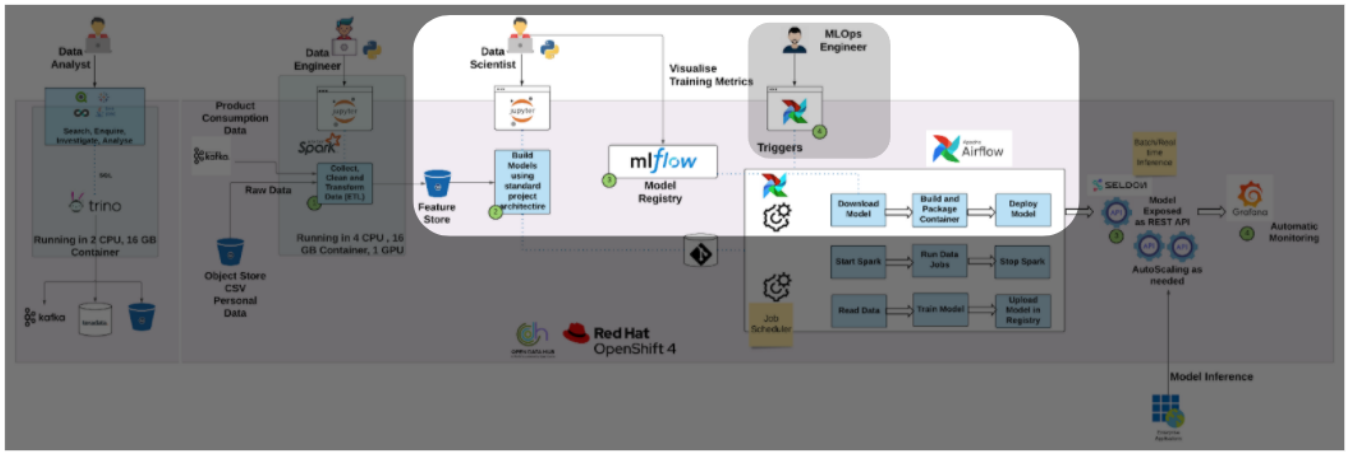
Lab 2 - Data Science

Introduction

Next we feature three Jupyter notebooks the data scientist uses, pulling the prepared CSV data that the data engineer pushed to S3 object storage in the previous lab:

1. They first visualise the data - to understand patterns in the data and whether there are any errors they need to fix before experimenting and training their models.
2. They then experiment with different algorithms, parameters and hyperparameters. They push each experiment to the model repository, Verta. This repository contains all of the data and the actual model binaries should they wish to
    a. Compare different experiments
    b. Return to and retrieve any experiment they ran
    c. Share their experiments with others. In this way, we're allowing silos between different actors in the workflow
3. They then choose one of their experiments, which they wish to proceed with and push to production - in the next part of the workflow, the ML OPs phase

This diagram illustrates the workflow we're implementing - the Data Science part of the overall AI/ML workflow:



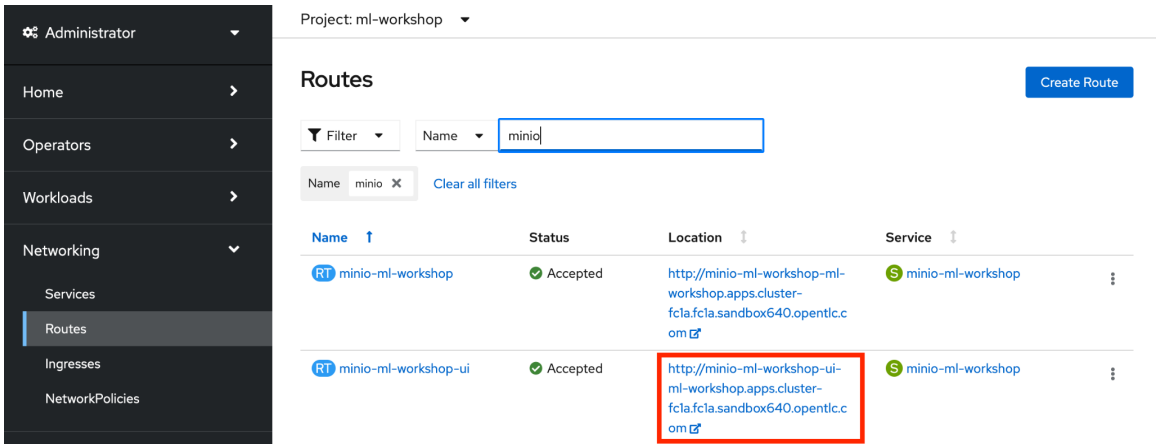
Instructions to access your prepared data file from the previous lab

You need to access the prepared CSV data file you created and pushed to S3 object storage, in the previous lab under the Data Engineer persona.

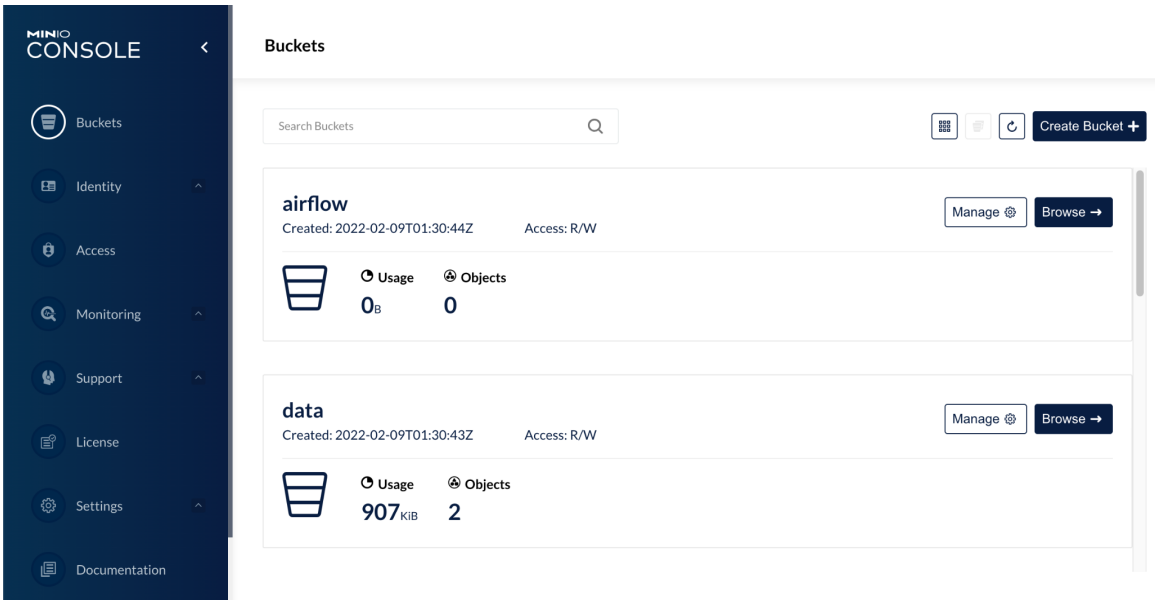
Login to OpenShift using the credentials your administrator gave you. Ensure your workshop project ml-workshop is selected.

The first thing you need to do is retrieve the path and file pertaining to your username - which you as a data engineer created previously.
Choose the **Administration perspective**

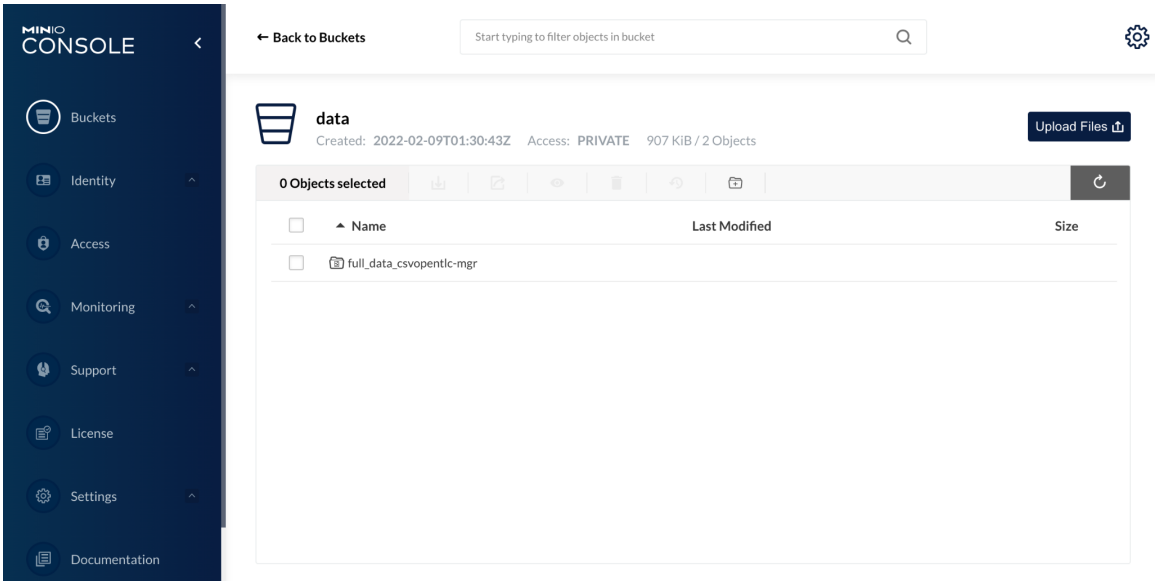
1. Navigate to **Networking > Routes**.
2. Filter on *minio* - and open the *minio-ml-workshop-ui* route as shown.



3. Enter the username and password minio / minio123

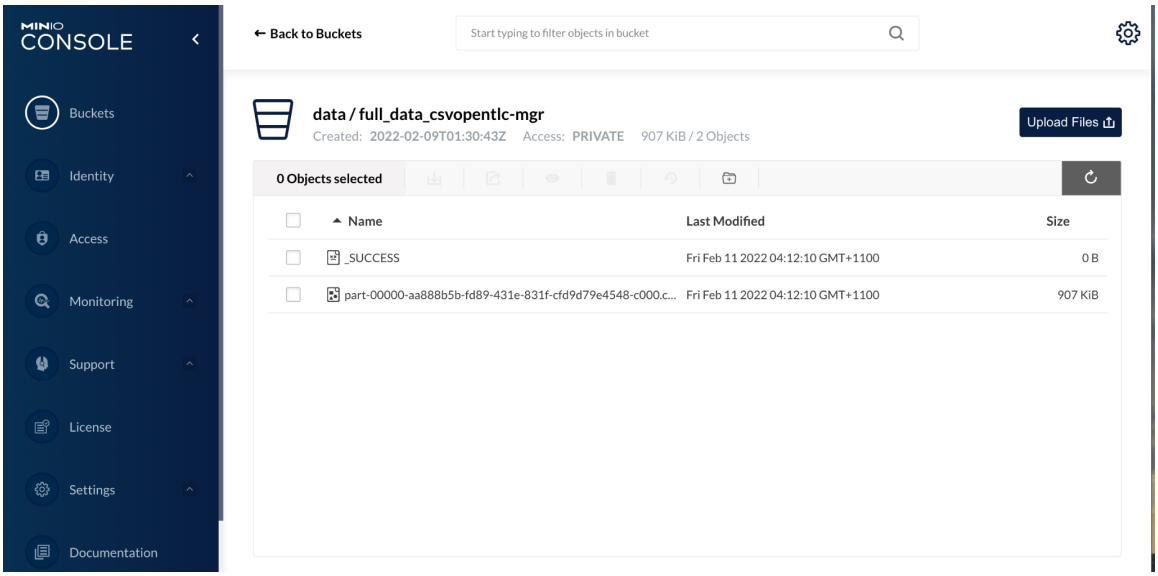


4. Scroll down to the **data** bucket, and click the **Browse**.
   Minio displays a list of folders in the *data* bucket. The folder-name format is: "*full_data_csv-<your username>*" E.g. For user29 the folder is: *full_data_csv-user29*

