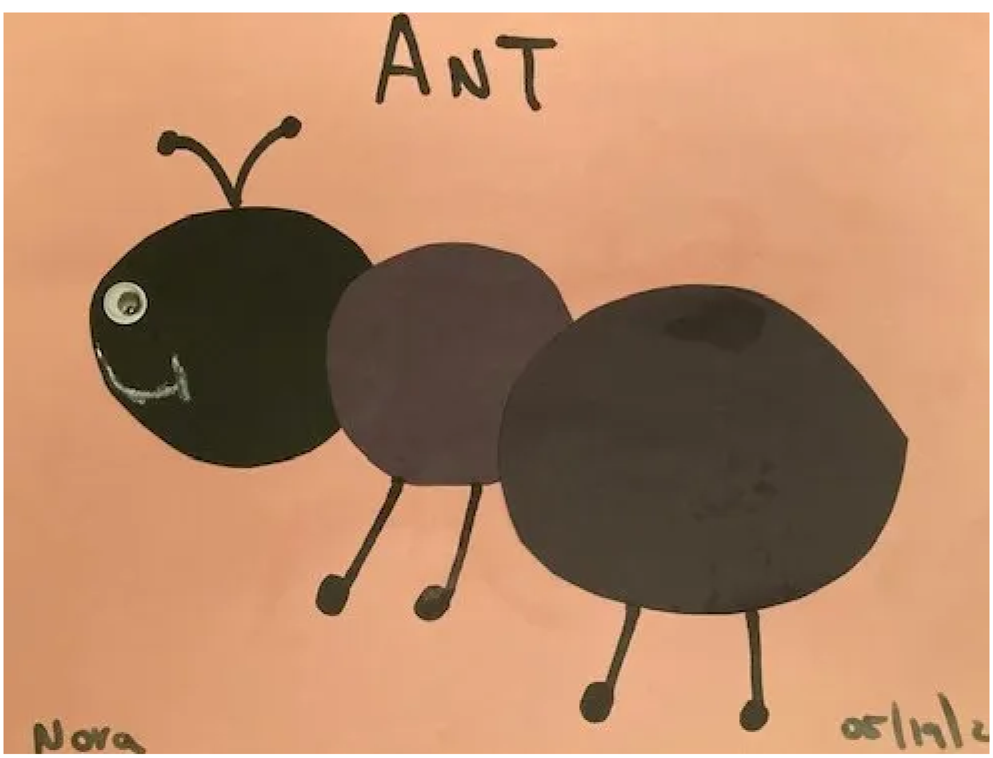
Databricks End-To-End Machine Learning - Create An Ingest-To-Serving MLOps Pipeline

Thomas Jaensch



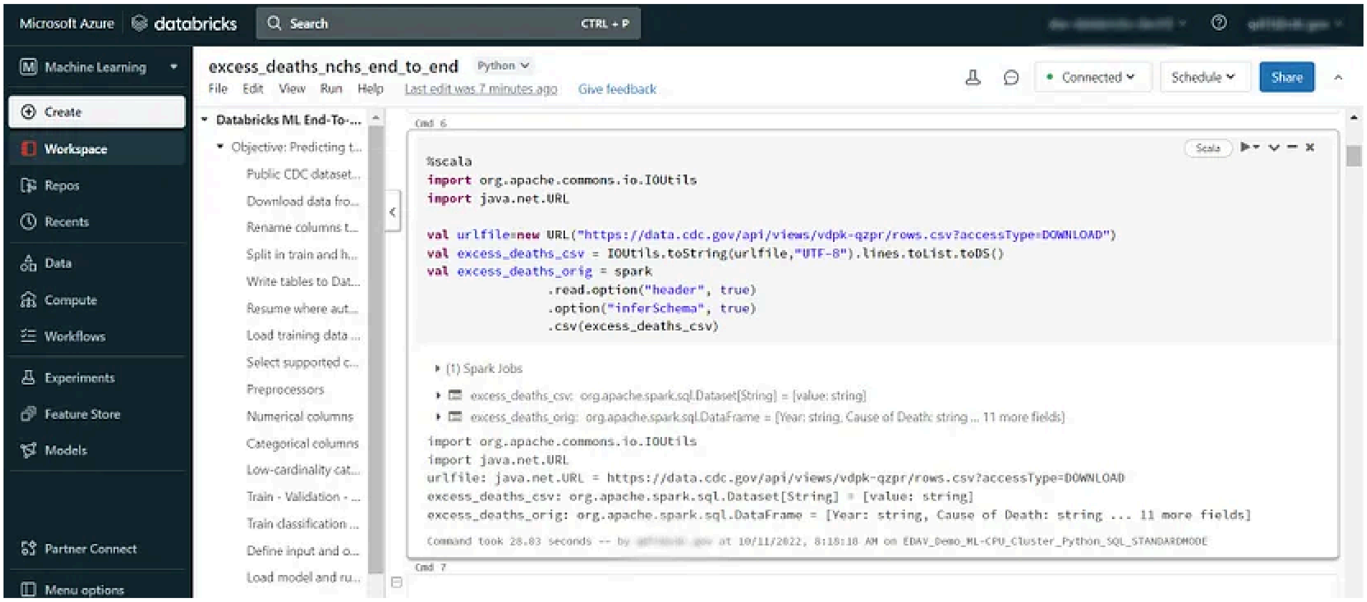
Art my daughter Nora produced at daycare

Hands on workflow how to build a Machine Learning XGBoost multi-class classification model all the way from real world data ingest (public CDC dataset) to model deployment and making real-time predictions. And once done, automate the whole enchilada MLOps-style.

The complete code to recreate the following demo yourself can be found in this GitHub repo: https://github.com/tjaensch/excess_deaths_nchs

Part 1: Ingest Data Into Databricks

Create a Python notebook in your Databricks workspace and attach it to a suitable Databricks ML cluster. I'm only using Scala in the following steps because it appeared to be the easiest way to get the data from the public CDC URL into a Spark dataframe without having to download files locally. This can be done in a Databricks Python notebook by using the Scala magic command `%scala` at the top of each cell with Scala code. All the actual Machine Learning code later on will be written in Python.



Download data from URL into a Spark dataframe:

```
%scala

import org.apache.commons.io.IOUtils
import java.net.URL

val urlfile=new URL("https://data.cdc.gov/api/views/vdpk-
qzpr/rows.csv?accessType=DOWNLOAD")
val excess_deaths_csv = IOUtils.toString(urlfile,"UTF-
8").lines.toList.toDS()
val excess_deaths_orig = spark
                .read.option("header", true)
                .option("inferSchema", true)
                .csv(excess_deaths_csv)
```

Rename columns to be able to save as Databricks table:

```
%scala

val excess_deaths = excess_deaths_orig.withColumnRenamed("Year",
"year")
                .withColumnRenamed("Cause of Death",
"cause_of_death")
                .withColumnRenamed("State", "state")
                .withColumnRenamed("State FIPS Code",
```

