virtualenv venv`. Once activated the virtual environment, we can edit and run `logs.py` writing the endpoint name `demo-model-deploy` and see the console output-

Figure 4: logs.py script output

5 / 13

demo-model-deploy

Details Test Consume Deployment logs

Model ID
AutoML88152109066:1

Created on
2/21/2021 6:02:58 PM

Last updated on
2/21/2021 6:39:07 PM

Image ID
--

REST endpoint
<http://66f1bb41-2922-4b94-ae09-fab5a70c7d3e.southcentralus.azurecontainer.io/score>

Key-based authentication enabled
true

Swagger URI
<http://66f1bb41-2922-4b94-ae09-fab5a70c7d3e.southcentralus.azurecontainer.io/swagger.json>

CPU
1.8

Memory
4 GB

Application Insights enabled
true

Application Insights url
<https://portal.azure.com#resource/subscriptions/81cefad3-d2c9-4f77-a466-99a7f541c7bb/resourcegroups/aml-quickstarts-139278/providers/microsoft.insights/components/mlappinsight139278>

Figure 5: Proof of Applications Insights activation

The activity report displays some graphs showing status, failed requests and response time during the periods which the endpoint is working.

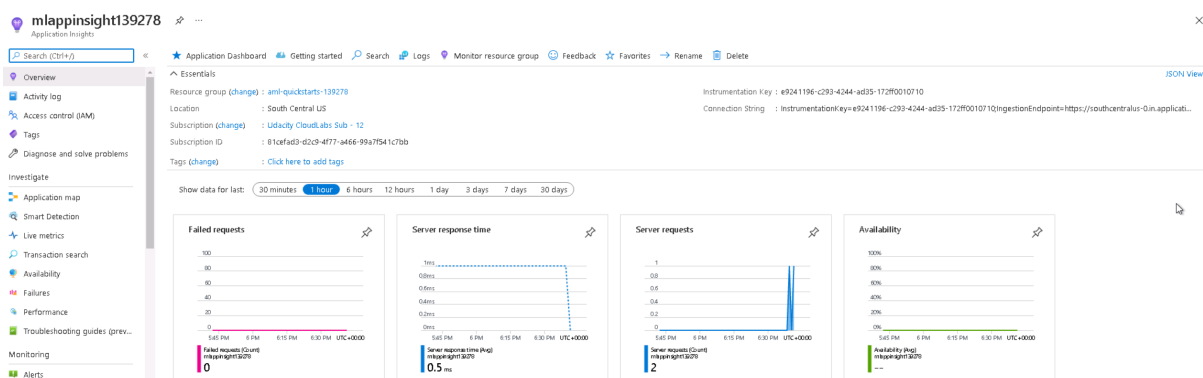


Figure 6: Endpoint activity report

5. Display Swagger Documentation

If we go back to the endpoint Details tab, a URL to a swagger.json file can be found. We download it into ./swagger/swagger.json file. This is going to be used to format API documentation in Swagger UI app.

In swagger.sh script, **swagger-ui** docker image is downloaded so that we can start containers on it. We map local port 9000 to the 8080 in the container, where swagger is running. This way, Swagger UI app is running on <http://localhost:9000/>.

For CORS activation, we have to run server.py in 8000 port and after that check <http://localhost:8000/swagger.json> in **Swagger** search bar.