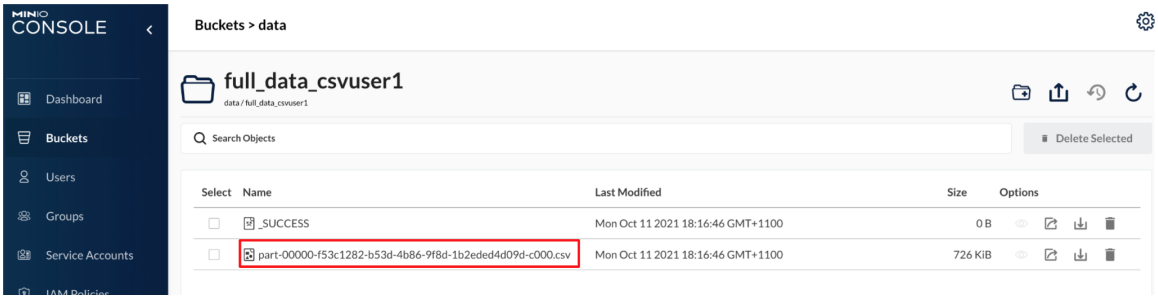


5. Scroll through the list of folders and locate the folder with your username
   Note: There will be many folders there - be sure to identify the one containing **your username.**

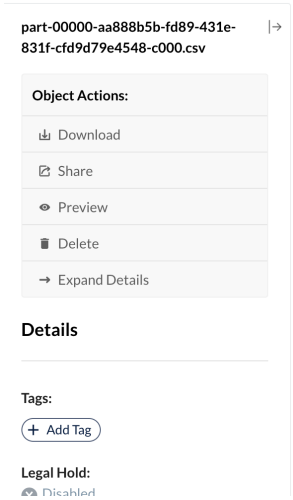
6. Click the folder that corresponds to your username.
   Minio displays the file(s) you created in the Data Engineering lab



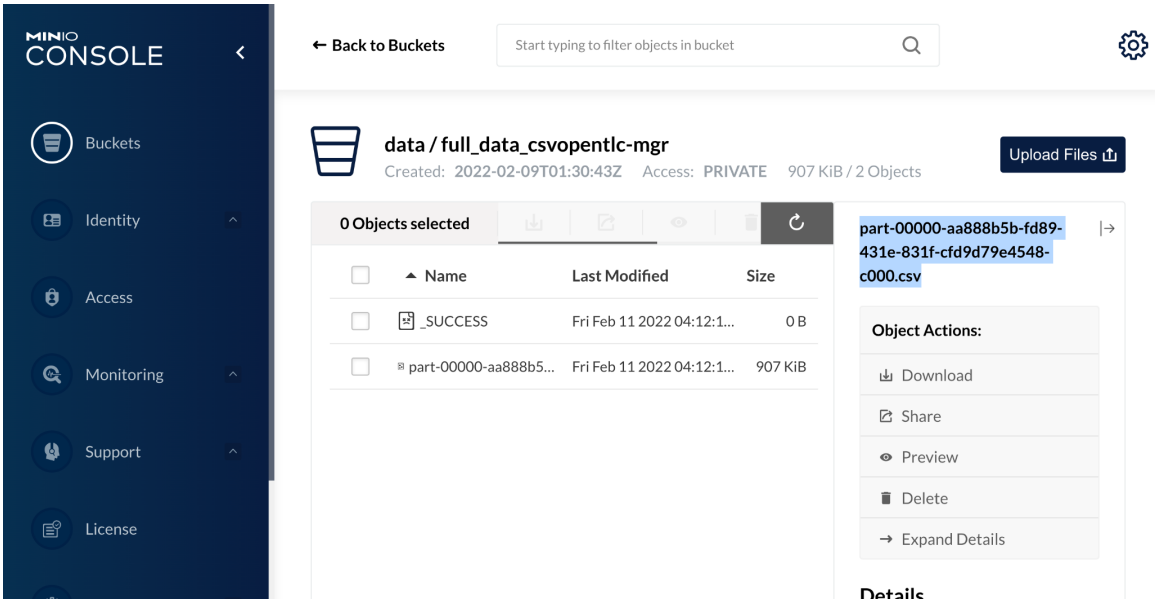
7. Click the file with name starting with "part":



Minio displays a panel containing the details of the file you created in the Data Engineering lab.



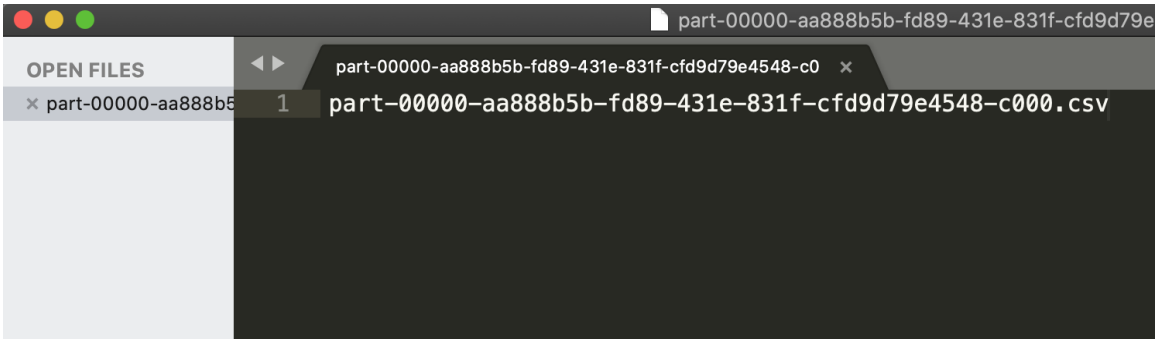
8. Highlight the filename path as shown below and copy the text to the clipboard. E.g. `part-00000-f53c1282-b53d-4b86-9f8d-1b2eded4d09d-c000.csv`



9. Copy this text to the Clipboard (E.g. Ctrl-C or Command-C)
   You need to copy this somewhere safe because we will refer to it throughout this section as **YOUR_CSV_FILE.**

10. Open any editor you choose and paste this into a text editor so you can use it later. E.g. Sublime Text or Microsoft Word.

You file path should now look similar to this:



As mentioned we'll refer to this long filename string as **YOUR_CSV_FILE** - which later you'll paste into the hyper_parameters.py source file**.**

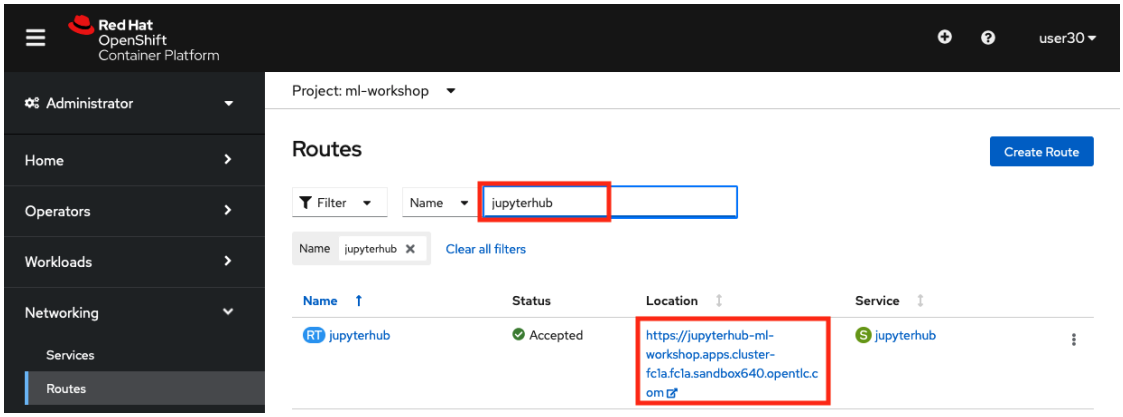
Part 1: Visualise Data

Now to our Data Science focused Jupyter notebooks. As we did with Minio, we will find the url for Jupyterhub within the routes of the OPenShift console.

1. Open the browser tab with the OpenShift console.
2. Open the **Administrator perspective**.
3. Click **Networking > Routes**.
4. Type **Jupyterhub** in the **Filter** text box.

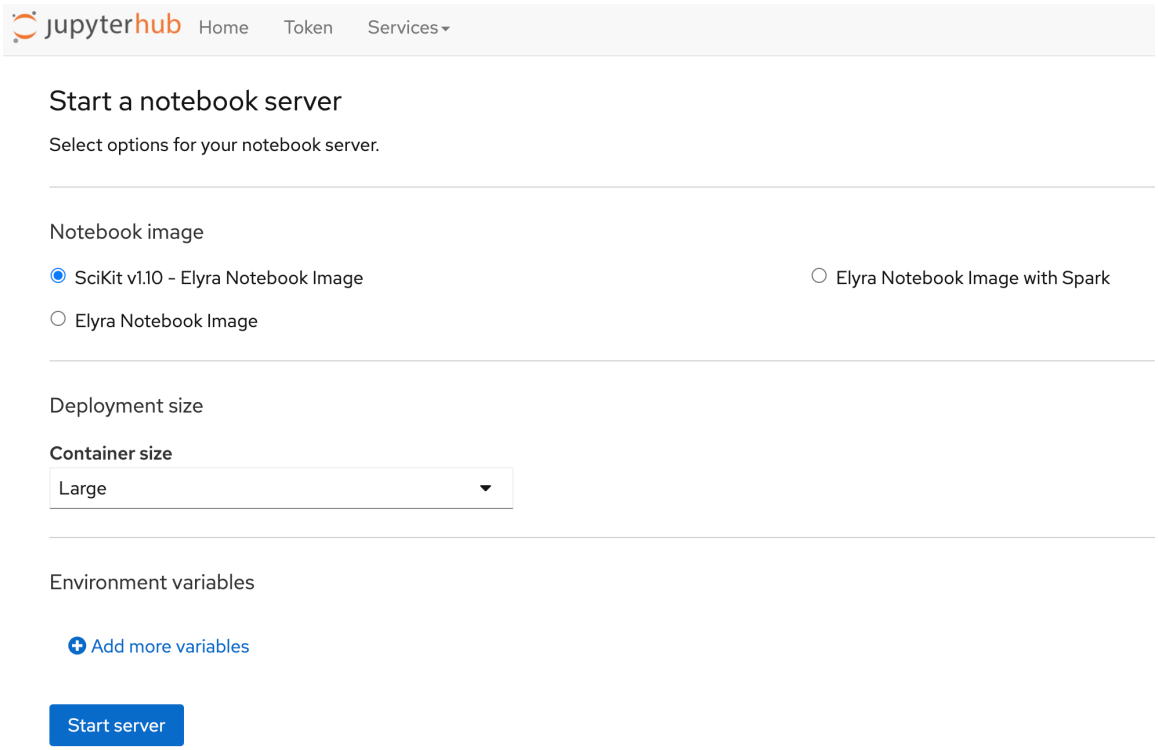
OpenShift reduces the list of routes as you type the filter.

5. Click the Jupyterhub link in the **Location** column of the **Routes** display.



Because you shutdown your Jupyter server at the end of the last workshop, you'll again be presented with the Jupyter screen where you choose the base image to work with. As we're now assuming the role of a data scientist, do the following:

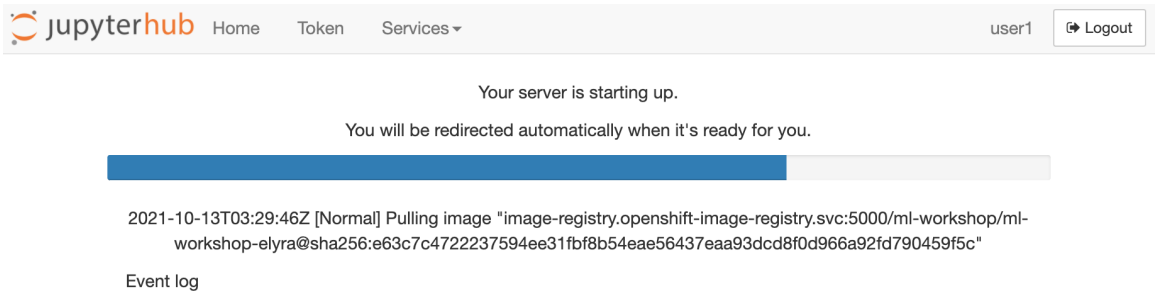
JupyterHub displays the Start Notebook Server page.