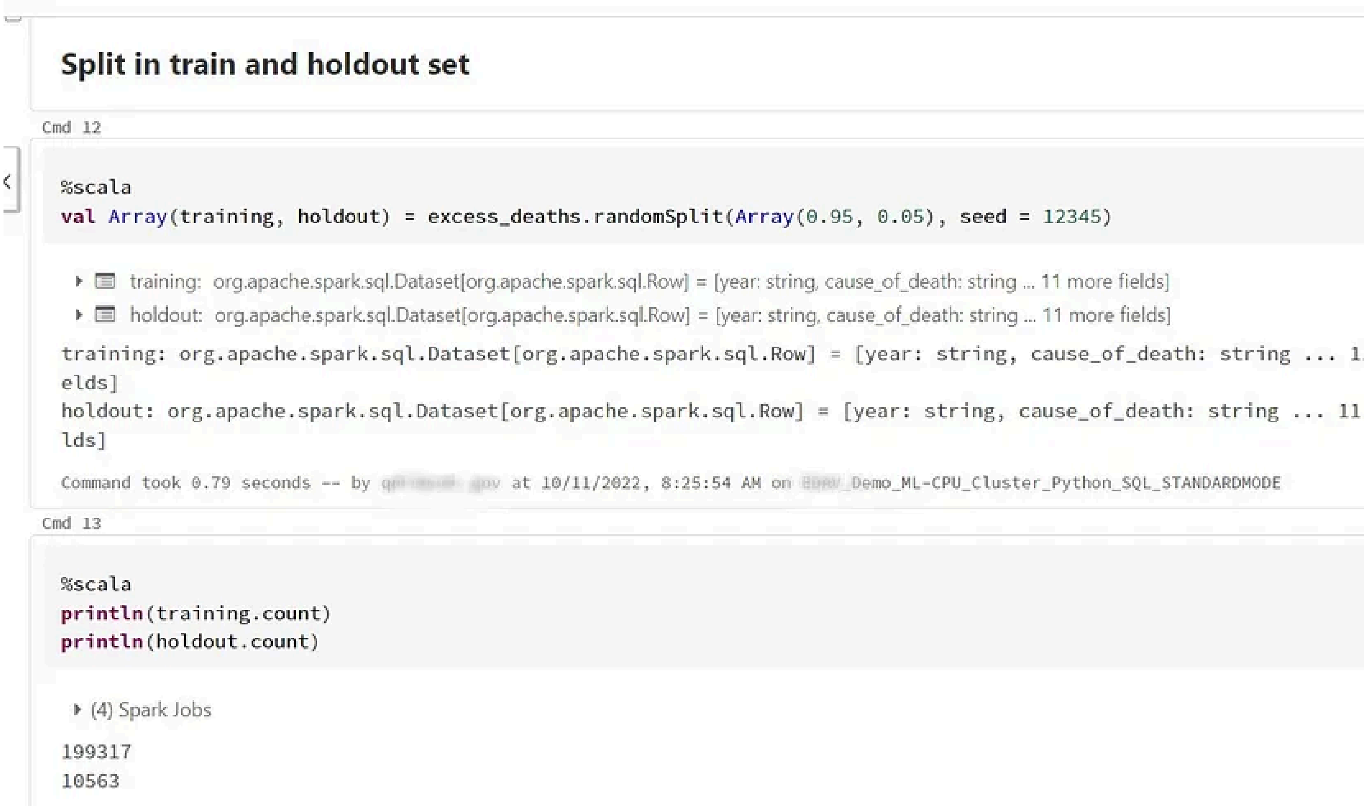
```
"state_fips_code")
                .withColumnRenamed("HHS Region", "hhs_region")
                .withColumnRenamed("Age Range", "age_range")
                .withColumnRenamed("Benchmark", "benchmark")
                .withColumnRenamed("Locality", "locality")
                .withColumnRenamed("Observed Deaths",
"observed_deaths")
                .withColumnRenamed("Population", "population")
                .withColumnRenamed("Expected Deaths",
"expected_deaths")
                .withColumnRenamed("Potentially Excess Deaths",
"potentially_excess_deaths")
                .withColumnRenamed("Percent Potentially Excess
Deaths", "percent_potentially_excess_deaths")
```

Split in training and holdout set:

```
%scala

val Array(training, holdout) = excess_deaths.randomSplit(Array(0.95,
0.05), seed = 12345)
```

I am going to use Databricks AutoML in the next step which does its own training/evaluation/test split so the above is mainly to have some data for testing (holdout) the best AutoML model after it has been created on data that the AutoML process has not seen at all yet.

At the time of writing there were 199317 examples in the training, and 10563 in the holdout dataset.

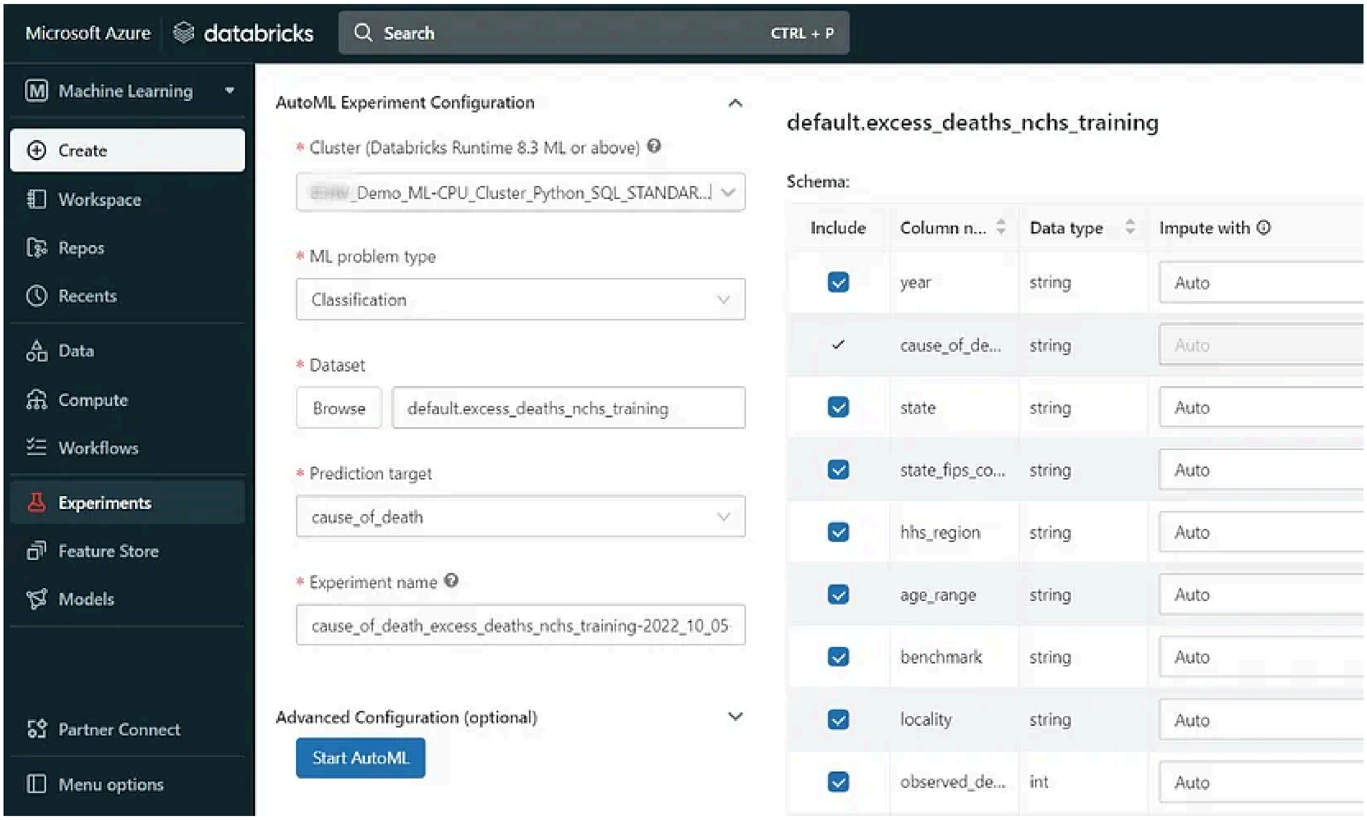


Write the Spark dataframes as tables to the Databricks default database to be used in later steps for training and testing:

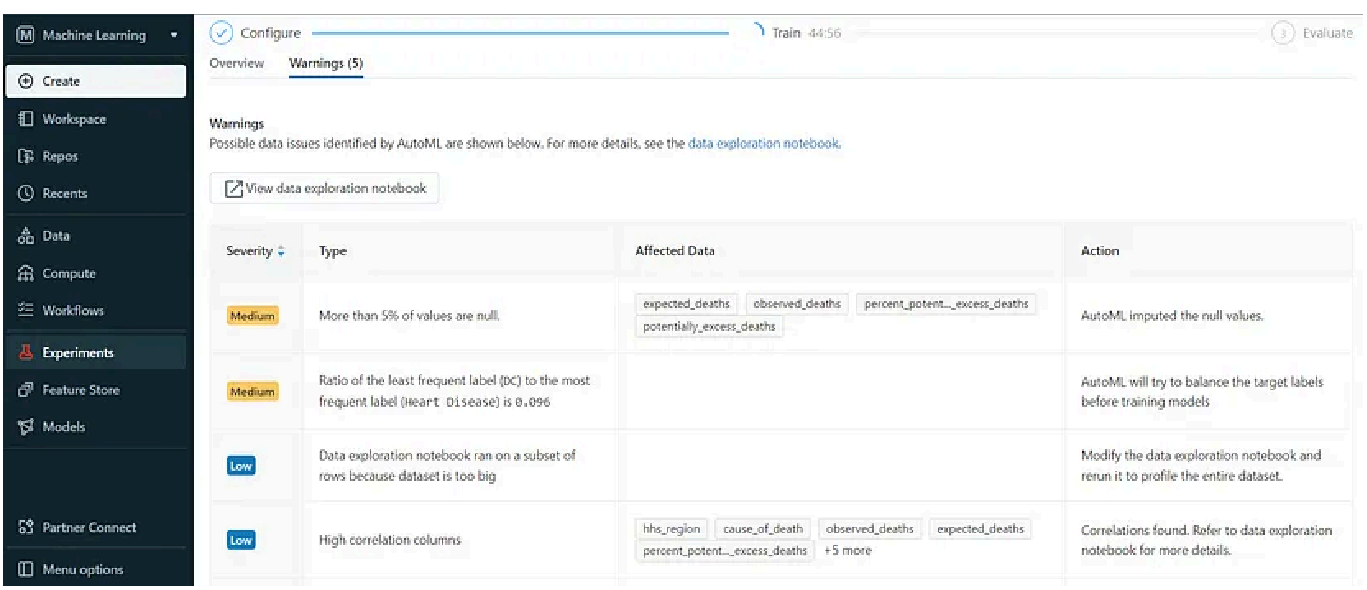
```
%scala

excess_deaths.write.mode("overwrite").format("parquet").saveAsTable("
default.excess_deaths_nchs_all")
training.write.mode("overwrite").format("parquet").saveAsTable("defau
lt.excess_deaths_nchs_training")
test.write.mode("overwrite").format("parquet").saveAsTable("default.e
xcess_deaths_nchs_holdout")
```

Part 2: Create AutoML Classification Experiment

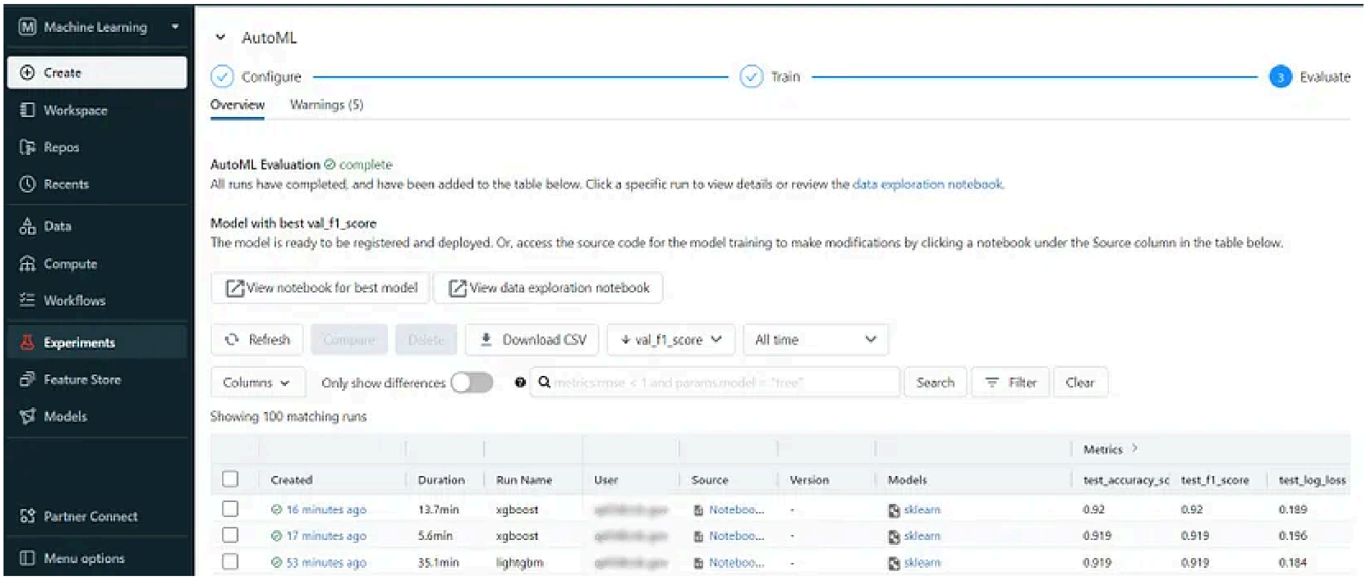
Instead of writing all the necessary ML code from scratch in a notebook to build a classification model, I am going to leverage Databricks' AutoML capabilities. The AutoML Experiment (see below) is easy to configure and will run dozens of trials with different algorithms and in the end produce one winning experiment that I am going to use to build the end-to-end MLOps pipeline.



As the experiment is running, AutoML is discovering data issues that are automatically being addressed.



Once the training run has finished after one hour which was the default timeout at the time of writing, the Experiments page will display a long list of trials it ran during the experiment with the best run at the top.



Now the resulting best model could be used as is for making predictions on new data, or the holdout data we saved previously, but the point of this post is to demonstrate how the code created by the best AutoML experiment can be reused to create a repeatable process that downloads fresh data each time from the CDC download URL, and runs the same code that worked best this time on this type of data as a scheduled process on new data in case the CDC updates the public dataset.

The neat thing about Databricks ML is that the AutoML experiments generate notebooks (under 'Source' in the image above) for each trial run that can be used to rerun the exact same trial with the same specs, inputs, etc., but that auto-generated code can also be used and enhanced which is what I am going to do. The auto-generated code for the best experiment I got at the time of writing can be found here.

The winning algorithm out of the trials that AutoML produced was, rather unsurprisingly, XGBoost. The AutoML process automatically did a bunch of preprocessing, imputing of missing values, scaling, one-hot-encoding, etc. without me having to do a thing at all, and it also did an internal train-validation-test split on the training data we fed into it above. 60 percent of the training dataset was then used for training the model, and 20 percent each for validation and testing. I am going to change this distribution later in the end-to-end notebook to 90 percent for training and 5 percent each for validation and testing since the more data for training the better the model metrics I achieved.