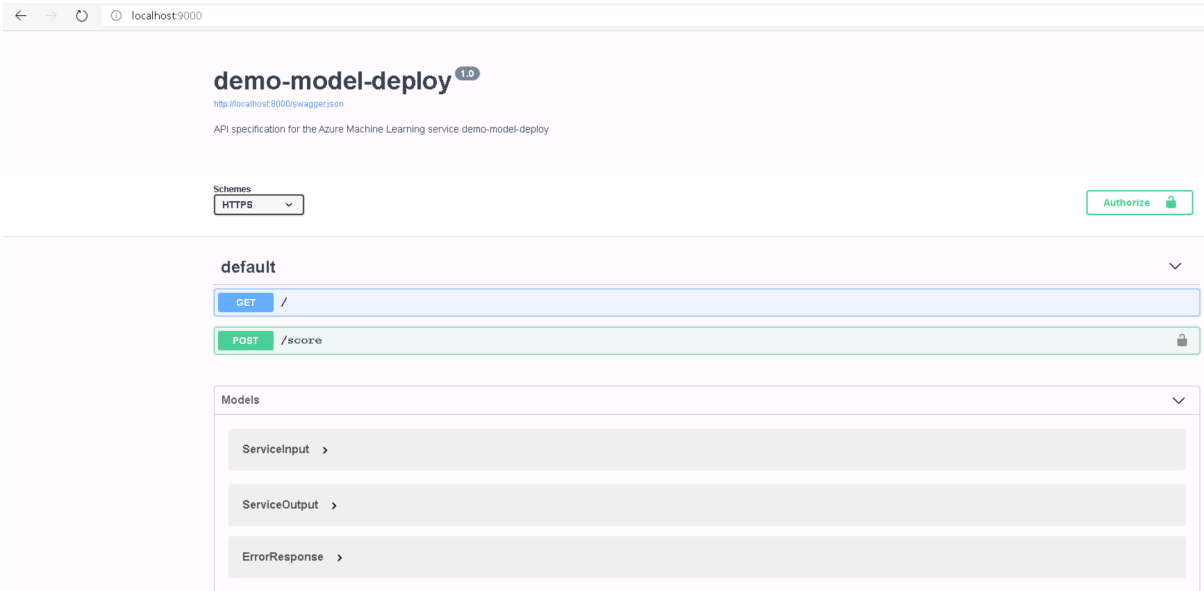


Figure 7: Swagger documentation for demo-model-deploy

6. Consume Endpoint and benchmarking

In software development, everything has to be tested and monitorized. To test our endpoint does correctly its work, we run endpoint.py to send a test POST request to the API. To do that, we create data.json example file from a python dictionary hardcoded into endpoint.py and send it using python requests library.

```

GNU nano 4.9.3
import requests
import json

# URL for the web service, should be similar to:
# 'http://6530a665-66f3-49c8-a953-b82a2d312917.eastus.azurecontainer.io/score'
scoring_uri = 'http://66f1bb41-2922-4b94-ae09-fab5a70c7d3e.southcentralus.azurecontainer.io/score'
# If the service is authenticated, set the key or token
key = 'UQTwrNMHnpVlcGdw0wVx0wDjNYId9jfw8'

# Two sets of data to score, so we get two results back
data = {"data":
    [
        {
            "age": 17,
            "campaign": 1,
            "cons.conf.idx": -46.2,
            "cons.price.idx": 92.893,
            "contact": "cellular",
            "day_of_week": "mon",
            "default": "no",
            "duration": 971,
            "education": "university.degree",
            "emp.var.rate": -1.8,
            "euribor3m": 1.299,
            "housing": "yes",
            "job": "blue-collar",
            "loan": "yes",
            "marital": "married",
            "month": "may",
            "nr.employed": 5099.1,
            "pdays": 999,
            "poutcome": "failure",
            "previous": 1
        },
        {
            "age": 87,
            "campaign": 1,
            "cons.conf.idx": -46.2,
            "cons.price.idx": 92.893,
            "contact": "cellular",
            "day_of_week": "mon",
            "default": "no",
            "duration": 471,
            "education": "university.degree",
            "emp.var.rate": -1.8,
            "euribor3m": 1.299,
            "housing": "yes",
            "job": "blue-collar",
            "loan": "yes",
            "marital": "married",
            "month": "may",
            "nr.employed": 5099.1,
            "pdays": 999,
            "poutcome": "failure",
            "previous": 1
        }
    ]
}

# Convert to JSON string
input_data = json.dumps(data)
with open("data.json", "w") as _f:
    _f.write(input_data)

# Set the content type
headers = {'Content-Type': 'application/json'}
# If authentication is enabled, set the authorization header
headers['Authorization'] = f'Bearer {key}'

# Make the request and display the response
resp = requests.post(scoring_uri, input_data, headers=headers)
print(resp.json())