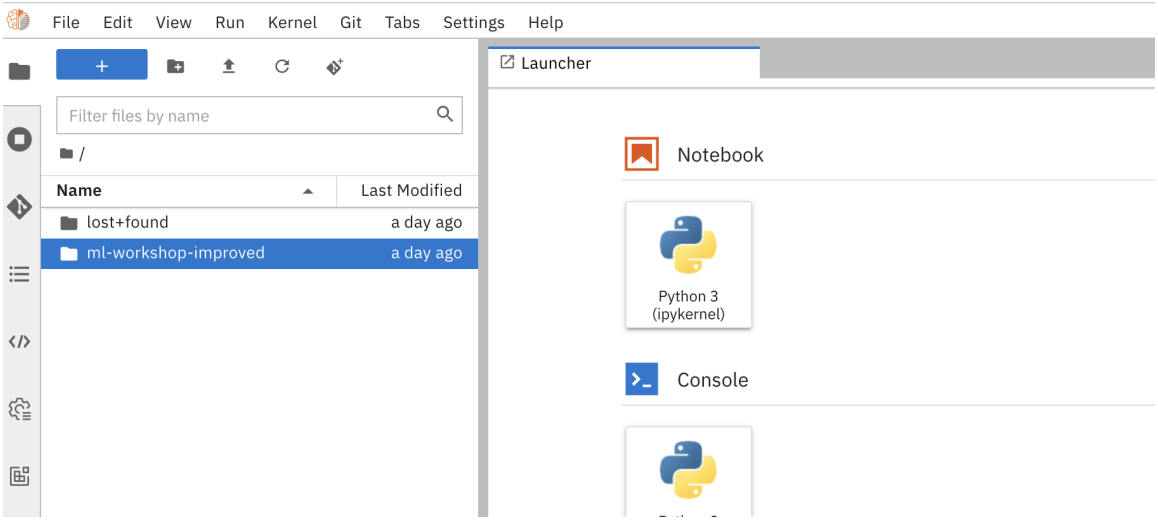
6. Click **SciKit v1.10 - Elyra Notebook Image**
7. Select **Large**
8. Click **Start Server**.

Warning: Please select the correct notebook image, otherwise the lab will not work.

Jupyterhub starts the notebook server for the Data Scientist.



After a few minutes the notebook will have started and the Jupyter notebook will be displayed.



Observe:

- The same *ml-workshop-improved* folder we previously pulled down from our GitHub repository
  git clone https://github.com/bryonbaker/ml-workshop-improved

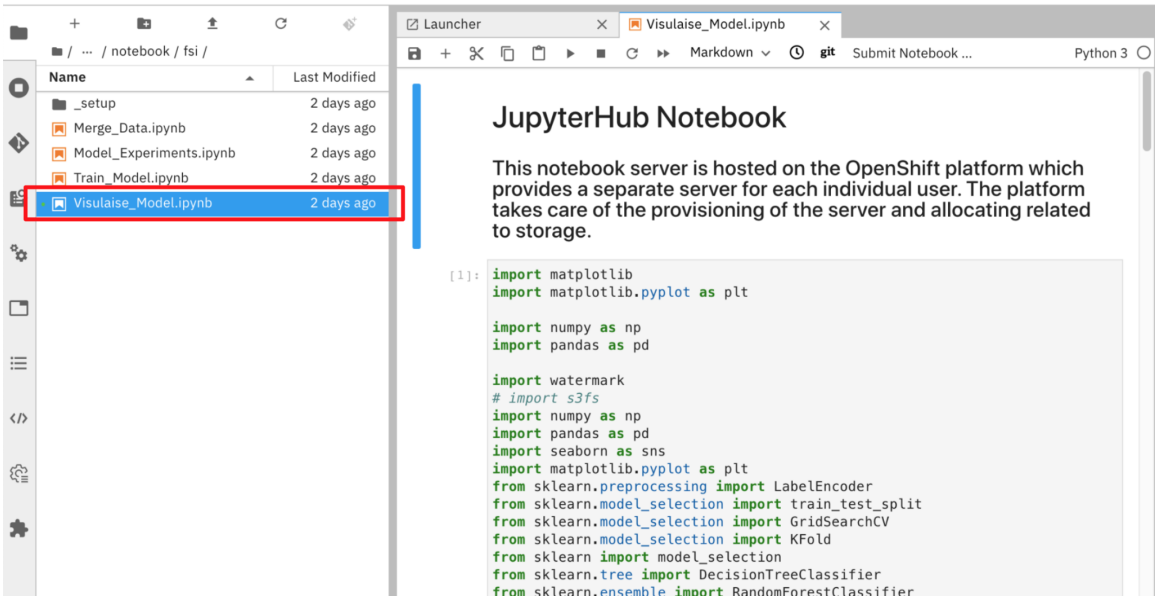
Before we get going, you need to make some small changes to the code.

- Navigate to **ml-workshop-improved/notebook** and open **hyper_parameters.py**
- Locate line of code with **minioFilename** and paste the file name you saved earlier (**YOUR_CSV_FILE**) in this lab into the code as illustrated below. Note, your file name will be unique to you

```
def get_hyper_paras():
    # ###########################################################################
    #
    # PLACE YOUR STUDENT CONFIGURATION INFORMATION IN THIS SECTION OF THE CODE
    #
    user_id = "opentlc-mgr"
    minioFilename = "part-00000-aa888b5b-fd89-431e-831f-cfd9d79e4548-c000.csv"
    # ###########################################################################
```

9. Click **File > Save to** save the file.

Now to run your first notebook, double click the file **Visulaise_Model.ipynb** as shown (The name is a little misleading - it's really there to visualise the data not the model.)

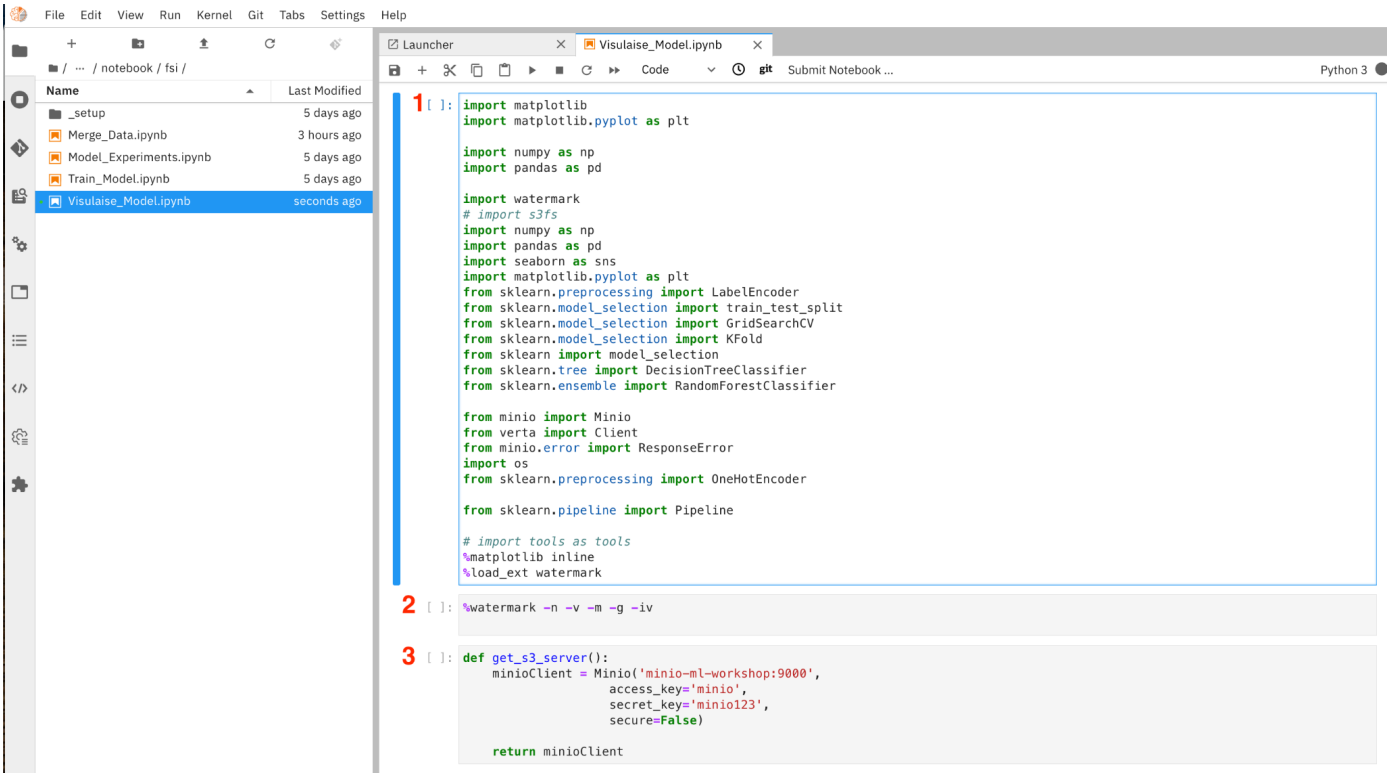


10. Scroll up to the top of the notebook
11. Click in cell **[1]**.

You will now step through the notebook one cell at a time.

12. Type **[Shift] + [Return]** to step through each cell in the notebook.

Now, as previously, select the first cell and walk through each cell executing you go by clicking SHIFT + RETURN.



1. Import our desired Python libraries. (notice we don't need to do any ***pip install***s - our administrator has bundled all of our required libraries into this base container image - which we selected earlier the *MLWorkShop Notebook Image*)
2. *watermark* outputs the versions of various components, libraries, operating system attributes etc.
3. Here we connect to our S3 object store, Minio, using the URL and credentials shown

4

```
minioClient = get_s3_server()
data_file = minioClient.fget_object("data", "full_data_csvuser29/part-00000-8a222f86-81cd-49e5-bc60-e28e3ac60d65-c000.csv", "/tmp/data.csv")
data_file_version = data_file.version_id
data = pd.read_csv('/tmp/data.csv')
data.head(5)
```

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	Premium	RelationshipManager	PrimaryChannel	HasCreditCard	...	IncomeProtection	WealthManagement	HomeEqui
0	148	Male	0	No	No	1	Yes	No	Mobile	No	...	No	No	
1	463	Male	0	Yes	Yes	4	Yes	Yes	Branch	No	...	Yes	No	
2	471	Female	1	No	No	17	Yes	No	No	Not Available	...	Not Available	Not Available	Not A
3	496	Male	0	No	No	22	No	Not available	Mobile	No	...	Yes	No	
4	833	Female	0	Yes	Yes	70	Yes	No	Mobile	Yes	...	Yes	Yes	

5 rows × 21 columns

Use pandas.DataFrame functions

- *shape* to return the dimensionality
- *info* to print a concise summary of the DataFrame
- *describe* to generate descriptive statistics of the DataFrame's columns
- *isnull().sum()* to sum the empty values
- finally determine Churn and Total Changes

5

```
data.shape
```

```
(7043, 21)
```

6

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7043 entries, 0 to 7042
Data columns (total 21 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   customerID           7043 non-null   int64
 1   gender               7043 non-null   object
 2   SeniorCitizen        7043 non-null   int64
 3   Partner              7043 non-null   object
 4   Dependents           7043 non-null   object
 5   tenure               7043 non-null   int64
 6   Premium              6978 non-null   object
 7   RelationshipManager  6978 non-null   object
 8   PrimaryChannel       6978 non-null   object
 9   HasCreditCard        6978 non-null   object
 10  DebitCard            6978 non-null   object
 11  IncomeProtection     6978 non-null   object
 12  WealthManagement     6978 non-null   object
 13  HomeEquityLoans      6978 non-null   object
 14  MoneyMarketAccount   6978 non-null   object
 15  CreditRating         6978 non-null   object
 16  PaperlessBilling     6978 non-null   object
 17  AccountType          6978 non-null   object
 18  MonthlyCharges       6978 non-null   float64
 19  TotalCharges         6967 non-null   float64
 20  Churn                6978 non-null   object
dtypes: float64(2), int64(3), object(16)
memory usage: 1.1+ MB
```

7