The hyperparameters of the best XGBoost model produced look as follows:

```
xgbc_classifier = TransformedTargetClassifier(
    classifier=XGBClassifier(
        colsample_bytree=0.6925102629069848,
        learning_rate=0.16153054387601637,
        max_depth=8,
        min_child_weight=6,
        n_estimators=1573,
        n_jobs=100,
        subsample=0.7774586806850589,
        verbosity=0,
        random_state=538597927,
    ),
    transformer=LabelEncoder()  # XGBClassifier requires the target
values to be integers between 0 and n_class-1
)
```

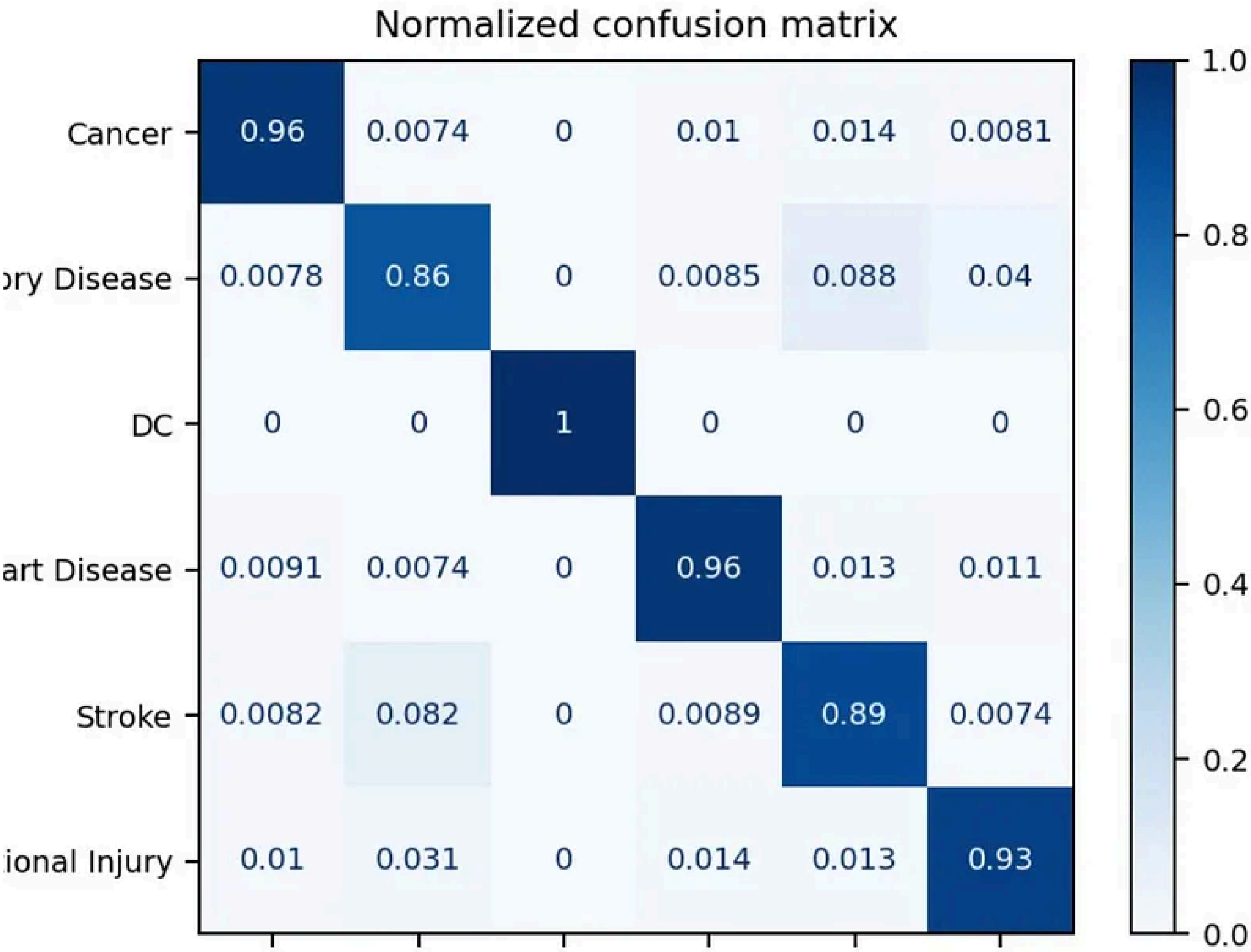
AutoML also generates MLFlow code automatically that tracks all experiments and models in Databricks and can be reused as well going forward. I didn't have to write any of this myself:

```
# Enable automatic logging of input samples, metrics, parameters, and models
mlflow.sklearn.autolog(log_input_examples=True, silent=True)

with mlflow.start_run(experiment_id="1860757565312910", run_name="xgboost") as mlflow_run:
    # AutoML balanced the data internally and use _automl_sample_weight_4cd4 to calibrate the probability distribution
    xgbc_sample_weight = X_train.loc[:, "_automl_sample_weight_4cd4"].to_numpy()

    model.fit(X_train, y_train, classifier__early_stopping_rounds=5, classifier__verbose=False, classifier__eval_set=[(X_val_processed,y_val_process

    # Log metrics for the training set
    xgbc_training_metrics = mlflow.sklearn.eval_and_log_metrics(model, X_train, y_train, prefix="training_", sample_weight=xgbc_sample_weight)

    # Log metrics for the validation set
    xgbc_val_metrics = mlflow.sklearn.eval_and_log_metrics(model, X_val, y_val, prefix="val_")

    # Log metrics for the test set
    xgbc_test_metrics = mlflow.sklearn.eval_and_log_metrics(model, X_test, y_test, prefix="test_")

    # Display the logged metrics
    xgbc_val_metrics = {k.replace("val_", ""): v for k, v in xgbc_val_metrics.items()}
    xgbc_test_metrics = {k.replace("test_", ""): v for k, v in xgbc_test_metrics.items()}
    display(pd.DataFrame([xgbc_val_metrics, xgbc_test_metrics], index=["validation", "test"]))
```

```
/databricks/python/lib/python3.9/site-packages/xgboost/sklearn.py:1224: UserWarning: The use of label encoder in XGBClassifier is deprecated and wil
l be removed in a future release. To remove this warning, do the following: 1) Pass option use_label_encoder=False when constructing XGBClassifier o
bject; and 2) Encode your labels (y) as integers starting with 0, i.e. 0, 1, 2, ..., [num_class - 1].
  warnings.warn(label_encoder_deprecation_msg, UserWarning)
2022/10/05 20:20:26 WARNING mlflow.sklearn.utils: roc_auc_score failed. The metric training_roc_auc_score will not be recorded. Metric error: sample
_weight is not supported for multiclass one-vs-one ROC AUC, 'sample_weight' must be None in this case.
```

	precision_score	recall_score	f1_score	accuracy_score	log_loss	roc_auc_score	score
validation	0.921369	0.921325	0.921315	0.921325	0.187503	0.995584	0.921325

In addition to that, the auto-generation also includes code to determine feature importance, stubs how to register and load the generated model, and even a confusion matrix generated on the validation data AutoML produced internally:



Part 3: Build End-To-End MLOps Workflow From Data Ingest To Model Deployment

What I did next is basically reuse the auto-generated model code from the best AutoML trial and enhance it with data ingest and model deployment code so the resulting notebook here can be run from start to finish and that way each time pull fresh data from the public CDC URL, save it to the Databricks default database, train an XGBoost model with the best hyperparameters from the top AutoML run, do batch inference on the holdout dataset, and in case the batch predictions are better than the 90 percent success threshold, automatically deploy the model into the Databricks Model Registry so it can be used either for batch predictions or real-time model serving via a REST API interface.