```

Figure 8: Endpoint test request

The obtained output is the class corresponding to the 2 samples sent.


```
PS C:\Users\demouser\Desktop\nd00333_AZMLND_C2-master\starter_files> python .\endpoint.py
{"result": ["yes", "no"]}
```

Figure 9: Endpoint response for test samples

Now we want to know the average time needed by the endpoint to process requests and send responses. For that we use **Apache Benchmark**, a tool for benchmarking your Apache Hypertext Transfer Protocol (HTTP) server. It is designed to give you an impression of how your current Apache installation performs. This especially shows you how many requests per second your Apache installation is capable of serving. More info [here](#).

First of all we check that it is installed displaying the help `ab -h`. Once we see that it works, Authorization key and URI can be introduced into `benchmark.sh` and run the script.

```
GNU nano 4.9.3 benchmark.sh
# HTTP/1.0 200 OK
# Content-Length: 33
# Content-Type: application/json
# Date: Thu, 30 Jul 2020 12:33:34 GMT
# Server: nginx/1.10.3 (Ubuntu)
# X-MS-Request-Id: b48d88da-0b4e-44fd-ae5-04043bfa77f1
# X-MS-Run-Function-Failed: False
# [{"result": ["yes", "no"]}]"
# LOG: Response code = 200
# ..done
#
# Server Software:      nginx/1.10.3
# Server Hostname:      8530a665-66f3-49c8-a953-b82a2d312917.eastus.azurecontainer.io
# Server Port:          80
#
# Document Path:        /score
# Document Length:      33 bytes
#
# Concurrency Level:    1
# Time taken for tests:  1.599 seconds
# Complete requests:    10
# Failed requests:      0
# Total transferred:    2600 bytes
# Total body sent:      10560
# HTML transferred:     330 bytes
# Requests per second:  6.25 [#/sec] (mean)
# Time per request:     159.918 [ms] (mean)
# Time per request:     159.918 [ms] (mean, across all concurrent requests)
# Transfer rate:         6.45 [Kbytes/sec] received
#                        8.04 kb/s total
#
# Connection Times (ms)
#   min  mean[+/-sd] median  max
# Connect:    21   23   0.8    23   24
# Processing: 92  137  28.3   151  176
# Waiting:    92  137  28.3   151  176
# Total:     114  160  28.0   172  199#
ab -n 10 -v 4 -p data.json -T 'application/json' -H 'Authorization: Bearer UQTwrNWhnpVlcGdwwxYxMdJNYId9jfw8' http://66f1bb41-2922-4b94-ae09-fab5a70c7d3e.southcentralus.azurecontainer.io/score
```

Figure 10: benchmark.sh script content

```

Server Software:      nginx/1.10.3
Server Hostname:      66f1bb41-2922-4b94-ae09-fab5a70c7d3e.southcentralus.azurecontainer.io
Server Port:          80

Document Path:        /score
Document Length:       33 bytes

Concurrency Level:     1
Time taken for tests:   2.269 seconds
Complete requests:     10
Failed requests:        0
Total transferred:     2600 bytes
Total body sent:       10640
HTML transferred:      330 bytes
Requests per second:   4.41 [#/sec] (mean)
Time per request:      226.853 [ms] (mean)
Time per request:      226.853 [ms] (mean, across all concurrent requests)
Transfer rate:         1.12 [Kbytes/sec] received
                       4.58 kb/s sent
                       5.70 kb/s total

Connection Times (ms)
              min    mean[+/-sd] median    max
Connect:        2      3    0.5        3      4
Processing:    111    224  46.4       238    270
Waiting:       111    224  46.4       238    270
Total:         114    227  46.6       241    274

Percentage of the requests served within a certain time (ms)
 50%    241
 66%    252
 75%    261
 80%    266
 90%    274
 95%    274
 98%    274
 99%    274
100%    274 (longest request)

demouser@labvm MINGW64 ~/Desktop/nd00333_AZMLND_C2-master/starter_files
$ |

```

Figure 11: Benchmarking results

We can see that, calling 10 times the API with data.json POST (the same used for testing the endpoint), all of them are done without issues in 226 ms in average. This is a very acceptable responding time fro the majority of the real applications purposes.