```
data.describe()
```

	customerID	SeniorCitizen	tenure	MonthlyCharges	TotalCharges
count	7043.000000	7043.000000	7043.000000	6978.000000	6967.000000
mean	3522.000000	0.162147	32.371149	64.706614	2280.638015

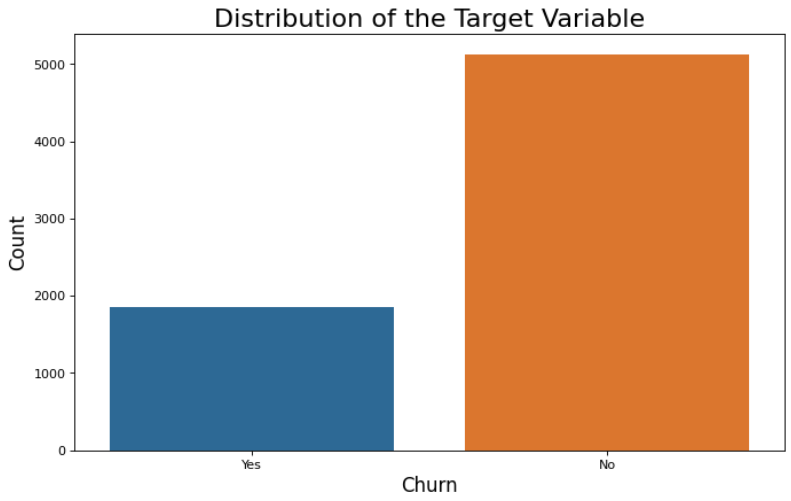
4. In this cell we also output the first 5 lines of the file - so the data scientist can get a quick view of the data.
5. We output the dimensions of the data in rows and columns (features)
6. Here we output various data around the columns (features) including their types, names etc
7. Using *describe(),* we output various statistical data associated with the entire dataset, max, mean etc. values for numeric columns.

8
```
[8]: data.isnull().sum()
```

```
[8]: customerID              0
     gender                  0
     SeniorCitizen           0
     Partner                 0
     Dependents              0
     tenure                  0
     Premium                65
     RelationshipManager    65
     PrimaryChannel         65
     HasCreditCard          65
     DebitCard              65
     IncomeProtection       65
     WealthManagement       65
     HomeEquityLoans        65
     MoneyMarketAccount     65
     CreditRating           65
     PaperlessBilling       65
     AccountType            65
     MonthlyCharges         65
     TotalCharges           76
     Churn                  65
     dtype: int64
```

9
```
[9]: fig = plt.figure(figsize=(10,6), dpi=80)
     ax = sns.countplot(x="Churn", data=data)
     ax.set_title('Distribution of the Target Variable', fontsize=20)
     ax.set_xlabel('Churn', fontsize = 15)
     ax.set_ylabel('Count', fontsize = 15)
```

```
[9]: Text(0, 0.5, 'Count')
```



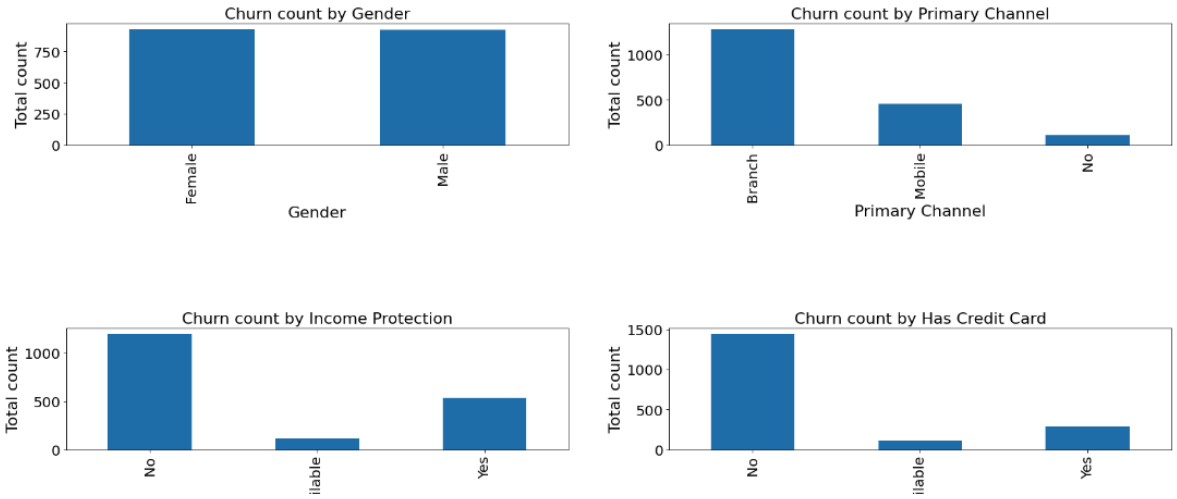
10
```
[10]: # Convert binary variable into numeric so plotting is easier. We need to later take mean
      data['Churn'] = data['Churn'].map({'Yes': 1, 'No': 0})
```

8. We output the sum of rows with null values with nulls - to assess data for errors, e.g null for *charges* indicates an error.
9. Here we output the total count of the ***labeled*** column, Churn. We need a decent spread, and we have it - with just over 2 to 1.
10. Here we make a simple conversion from Yes and No to 1 and 0, to facilitate plotting.

11

```
[11]: fig, ((ax1,ax2),(ax3,ax4), (ax5,ax6)) = plt.subplots(ncols=2, nrows=3, figsize=(25,17), dpi = 80)
plt.subplots_adjust(left=None, bottom=None, right=None, top=None ,wspace=None, hspace=1.5)
plt.rc('xtick', labelsize = 12)    # fontsize of the tick labels
plt.rc('ytick', labelsize = 12)

data.groupby('gender').Churn.sum().plot(kind='bar', ax = ax1)
ax1.set_ylabel('Total count',fontsize = 20)
ax1.set_xlabel('Gender',fontsize = 20)
ax1.tick_params(labelsize = 18)
ax1.set_title('Churn count by Gender',fontsize = 20)

data.groupby('PrimaryChannel').Churn.sum().plot(kind='bar', ax=ax2)
ax2.set_ylabel('Total count',fontsize = 20)
ax2.set_xlabel('Primary Channel',fontsize = 20)
ax2.tick_params(labelsize = 18)
ax2.set_title('Churn count by Primary Channel',fontsize = 20)

data.groupby('IncomeProtection').Churn.sum().plot(kind='bar', ax=ax3)
ax3.set_ylabel('Total count',fontsize = 20)
ax3.set_xlabel('Income Protection',fontsize = 20)
ax3.tick_params(labelsize = 18)
ax3.set_title('Churn count by Income Protection',fontsize = 20)

data.groupby('HasCreditCard').Churn.sum().plot(kind='bar', ax=ax4)
ax4.set_ylabel('Total count',fontsize = 20)
ax4.set_xlabel('Has Credit Card',fontsize = 20)
ax4.tick_params(labelsize = 18)
ax4.set_title('Churn count by Has Credit Card',fontsize = 20)

data.groupby('WealthManagement').Churn.sum().plot(kind='bar',ax=ax5)
ax5.set_ylabel('Total count',fontsize = 20)
ax5.set_xlabel('Wealth Management',fontsize = 20)
ax5.tick_params(labelsize = 18)
ax5.set_title('Churn count by Wealth Management',fontsize = 20)

data.groupby('CreditRating').Churn.sum().plot(kind='bar',ax=ax6)
ax6.set_ylabel('Total count',fontsize = 20)
ax6.set_xlabel('Credit Rating Type',fontsize = 20)
ax6.tick_params(labelsize = 18)
ax6.set_title('Churn count by Credit Rating',fontsize = 20)
```

```
[11]: Text(0.5, 1.0, 'Churn count by Credit Rating')
```



11. This cell visually outputs churn count by various features in the data set

12
```
data.replace(" ", np.nan, inplace=True)
```

13
```
data.isna().sum()
```

```
customerID              0
gender                  0
SeniorCitizen           0
Partner                 0
Dependents              0
tenure                  0
Premium                65
RelationshipManager    65
PrimaryChannel         65
HasCreditCard          65
DebitCard              65
IncomeProtection       65
WealthManagement       65
HomeEquityLoans        65
MoneyMarketAccount     65
CreditRating           65
PaperlessBilling       65
AccountType            65
MonthlyCharges         65
TotalCharges           76
Churn                  65
dtype: int64
```

14
```
data['TotalCharges'] = pd.to_numeric(data['TotalCharges'])
```

15
```
mean = data['TotalCharges'].mean()
data.fillna(mean, inplace=True)
# Now we know that total charges has nan values
data.isna().sum()
```

```
customerID             0
gender                 0
SeniorCitizen          0
Partner                0
Dependents             0
tenure                 0
Premium                0
RelationshipManager    0
PrimaryChannel         0
HasCreditCard          0
DebitCard              0
IncomeProtection       0
WealthManagement       0
HomeEquityLoans        0
MoneyMarketAccount     0
CreditRating           0
PaperlessBilling       0
AccountType            0
MonthlyCharges         0
TotalCharges           0
Churn                  0
dtype: int64
```

16
```
plt.figure(figsize=(10,8), dpi=80)
# Sns.set(rc={'figure.figsize':(25,15)})
ax = sns.catplot(x="CreditRating", y="TotalCharges", hue="Churn", kind="box", data=data, height = 6,aspect = 1.5,palette = 'RdBu')
plt.title('Comparison of Total Charges for each CreditRating',fontsize = 20)
plt.xlabel('CreditRating',fontsize = 15)
plt.ylabel('Total Charges',fontsize = 15)
```

```
Text(17.690962577160498, 0.5, 'Total Charges')
<Figure size 800x640 with 0 Axes>
```

12. Replace spaces with numpy NAN values
13. Output sum of NAN and None values
14. Convert to numeric
15. Fill NANs with the mean
16. Here we output a box plot - a useful visualization of 2 dimensions by the labeled column *Churn*

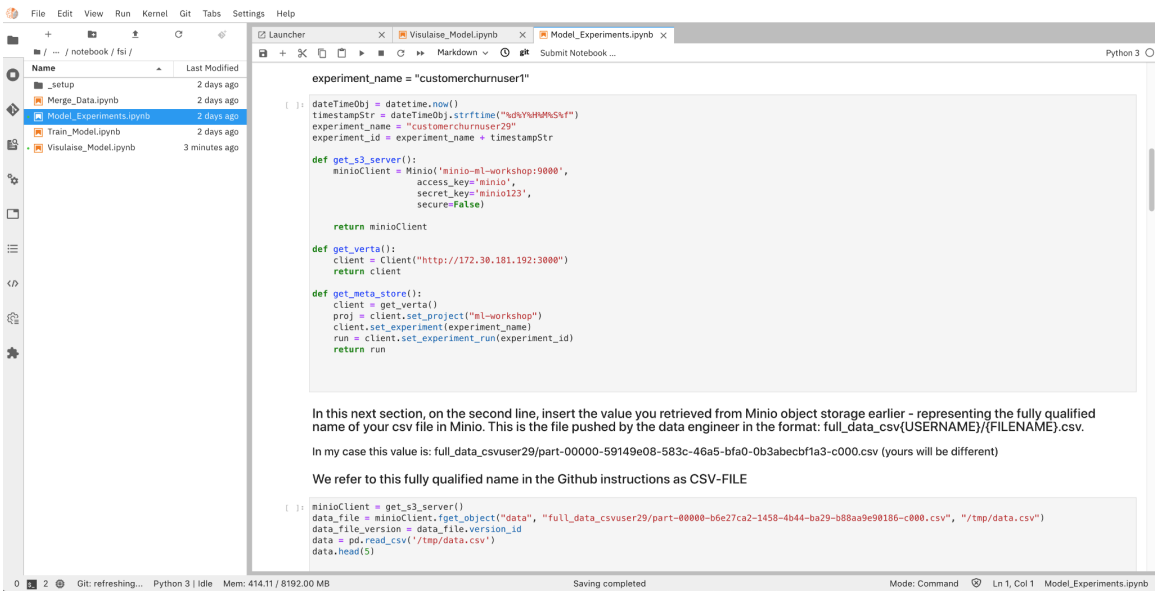
Part 2: Experiment with Models

At this point, as a data scientist, we have a good understanding of the data. Now it's time to start experimenting with different models, parameters and hyper parameters.

As we experiment, we want our notebook to create an experiment id for every experiment (which is guaranteed to be unique within our team, as it uses user id and timestamp as a basis).

This experiment id is then used as an identifier when we push our experiment metadata and binaries to our model repository, Verta. In this way, we retrieve and repeat any experiment we have done, as well as share this experiment with other team members, breaking down silos between teams and individuals in AI//ML workflows.

1. Using the File Explorer, open the **Model_Experiments.ipynb** notebook. Jupyterhub opens the code windows.



Next, just as you did in the previous lab you will step through the code one cell at a time. **Note there are 3 changes you'll need to make to your cells - which we'll highlight below.**

2. Scroll up to the top of the notebook
3. Click in cell **[1]**.

You will now step through the notebook one cell at a time.

4. Type **[Shift] + [Return]** to step through each cell in the notebook.

Make sure you **pause** and make the changes **as indicated in red**, before executing certain cells.

[1]:
```
import os
# os.environ["MODIN_ENGINE"] = "ray"
```

[2]:
```
import matplotlib
import matplotlib.pyplot as plt

import numpy as np
# import pandas as pd
# import modin.pandas as pd

import watermark
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold
from sklearn import model_selection
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from datetime import datetime
import verta.integrations.sklearn
from minio import Minio
from verta import Client
from minio.error import ResponseError
import os
from sklearn.preprocessing import OneHotEncoder

from sklearn.pipeline import Pipeline


# import tools as tools
%matplotlib inline
%load_ext watermark
```

[3]:
```
%watermark -n -v -m -g -iv
```

1. Do imports - more applicable when we want to use *Ray*.
2. More imports of libraries we need
3. See *watermark* description of this cell above in - ***first Data Science workshop - Visualisation***

Load Lab parameters. Before running this cell, ensure you set the s3BucketFullPath va
hyper_parameters.py

4
```
from hyper_parameters import get_hyper_paras
user_id,PROJECT_NAME,EXPERIMENT_NAME,experiment_name,s3BucketFullPath = get_hyper_paras()
```

In this next section, we initialise our variables and our Object Storage implementation, Minio

5
```
HOST = "http://mlflow:5500"