Since early stopping is enabled on the MLFlow experiment, model training now is much quicker since it doesn't improve anymore on the previously determined best hyperparameters and just stops right there, and the model metrics look like this:

```
# Log metrics for the test set
xgbc_test_metrics = mlflow.sklearn.eval_and_log_metrics(model, X_test, y_test, prefix="test_")

# Display the logged metrics
xgbc_val_metrics = {k.replace("val_", ""): v for k, v in xgbc_val_metrics.items()}
xgbc_test_metrics = {k.replace("test_", ""): v for k, v in xgbc_test_metrics.items()}
display(pd.DataFrame([xgbc_val_metrics, xgbc_test_metrics], index=["validation", "test"]))
```

▾ (1) MLflow run

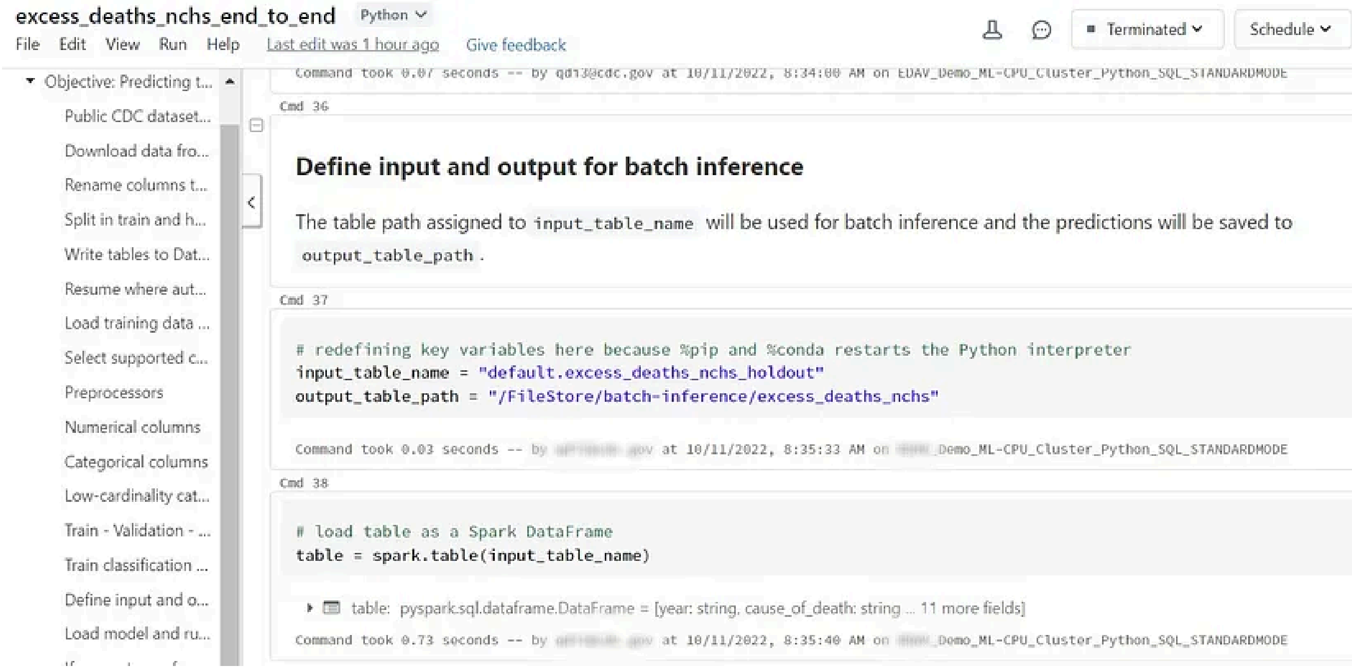
Logged 1 run to an experiment in MLflow. Learn more

```
/databricks/python/lib/python3.9/site-packages/xgboost/sklearn.py:1224: UserWarning: The use of label encoder in XGBCla
ssifier is deprecated and will be removed in a future release. To remove this warning, do the following: 1) Pass option
use_label_encoder=False when constructing XGBClassifier object; and 2) Encode your labels (y) as integers starting with
0, i.e. 0, 1, 2, ..., [num_class - 1].
  warnings.warn(label_encoder_deprecation_msg, UserWarning)
```

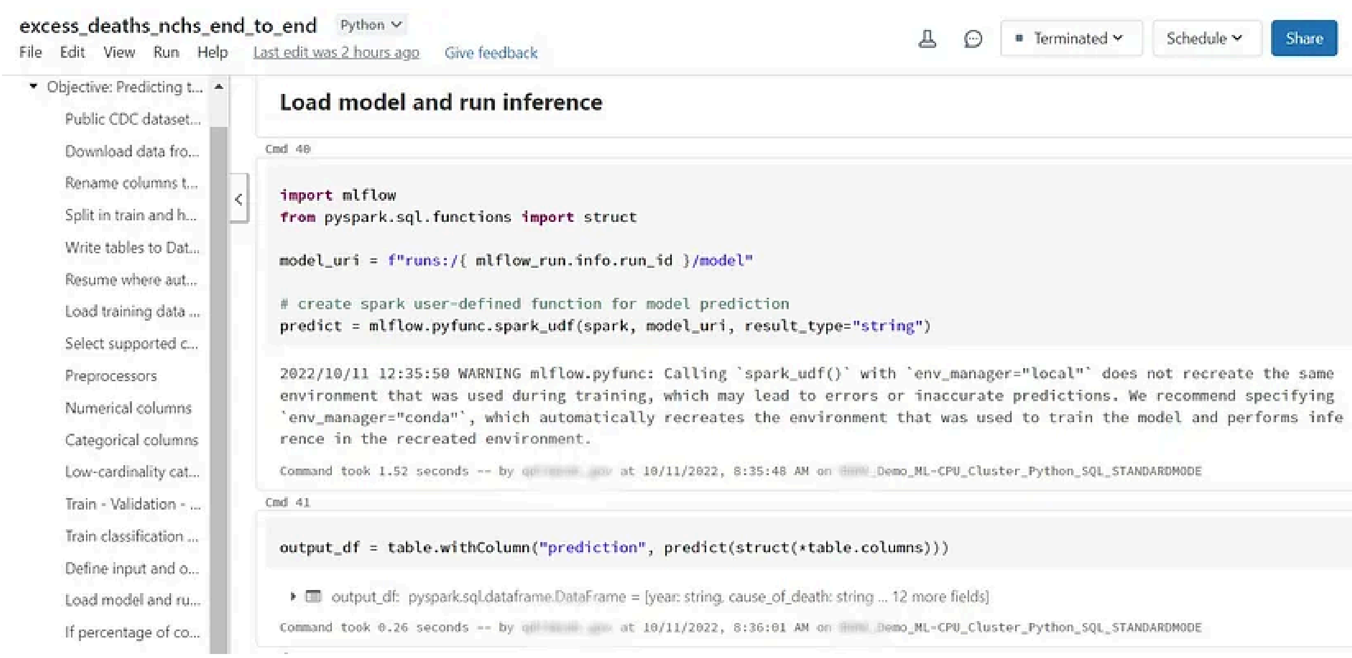
Table Data Profile

	precision_score	recall_score	f1_score	accuracy_score	log_loss	roc_auc_score
1	0.9175218139817276	0.9177723269037796	0.9176010479449093	0.9177723269037796	0.20029566126041656	0.9955439093436397
2	0.9281755783860051	0.928284854563691	0.9280877351868554	0.928284854563691	0.18478242467713377	0.9964303373217934