7. Create, Publish and Consume a Pipeline

Introduction to Pipelines

Pipelines are all about Automation. Automation connects different services and actions together to be part of a new workflow that wasn't possible before.

There are some good examples of how different services can communicate to the pipeline endpoint to enable automation.

- A hosted project using version control: when a new change gets merged, a trigger is created to send an HTTP request to the endpoint and train the model.
- A newer dataset gets uploaded to a storage system that triggers an HTTP request to the endpoint to re-train the model.
- Several teams that want to use AutoML with datasets that are hosted externally can configure the external cloud provider to trigger an HTTP request when a new dataset gets saved.

- A CI /CD platform like Jenkins, with a job that submits an HTTP request to Azure when it completes without error.

Pipeline endpoints can be consumed via HTTP, but it is also possible to do so via the Python SDK. Since there are different ways to interact with published Pipelines, this makes the whole pipeline environment very flexible.

It is key to find and use the correct HTTP endpoint to interact with a published pipeline. Sending a request over HTTP to a pipeline endpoint will require authentication in the request headers.

Process

In this part of the project, we create, publish and consume a Pipeline using Python SDK.

The first step to do it is to upload the Jupyter Notebook that contains the process of the Pipeline creation and change the experiment name so that it's the same used for AutoML `p2-ml_ops_udacity`.

To facilitate Workspace detection via Python SDK, the config.json file with the info about the workspace is downloaded from Azure Portal and uploaded to the notebook folder.

Using Python SDK and the uploaded notebook, we run the AutoML step and publish the Pipeline so that we can find it in MLStudio Pipelines section.

```
1 from azureml.widgets import RunDetails
2 RunDetails(pipeline_run).show()
```

Run 353e8550-331c-4521-973b-227bb9086440 Completed

[2021-02-21 20:34:53Z] Submitting 1 runs, first five are: b50ad284:7985a0dc-9d5d-40dd-92c6-c4bd0f5601e3
[2021-02-21 21:07:48Z] Completing processing run id 7985a0dc-9d5d-40dd-92c6-c4bd0f5601e3.

Run is completed.

Figure 12: Details about AutoML experiment run using Python SDK

quick-starts-ml-139278 > Pipelines

Pipelines

Pipeline runs Pipeline endpoints Pipeline drafts

+ New pipeline Refresh

+ Add filter

| Run | Run ID | Experiment | Status | Description | Submitted time | Duration | Submitted by | Tags |
|--------|-------------------------------------|------------------------|-----------|-------------------------|----------------------|----------|-----------------|--|
| Run 12 | 876f943-80d3-49c7-84d2-183925cd... | pipeline-rest-endpoint | Running | | Feb 21, 2021 9:13 PM | 2m 55s | ODL_User 139278 | azureml.pipelineid : 66990b07-c99a-4921-a49f-... |
| Run 1 | c01526de-b659-4779-b1f6-7c5b4b0c... | pipeline-rest-endpoint | Running | | Feb 21, 2021 9:08 PM | 7m 56s | ODL_User 139278 | azureml.pipelineid : e518dbd4-3e89-40f6-85d0-... |
| Run 74 | 353e8550-331c-4521-973b-227bb90... | p2-ml-ops-udacity | Completed | pipeline_with_automl... | Feb 21, 2021 8:34 PM | 33m 1s | ODL_User 139278 | azureml.pipelineComponent : pipeline-run |

Figure 13: Running pipeline on Machine Learning Studio Pipelines section

When the run is completed, we finally deploy it so that it can be run again as many times as it is needed using HTTP request.