#PROJECT_NAME = "CustomerChurnUser60"
#EXPERIMENT_NAME = "CustomerChurnUser60"

os.environ['MLFLOW_S3_ENDPOINT_URL']='http://minio-ml-workshop:9000'
os.environ['AWS_ACCESS_KEY_ID']='minio'
os.environ['AWS_SECRET_ACCESS_KEY']='minio123'
os.environ['AWS_REGION']='us-east-1'
os.environ['AWS_BUCKET_NAME']='mlflow'

dateTimeObj = datetime.now()
timestampStr = dateTimeObj.strftime("%d%Y%H%M%S%f")
experiment_id = experiment_name + timestampStr


def get_s3_server():
    minioClient = Minio('minio-ml-workshop:9000',
                    access_key=os.environ['AWS_ACCESS_KEY_ID'],
                    secret_key=os.environ['AWS_SECRET_ACCESS_KEY'],
                    secure=False)

    return minioClient


import mlflow

# Connect to local MLflow tracking server
mlflow.set_tracking_uri(HOST)

# Set the experiment name...
mlflow.set_experiment(EXPERIMENT_NAME)

mlflow.sklearn.autolog(log_input_examples=True)
```

In this next section, we pull in the merged CSV file prepared earlier by the data engineer.

6
```
minioClient = get_s3_server()
data_file = minioClient.fget_object("data", s3BucketFullPath, "/tmp/data.csv")
data_file_version = data_file.version_id
data = pd.read_csv('/tmp/data.csv')
data.head(5)
```

4. Here we pull in the user-specific parameters you added earlier to **hyper_parameters.py** for use in the file.
5. Here we add some simple integration code - to allow the data scientist
    a. Retrieve the CSV file prepared earlier data engineer, from Minio S3 object storage
    b. Participate their work in the workflow.
    The HOST line on top and 4 lines using ml-flow is all they need to push all of their experiments to our model registry **Ml Flow**
6. Here we retrieve the CSV we prepared in the Data Engineer lab earlier and output the first 5 lines of the file - so the data scientist can get a quick view of the data.

Run all the way down to cell 17, *Feature Engineering Pipeline*, as all cells until then are discussed above in – ***first Data Science workshop - Visualisation***

Feature Engineering pipeline

Use category_encoder's Ordinal encoding method which uses a single column of integers to rep
imported earlier. Then pickle it and transform it. Then use Onehot (or dummy) coding for categ
binary.

17
```
import category_encoders as ce
import joblib

names = ['gender', 'Partner', 'Dependents', 'PhoneService', 'StreamingTV', 'StreamingMovies', 'PaperlessBill
# for column in names:
#     labelencoder(column)

enc = ce.OrdinalEncoder(cols=names)
enc.fit(data)
joblib.dump(enc, 'enc.pkl')
labelled_set = enc.transform(data)
labelled_set.tail(5)
```

18
```
names = ['MultipleLines', 'InternetService', 'Contract', 'PaymentMethod', 'OnlineSecurity', 'OnlineBackup',
         'DeviceProtection', 'TechSupport']

ohe = ce.OneHotEncoder(cols=names)
ohe.fit(labelled_set)
joblib.dump(ohe, 'ohe.pkl')
final_set = ohe.transform(labelled_set)
final_set.tail(5)
```

Now we use scikit-learn's 'train_test_split' function to randomly split our data into training and t
from our training and testing datasets and output the shaope of our data.

19
```
labels = final_set['Churn']
X_train, X_test, y_train, y_test = train_test_split(final_set, labels, test_size=0.2)
X_train.pop('Churn')
X_train.pop('customerID')
X_test.pop('Churn')
X_test.pop('customerID')
print ('Training Data Shape',X_train.shape, y_train.shape)
print ('Testing Data Shape',X_test.shape, y_test.shape)
```

20
```
# Data For cross validation and GridSearch
Y = final_set['Churn']
X = final_set.drop(['Churn', 'customerID'], axis=1)
print ('Training Data Shape', X.shape)
print ('Testing Data Shape', Y.shape)
```

17. Here we use an Ordinal Encoder to convert simple binary values to a numeric representation. Output the data after applying the Ordinal Encoder.
18. Here we use a One Hot Encoder to convert multi valued features to a numeric representation. Output the data after applying the One Hot Encoder.
19. Here we split our data set into a training and a testing set, and discard unwanted columns customer id and our labeled column Churn.
20. Further data set refinement.

Create DecisionTreeClassifier object, extract hyper parameters, and then GridSearch will best_model frc

21
```
# Create decision tree object
DT = DecisionTreeClassifier()
# List of parameters
# entropy
criterion = ['gini']
max_depth = [5,10,15]
min_samples_split = [2,4,6]
min_samples_leaf = [4,5,6,8]
# Save all the lists in the variable
hyperparameters = dict(max_depth=max_depth, criterion=criterion,min_samples_leaf = min_samples_leaf ,min_samples_spli
```

22
```
model = GridSearchCV(DT, hyperparameters, cv=5, verbose=0)
best_model = model.fit(X,Y)
```

23
```
# Mean cross validated score
print('Mean Cross-Validated Score: ',best_model.best_score_)
print('Best Parameters',best_model.best_params_)
# You can also print the best penalty and C value individually from best_model.best_estimator_.get_params()
print('Best criteria:', best_model.best_estimator_.get_params()['criterion'])
print('Best depth:', best_model.best_estimator_.get_params()['max_depth'])
```

Use K-Folds cross-validator to split data in train/test sets. Create a dictionary of hyperparameter candid
assess results, print and store hyper parameters and accuracy and tag using 'DecisionTreeClassifier'

24
```
kfold = KFold(n_splits = 3)
hyperparameters = dict(max_depth=5, criterion='gini',min_samples_leaf = 3 ,min_samples_split = 10)
model = DecisionTreeClassifier(max_depth=5, criterion='gini',min_samples_leaf = 3 ,min_samples_split = 10)
model = model.fit(X_train, y_train)
joblib.dump(model, 'dct.pkl')
results = model_selection.cross_val_score(model,X,Y,cv = kfold)
print(results)
print('Accuracy',results.mean()*100)
```

21. Create a DecisionTreeClassifier with these hyper parameters
22. User GridSearch to output the best model / hyper parameters from the combinations supplied to its *fit* method.
23. Print out those best model parameters
24. Use K-Folds cross-validator to split data into train/test sets. Create a dictionary of hyperparameter candidates, train the model using a DecisionTreeClassifier. Print and store hyperparameters and accuracy in Verta and tag using 'DecisionTreeClassifier''. The ***store*** method lushes this metadata to Verta, which you'll see below.

Create RandomForestClassifier object, extract hyper parameters, and then the best_model

25
```
dateTimeObj = datetime.now()
timestampStr = dateTimeObj.strftime("%d%Y%H%M%S%f")
experiment_name = "customerchurnuser29"
experiment_id = experiment_name + timestampStr