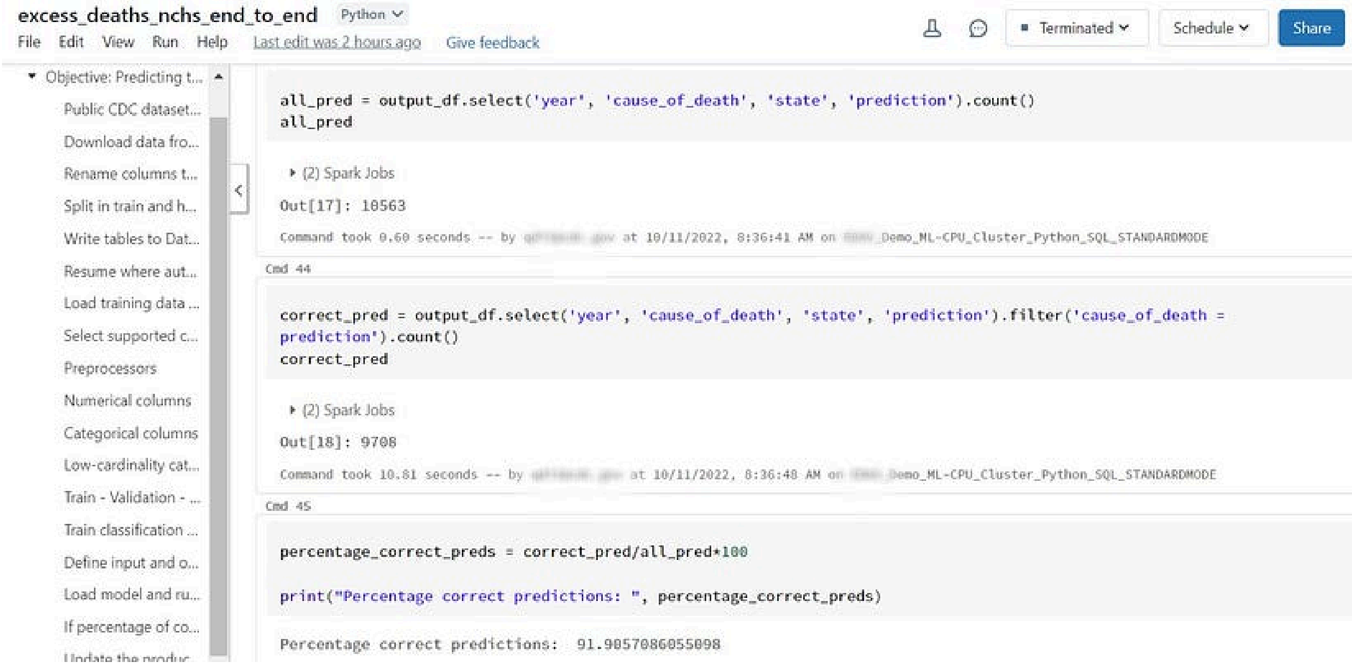
Next I am defining input and output for batch inference on the holdout dataset:



Load the generated model and run batch inference on the holdout dataset:

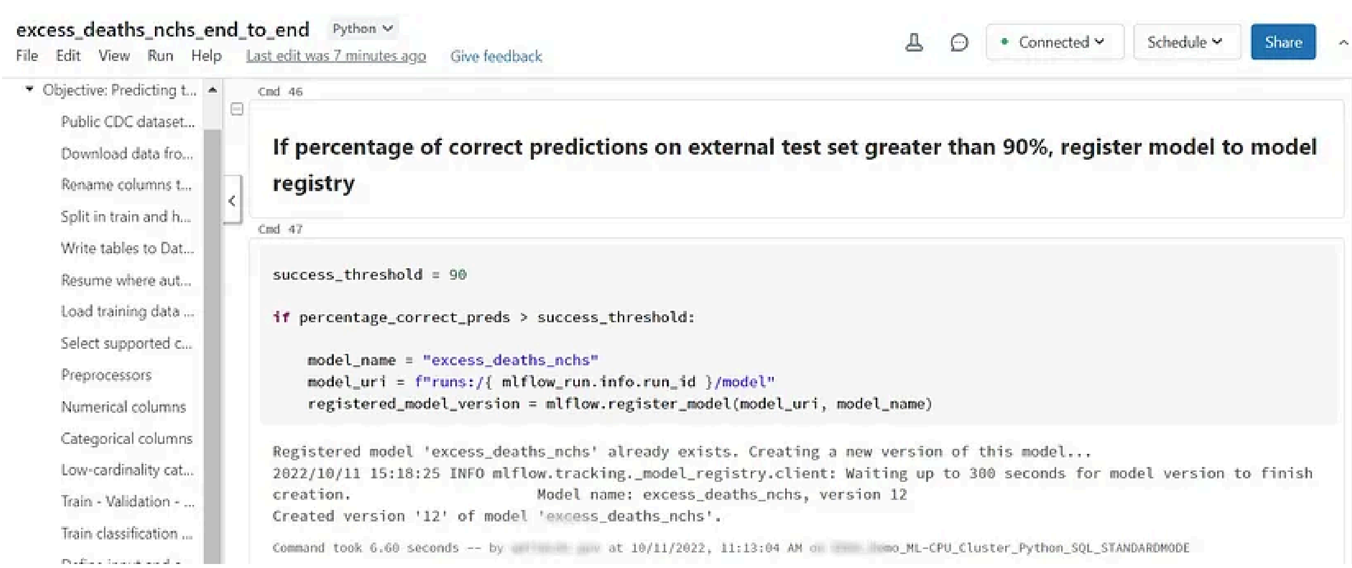


At the time of writing this post, the holdout dataset contained 10563 examples and the generated model predicted 9708 causes of death correctly which comes down to about 91.9% correct predictions:



Considering that this is multi-class classification and there are 5 possible options for causes of death that is a pretty solid outcome, given the minimal amount of effort regarding data preparation and cleansing, feature selection, feature engineering, etc.

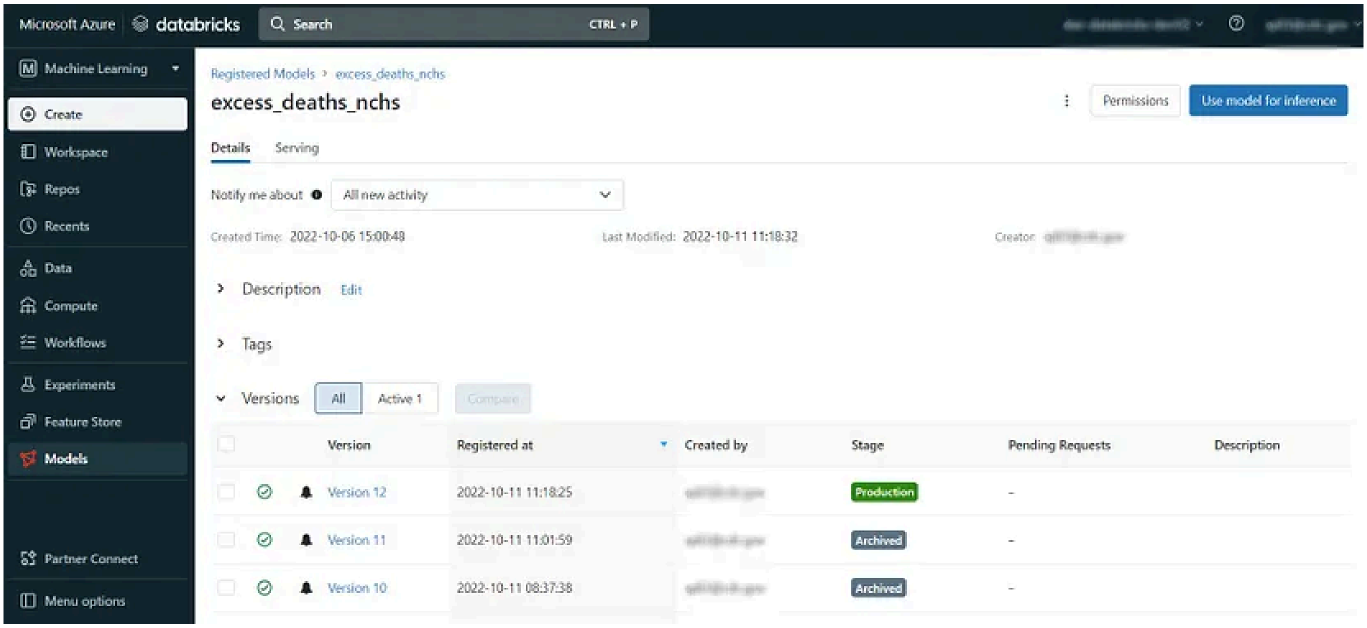
In the end-to-end notebook I then added code to register the model to the Databricks Model Registry in case the correct predictions on the holdout dataset are greater than 90 percent.