Endpoints

Real-time endpoints Pipeline endpoints

Refresh Disable Enable Include disabled



| Name ↓ | Description | Date updated | Updated by | Last run submit time | Last run status | Status |
|---|------------------|---------------------------|-----------------|---------------------------|-----------------|--------|
|  Bankmarketing Train | Training bank... | February 21, 2021 9:12 PM | ODL_User 139... | February 21, 2021 9:13 PM | Running | Active |
|  Bankmarketing Train | Training bank... | February 21, 2021 9:08 PM | ODL_User 139... | February 21, 2021 9:08 PM | Running | Active |

Figure 14: Available Pipeline Endpoints in ML Studio Endpoints section

If we access to the deployed Pipeline endpoint, we can check that all the steps are included, in this case the data ingestion and the AutoML Step. The URL to call the endpoint is available too. It can be accessed to it using typical HTTP authorized request but, usually, it is preferred to do it using Azure Python SDK for security and to have everything integrated in the same platform.

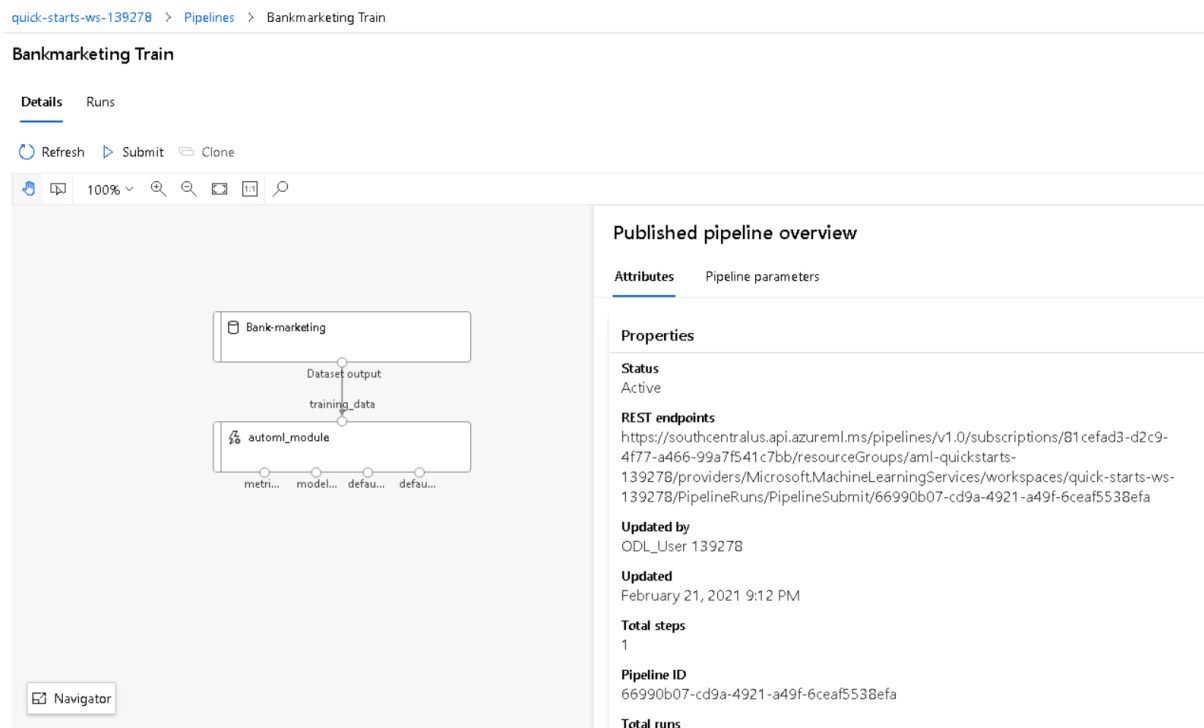
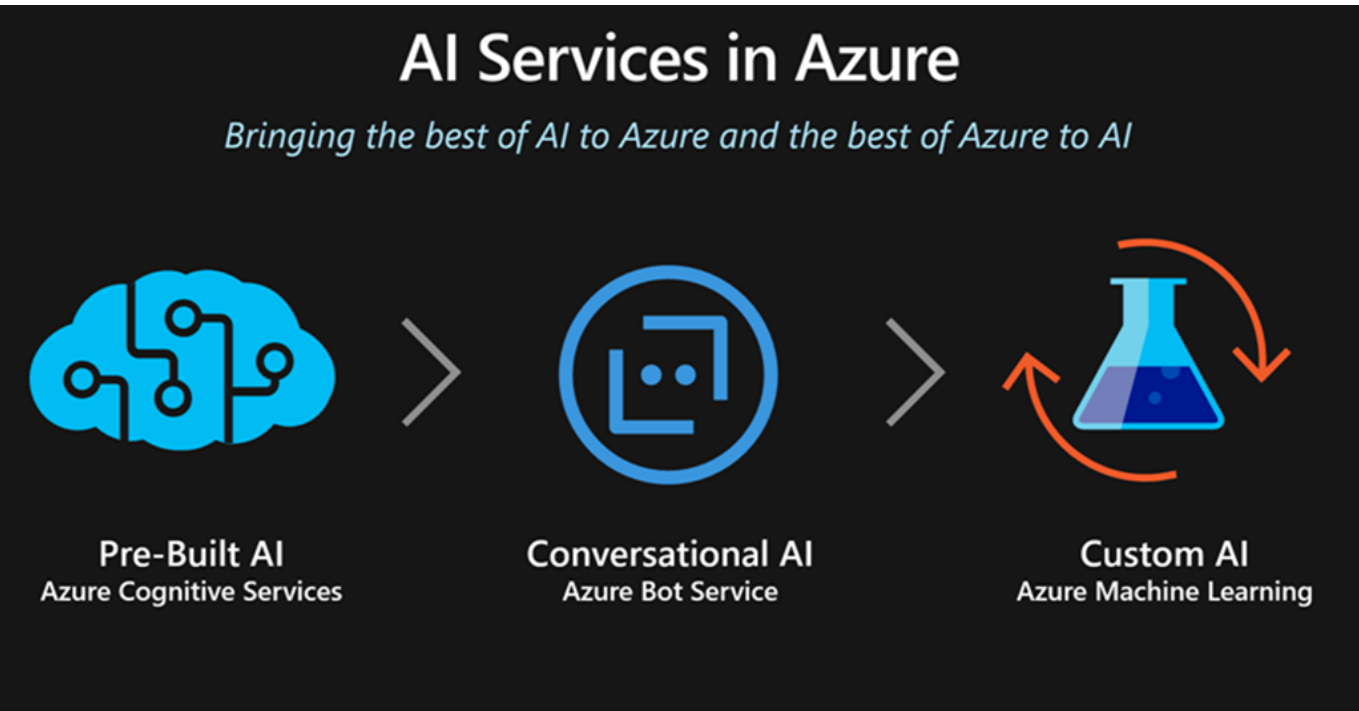


Figure 15: Active Pipeline workflow with endpoint URL

Screen Recording

Clicking on the following picture you can watch a **5 minutes video** with explanations about the content of the project.



Standout Suggestions

For future work, I would suggest making further tweaks to the AutoML step of the pipeline. There are a lot of settings involved and making changes to them could improve the search space and help find an even better model solution. There are also additional steps that could be added to the pipeline, perhaps doing some dataset cleanup or feature engineering before the AutoML step, or doing additional steps after the AutoML step has completed.

In addition, I propose connecting Application Insights to our Pipeline API and Scheduling periodical runs or each time Banking Dataset updates (trigger).