# Create random forest object
RF = RandomForestClassifier()
n_estimators = [18,22]
criterion = ['gini', 'entropy']
# Create a list of all of the parameters
max_depth = [30,40,50]
min_samples_split = [6,8]
min_samples_leaf = [8,10,12]
# Merge the list into the variable
hyperparameters = dict(n_estimators = n_estimators,max_depth=max_depth, criterion=criterion,min_samples_leaf = min
# Fit your model using gridsearch
model = GridSearchCV(RF, hyperparameters, cv=5, verbose=0)
best_model = model.fit(X, Y)
```

Extract best scores, params, criteria and depth from our model.

26
```
# Mean cross validated score
print('Mean Cross-Validated Score: ',best_model.best_score_)
print('Best Parameters',best_model.best_params_)
# You can also print the best penalty and C value individually from best_model.best_estimator_.get_params()
print('Best criteria:', best_model.best_estimator_.get_params()['criterion'])
print('Best depth:', best_model.best_estimator_.get_params()['max_depth'])
print('Best estimator:', best_model.best_estimator_.get_params()['n_estimators'])
```

As above, use K-Folds cross-validator to split data in train/test sets. Create a dictionary of hyperparar
RandomForestClassifier, assess results, print and store hyper parameters and accuracy and tag using

27
```
kfold = KFold(n_splits = 3)
hyperparameters = dict(max_depth=40, criterion='gini',min_samples_leaf = 12 ,min_samples_split = 8, n_estimators =
model = RandomForestClassifier(max_depth=40, criterion='gini',min_samples_leaf = 12 ,min_samples_split = 8, n_esti
model = model.fit(X_train, y_train)
joblib.dump(model, 'rft.pkl')
results = model_selection.cross_val_score(model,X,Y,cv = kfold)
print(results)
print('Accuracy',results.mean()*100)
# store = get_meta_store()
# store.log_hyperparameters(hyperparameters)
# store.log_model(model)
# store.log_metric('Accuracy',results.mean()*100)
# store.log_tag("RandomForestClassifier")
# store.log_attribute("data_file_location", "data/full_data_csv/a.csv")
# store.log_attribute("data_file_version", data_file_version)
```

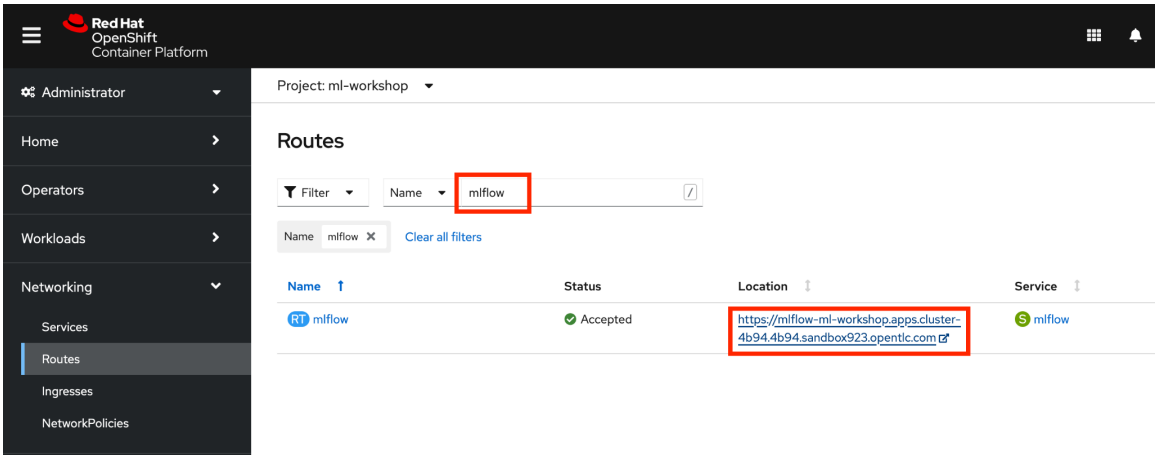
```
print('Notebook complete')
```

25. Modify the 3rd line substituting your username in place of user29.
    Then we create a Random Forest Classifier with these hyper parameters.
26. This cell and cell 27 are equivalents of the previous Decision Tree classifier cells.

Part 3: Visualise the Model Experiments.

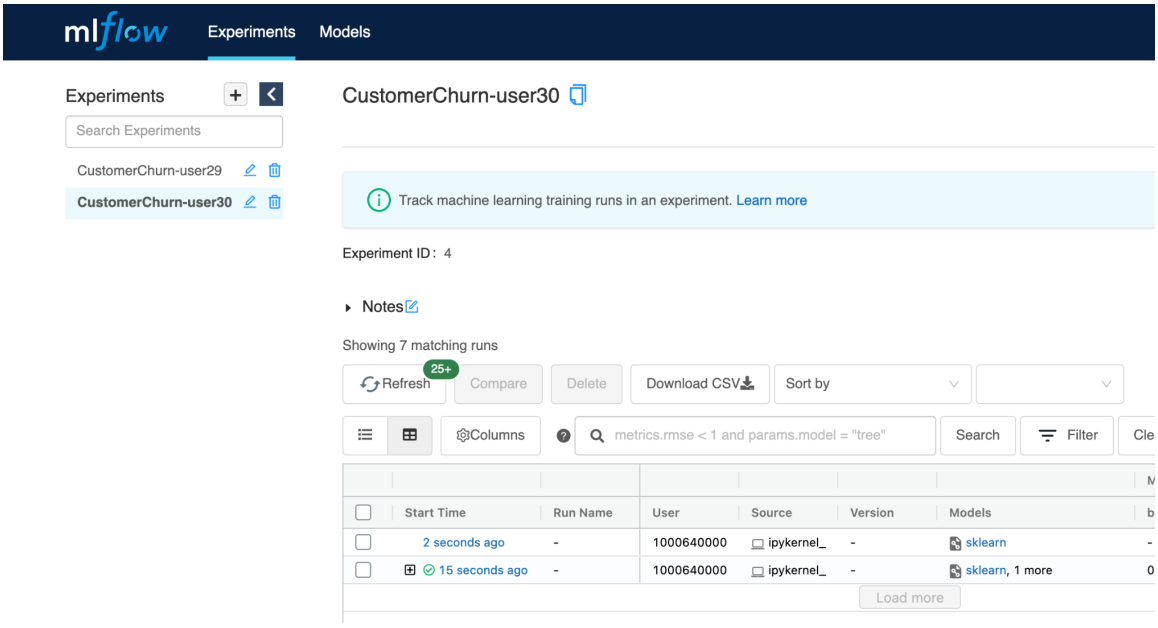
Let's use our model registry **MI-Flow** to analyse compare the model performance.

1. Open the OpenShift console tab in your browser.
2. Select the **Administrator Perspective**.
3. Click **Networking > Routes**.
4. Type ***mlflow*** in the Filter text box.
   OpenShift will display the link to the ***MI Flow*** tool.

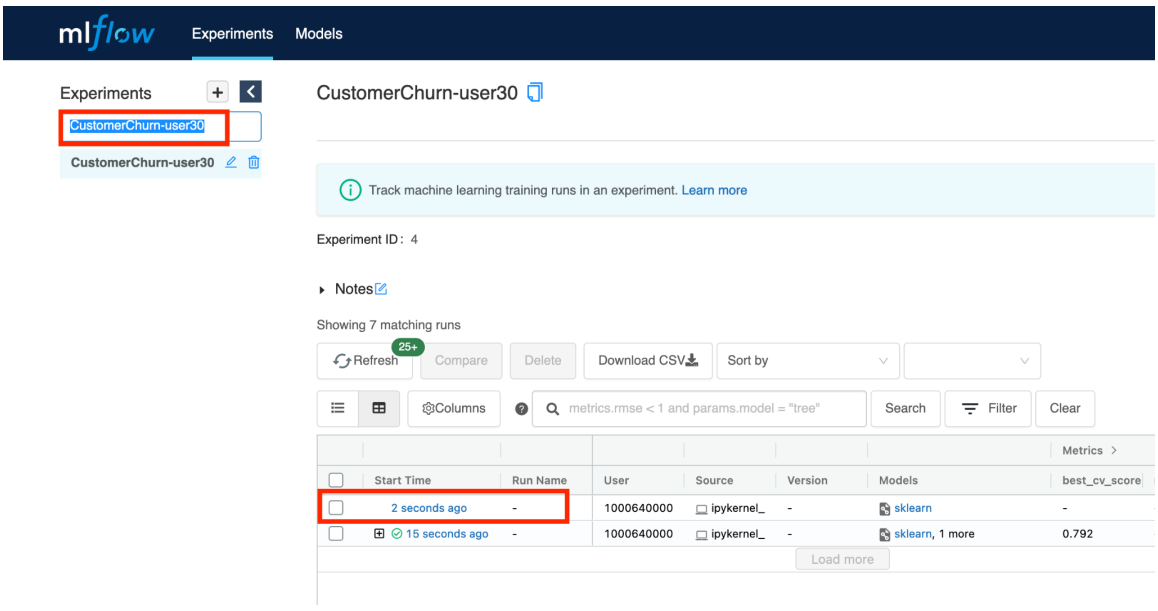


5. Click the hyperlink in the **Location** column
   OpenShift will launch the **MI-Flow** console in a new browser tab.

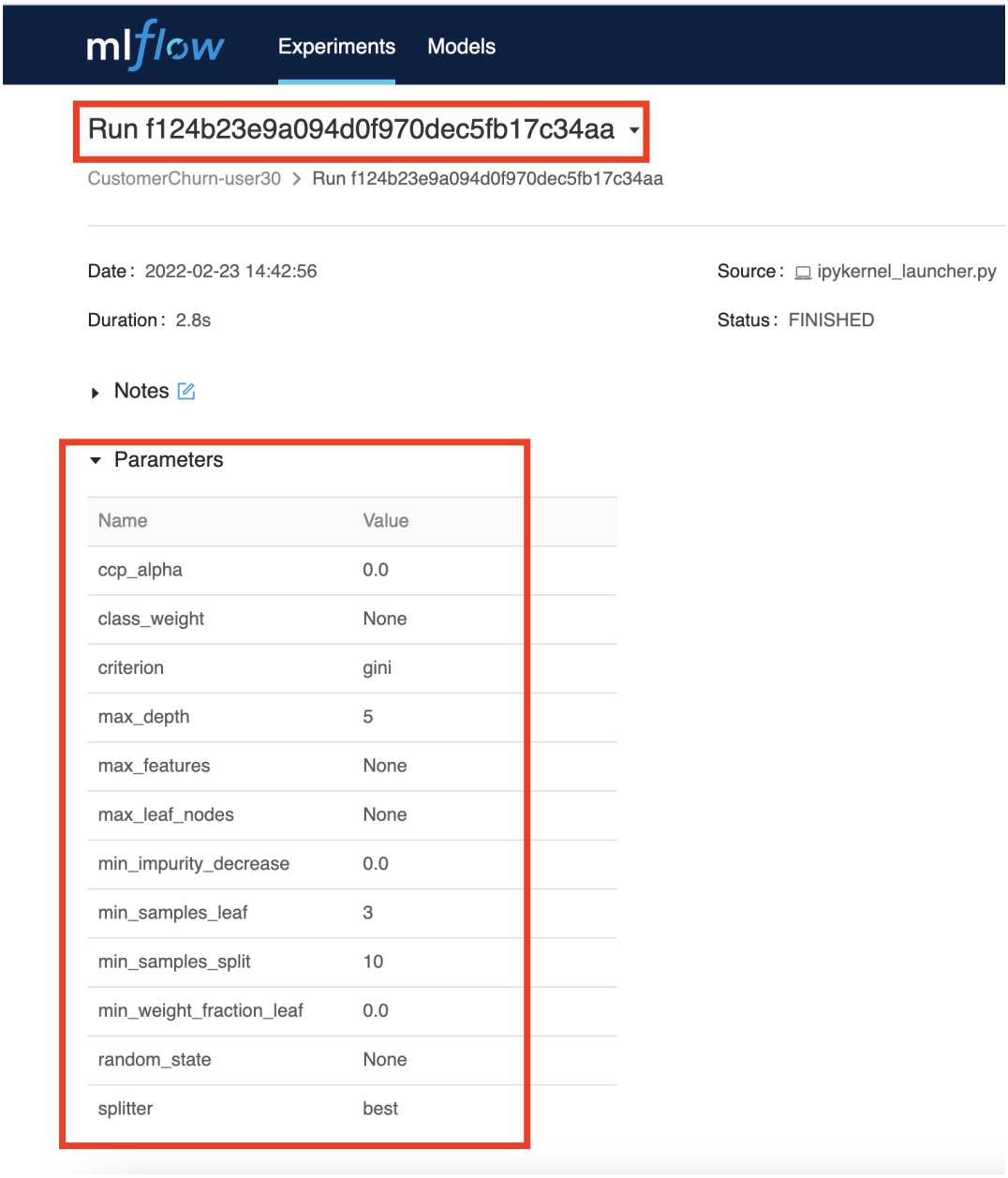
6. After logging in with your OpenShift credentials, you'll be presented with a screen like this - showing all users' experiments.



7. Filter on CustomerChurn-**userXX** , replacing **userXX** with your username in my case CustomerChurn-user30 - so you only see your own experiments.

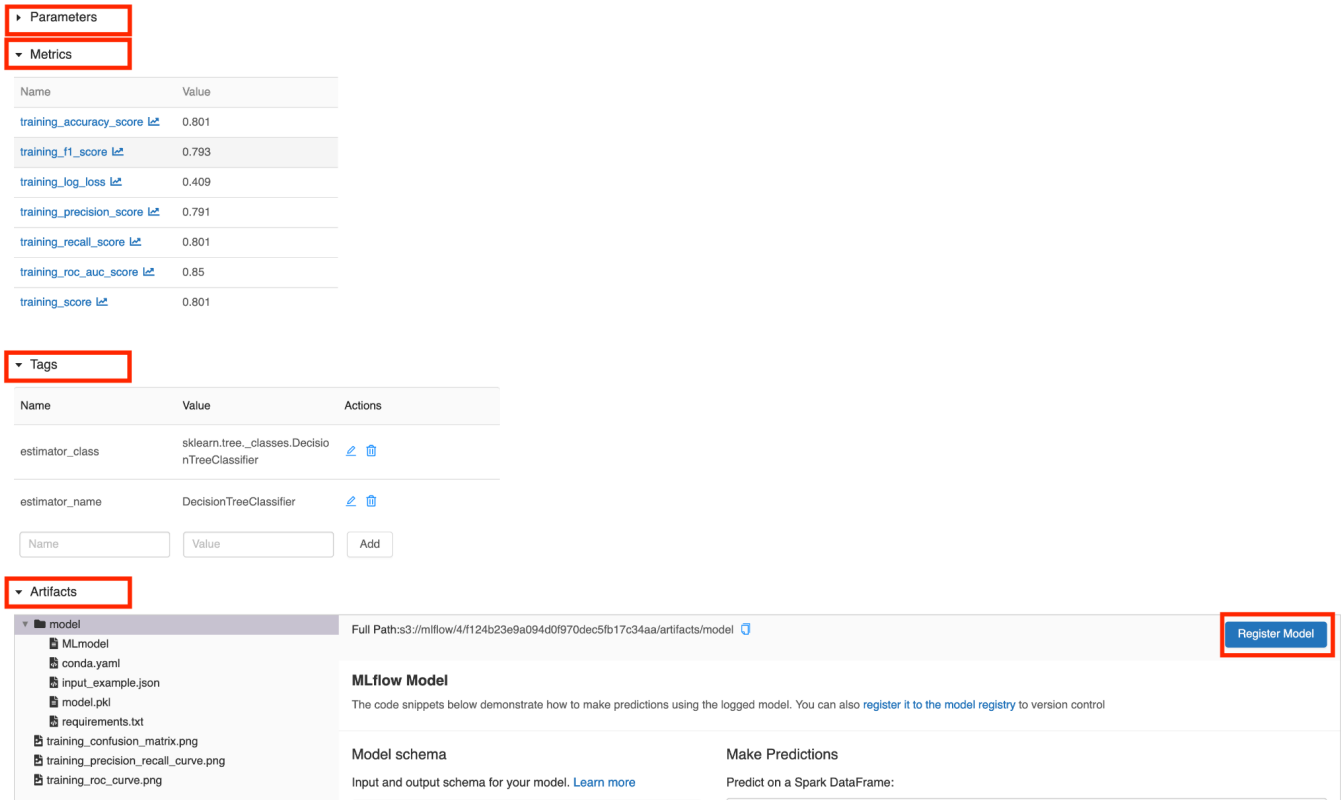


8. Click on the link under Start Time - to drill into one. You can see that out of the box, just by adding the simple ***mlflow*** based integration code you added earlier to your notebook, out of the box, you get an fantastic amount of useful information. Including
    a. every experiment gets an id - which s useful for sharing and traceability purposes later
    b. your parameters are recorded as shown:



Scroll down and you see you get more - parameters, metrics, tags and artifacts associated with the experiment's model output - binaries, yaml, json etc..

You also get the option to register the model - if you want to push it to production for example. Later on, you'll do this.



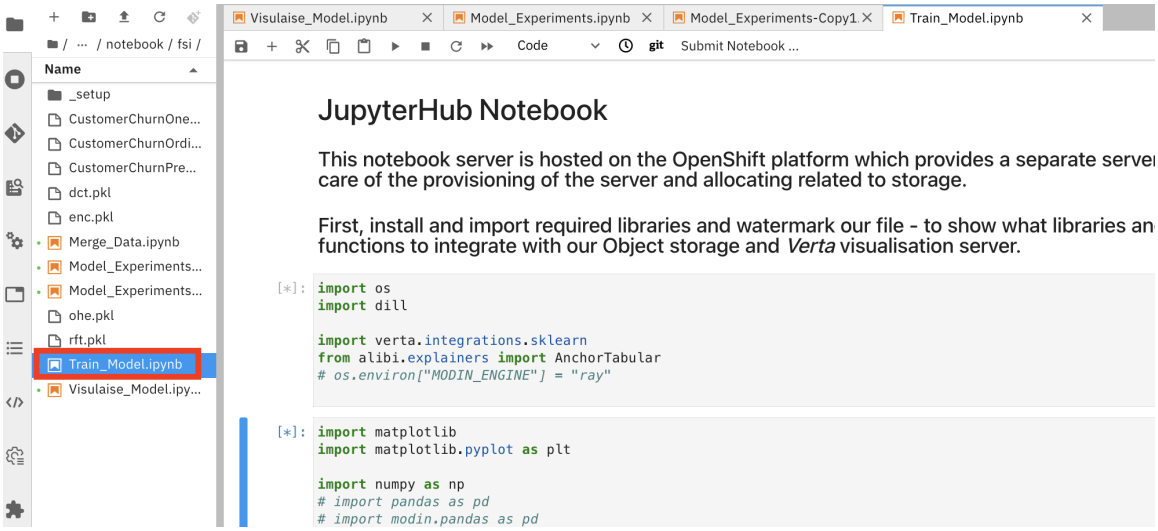
Ml Flow is a very powerful capability for the following reasons. Every experiment that is ever run:

- can be easily pushed to this repo
- gets a unique ID - that can be traced throughout the workflow and its metadata, parameters etc, examined here at any time
- each experiment can be easily retrieved and repeated here by ***the same*** data scientist
- each experiment can be easily shared then retrieved and repeated here by ***other*** data scientists

Part 4: Train the Model

Now following examination of our experiments In Verta, let's assume for performance reasons, as a data scientist, you've decided to proceed with the Decision Tree Classifier experiment and push that model to production.

1. Go back to Jupyter Hub and ensure you're in **ml-workshop-improved/notebook**
2. Open the ***Train_Model.ipynb Jupyter*** notebook.



You have already encountered most of the cells here in the 2 previous notebooks. Therefore, we will just it describe at a high level here - then you can run the notebook yourself.

- Assume the data scientist did the two experiments you ran previously in the ***Model_Experiments.ipynb*** notebook. (in reality, they would probably run a lot more than two)
- After examining both experiments in Jupyter and Ml Flow, they decide they want to proceed with DecisionTreeClassifier - for whatever reason.
- So in this notebook, ***Train_Model.ipynb*** they use the DecisionTreeClassifier algorithm and all of the same parameters/hyper parameters they used previously, then train the model in the same way, and again push it to Ml Flow - this time with the model binaries (CustomerChurnPredictor.sav, CustomerChurnOrdinalEncoder.pkl, CustomerChurnOneHotEncoder.pkl) so they can be pushed to production later.

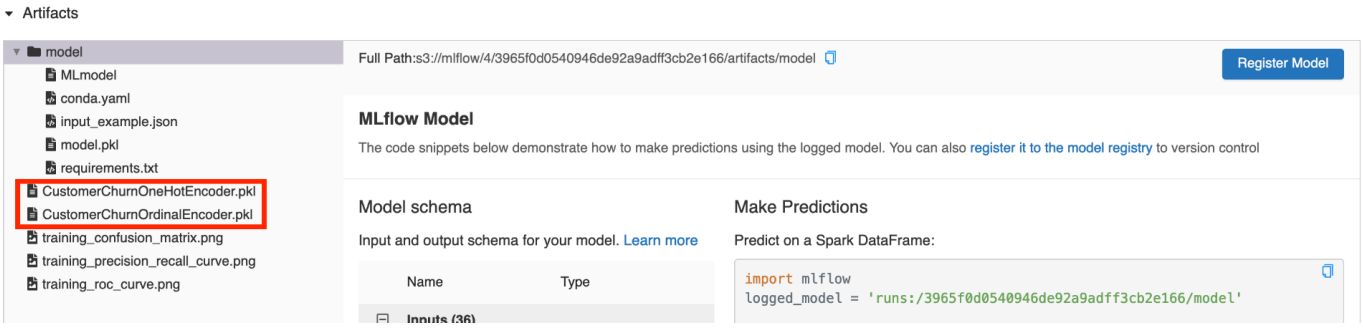
Now run the ***Train_Model.ipynb Jupyter*** notebook - cell by cell by continually hitting **SHIFT + ENTER**, on by choosing menu item **Kernel > Restart Kernel and Run all Cells**

Part 6: Register the chosen Model

Go back to ML Flow - as described above in section **Part 3: Visualise the Model Experiments**

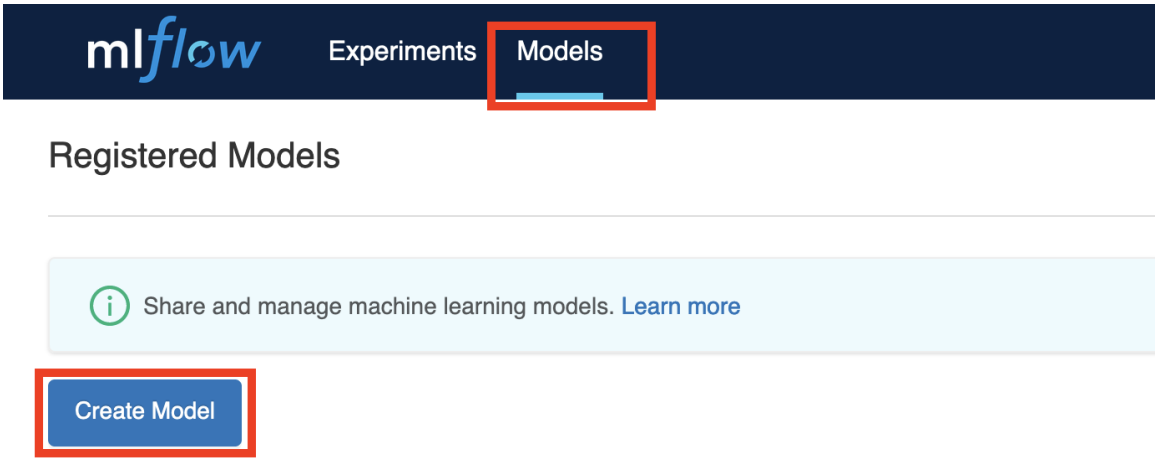
Find your latest experiment as described above. This will be the one associated with the ***Train_Model.ipynb*** experiment.(it needs to be this one - as the 2 encoder binaries are pushed to the repo).

Verify this by navigating down to Artifacts as described above in **Part 3: Visualise the Model Experiments.** Make sure the 2 encoders are there. If not, you probably selected the wrong experiment on the previous page. Make sure you have chosen the most recent.

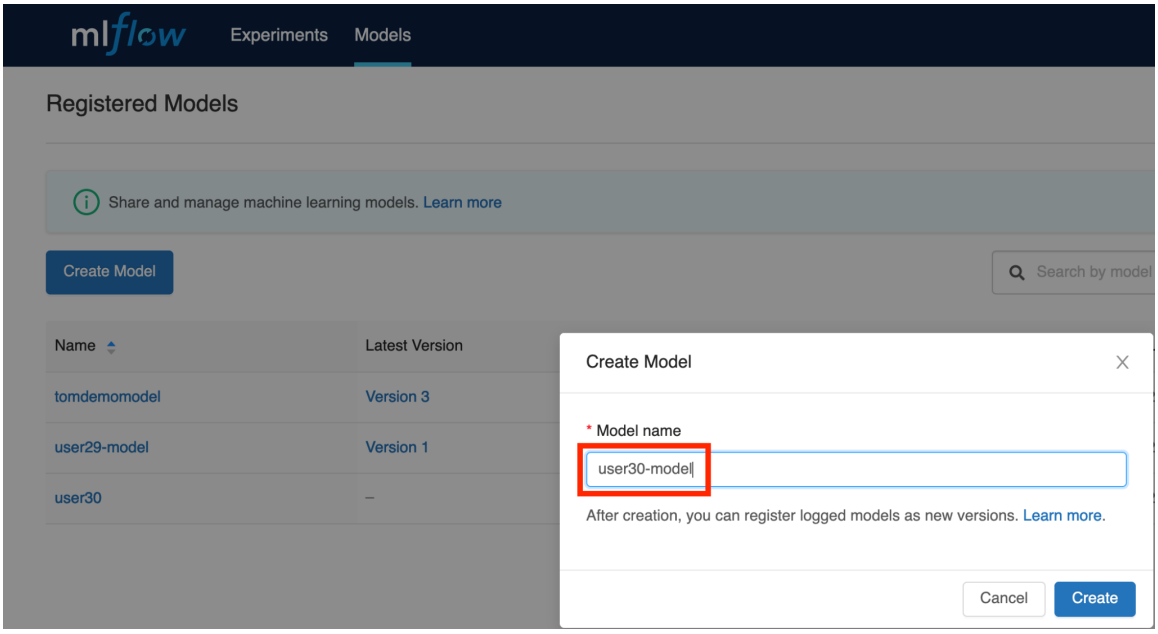


Now we need to register a Model in ML Flow - to which we add our chosen experiment - for later deployment to production.

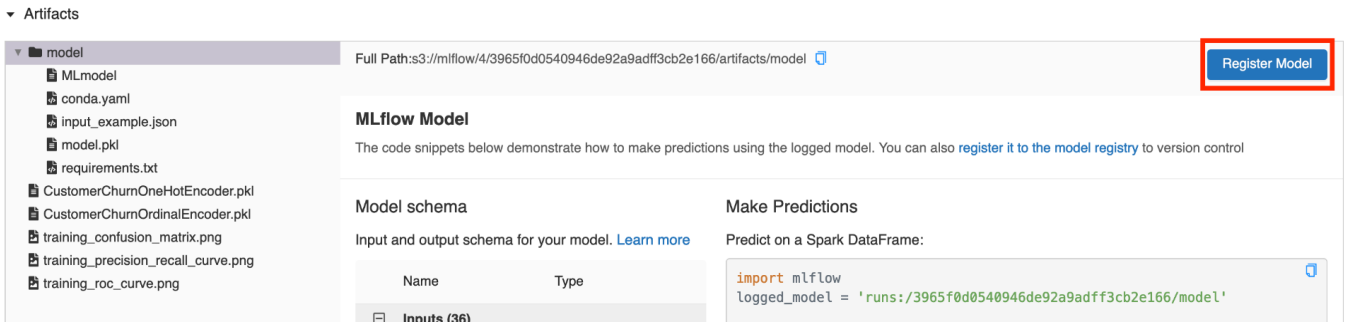
Leave this tab open and open ML Flow again - this time in a separate tab. Choose **Models** and **Create Model**:



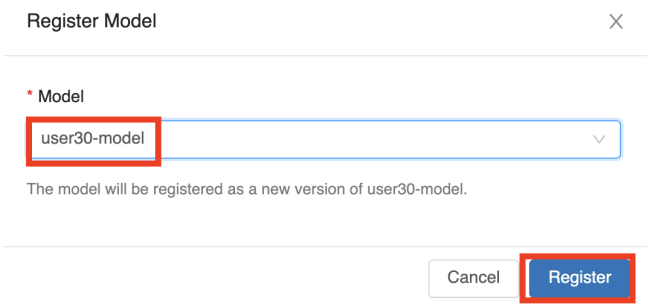
Name your model your *username-model*, i.e. **userXX-model** and click **Create**



Close this tab and go back to your other open Ml Flow tab. Click **Register Model**



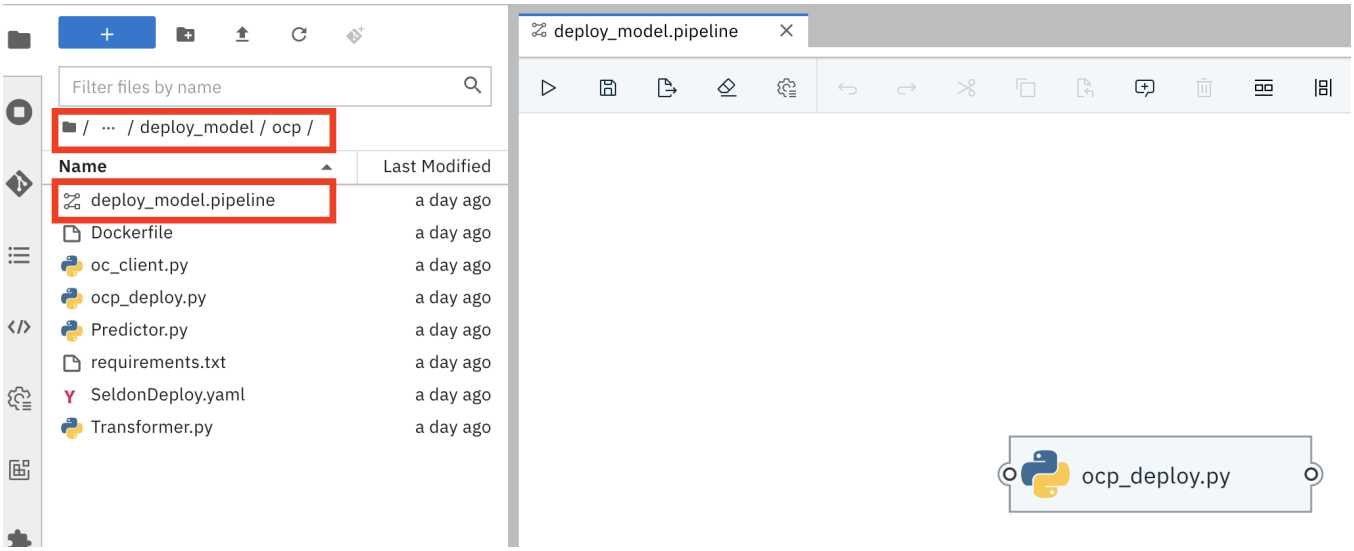
Choose your newly registered model and click **Register**:



You can see a new model is created with version 1. Each time you register a new model under this model name, its version is incremented.

We're now ready to run an Airflow deployment pipeline to deploy this chosen model to a higher environment like Test or even Production.

Move back to Jupyter Hub, navigate to **ml-workshop-improved/airflow/deploy_model/ocp/** and open **deploy_model.pipeline**. ocp_deploy.py should already be there on the canvas as shown:



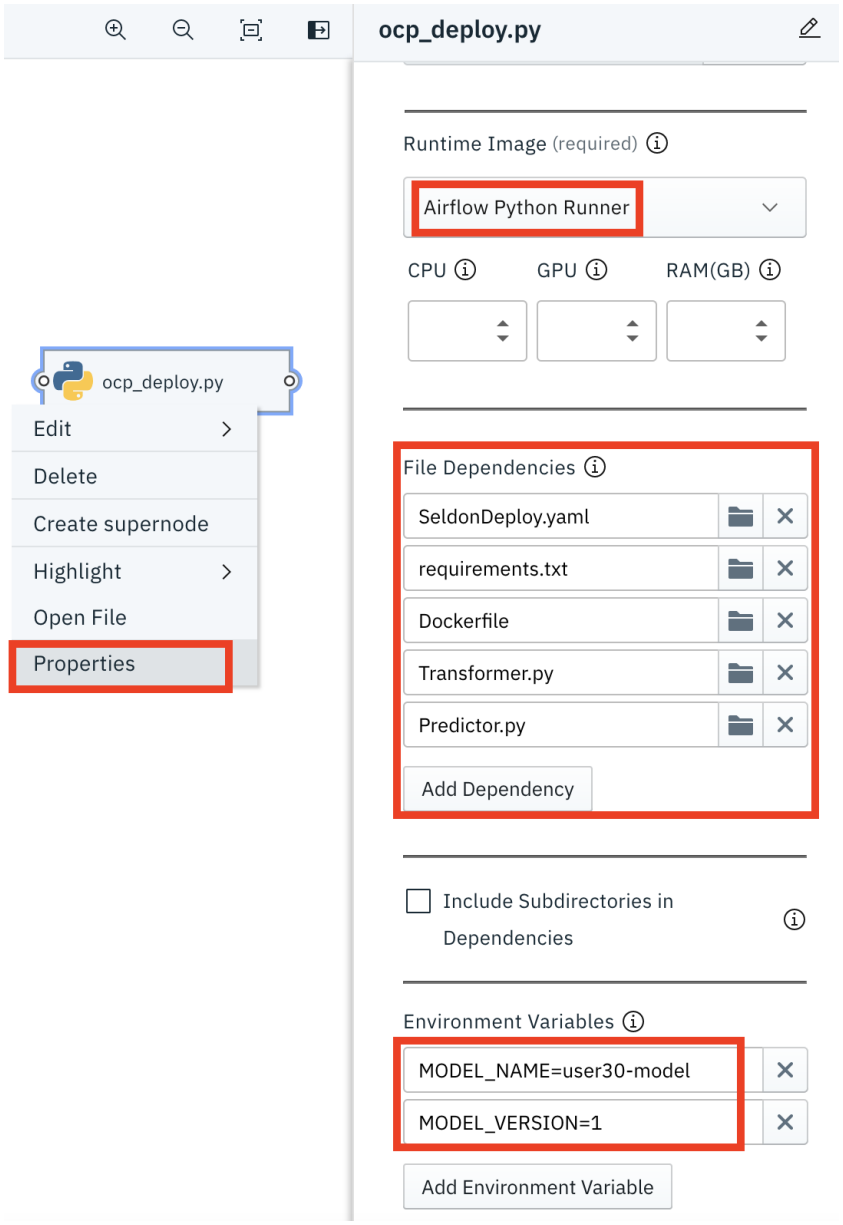
Right click on the **ocp_deploy.py** pipeline element and choose **Properties**.Ensure your selections look like the screenshot below. Pay particular attention to MODEL_NAME - you'll need to change it to:

MODEL_NAME=**userXX-model**

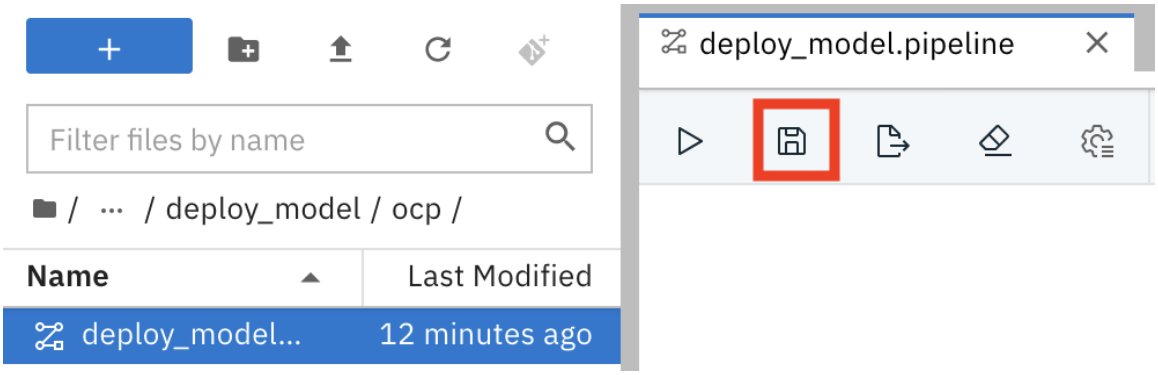
i.e. the model name you created in the previous step, in my case **.**

As this is the first version, you should not need to change your MODEL_VERSION yet, but change it running a subsequent time

This is how it should look:



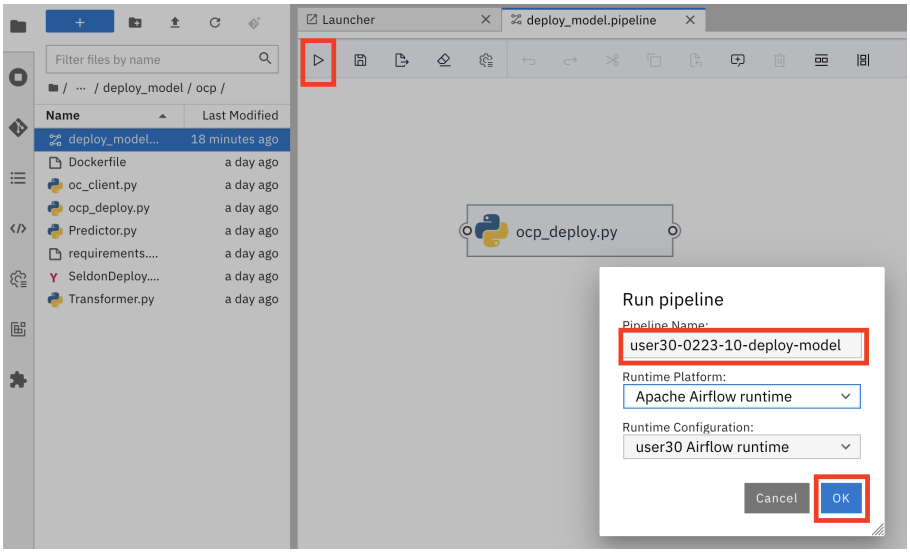
Save your work by clicking on the **Save** button



Run Pipeline

You can now run your pipeline. Click the Play button as shown. A useful naming convention is to enter a name in a format beginning with your username - followed by the month and day and the count of runs that day. This sends each run to the bottom of the list on the Airflow GUI.

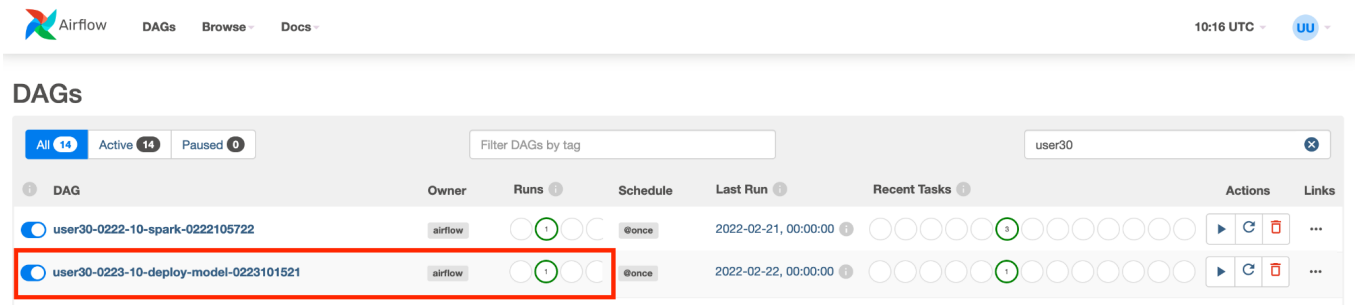
something like **user**XX-**MMDD**-01-deploy-model, in my case ***user30-0223-10-deploy-model***. Also choose your Airflow Runtime and Runner as shown:



After 2 informational popups, your pipeline will kick off.

Now it's time to view your pipeline in our workflow scheduler Airflow.

The same way you did previously with the Data Engineer pipeline, in a browser open your **Airflow Route URL.** After logging in with your OpenShift credentials. click DAGs and again filter on your username - and you'll see something like this:



You should see your pipeline runs - including the deploy model one you just kicked off. Browse around as before - check logs, see how it can be used to schedule this job periodically etc.

When the colour has turned dark green - as it will if all goes well, you're done! Otherwise, let an administrator know!

Now move to the next lab ***AI/ML on OpenShift Workshop - V2 - Lab 3 - ML OPs & Inference.***