If a registered model of the same name already exists (as pictured above), a new version of the model will automatically be created and all previous model versions will still be in there as well.

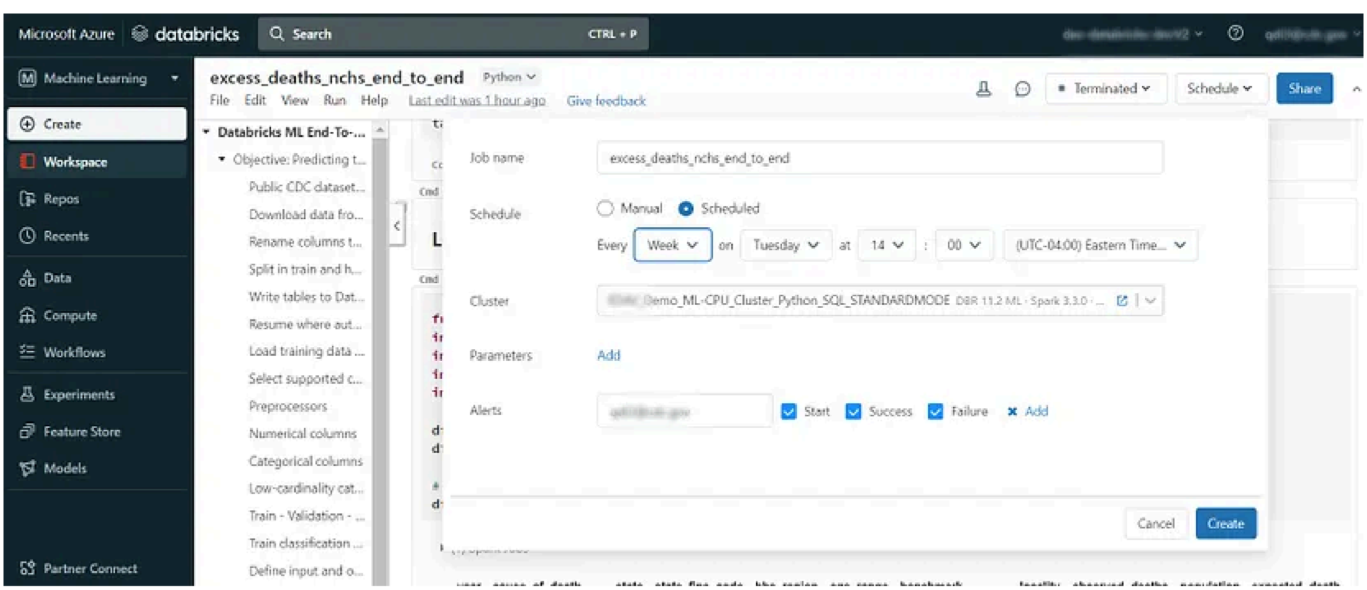
Finally, promote the newly minted model to 'Production' stage and set the previous one to 'Archived'.

In Databricks Model Registry that will look something like this:

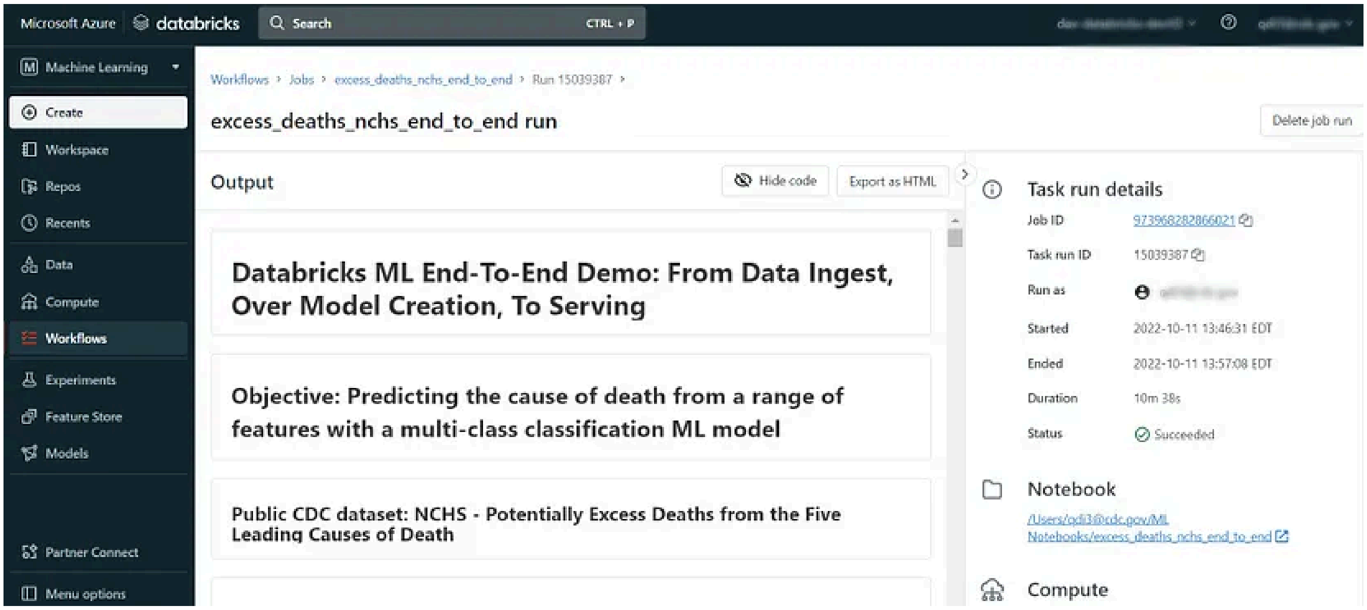


Part 4: Schedule MLOps Pipeline And Make Real-Time Predictions

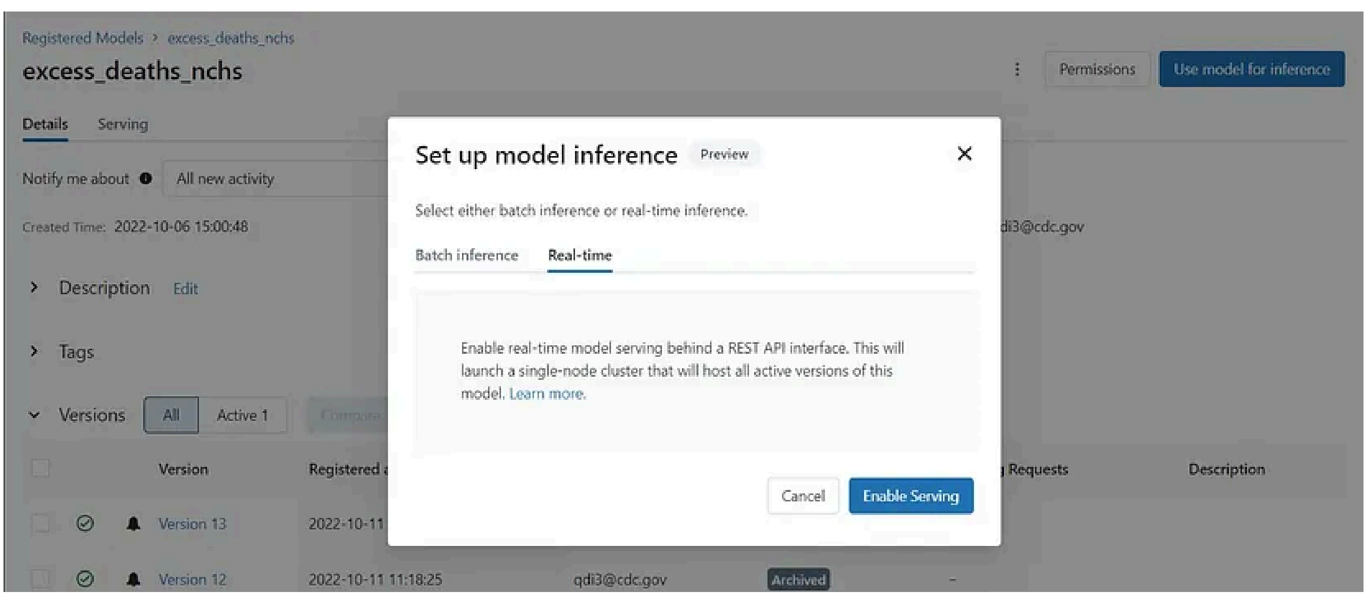
Now, to make this an automated, repeatable process that will build and deploy a new model on fresh data in case the CDC updates the source dataset and the new model surpasses the set threshold of 90 percent correct predictions on the new holdout dataset, all that needs to be done is to schedule a Databricks Workflow which can be achieved on the notebook page view by clicking on 'Schedule':



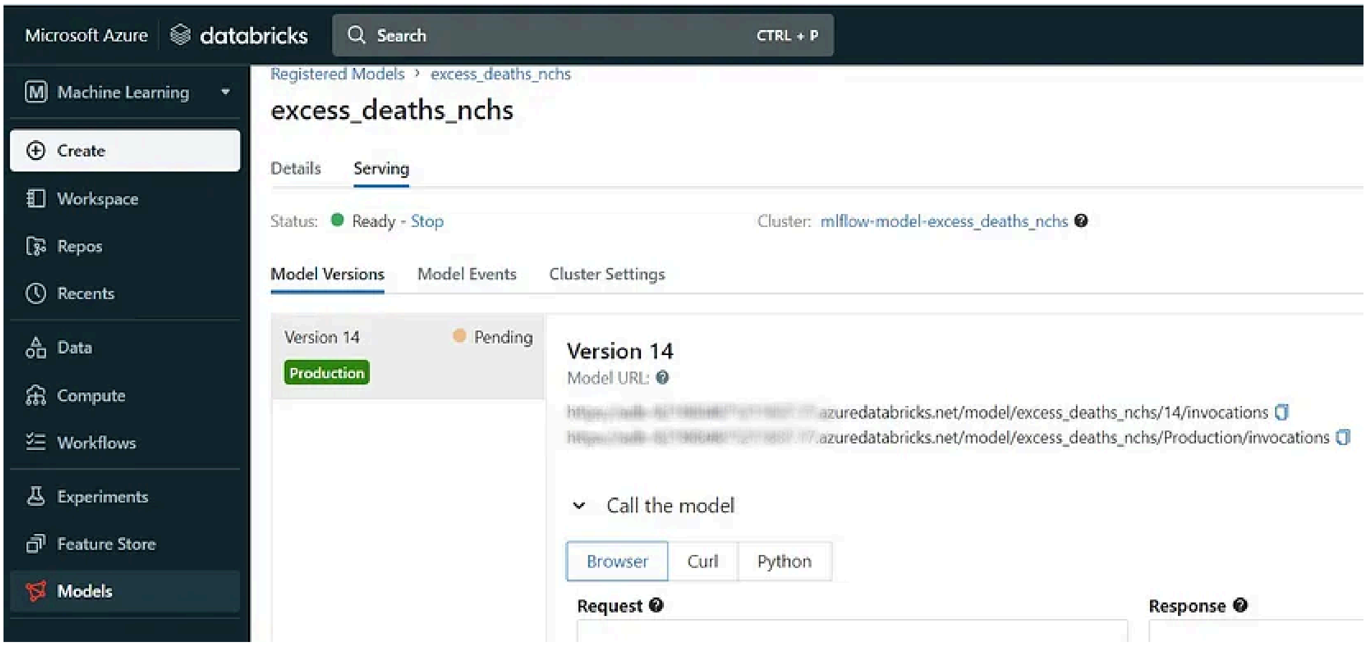
When a run has finished and succeeded, you will receive an email with a link to the task run details:



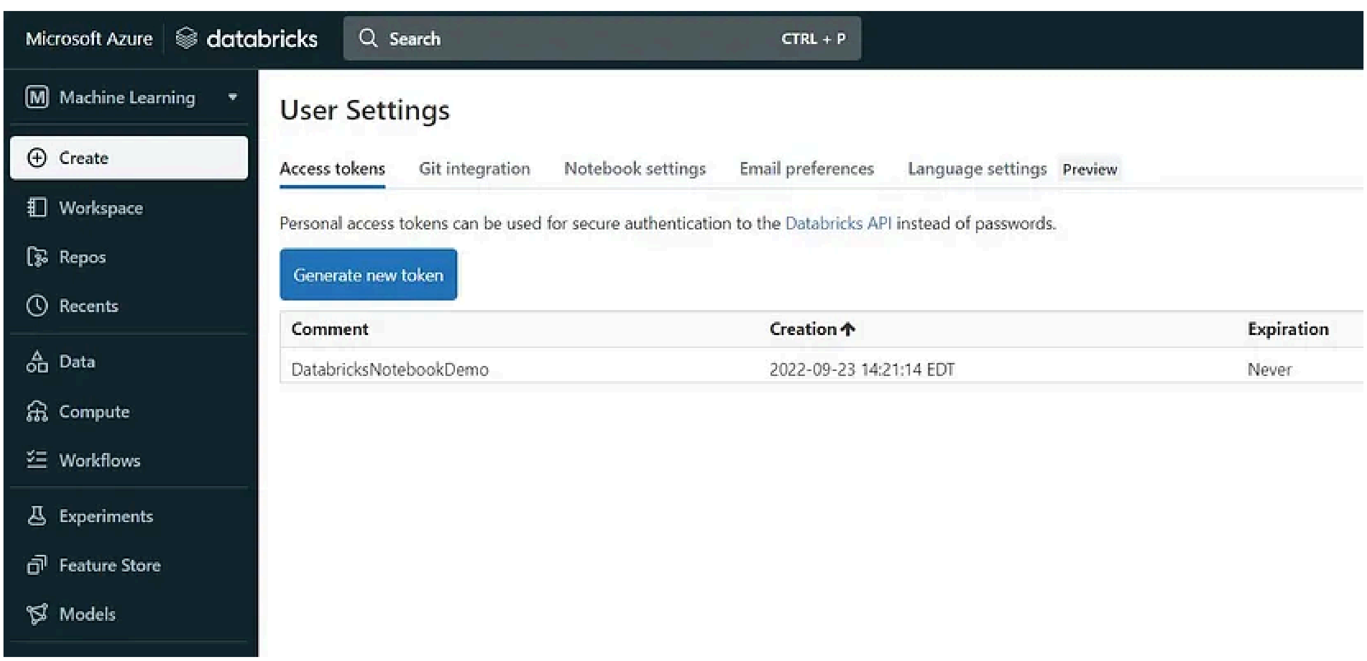
To use the 'Production' version of the model for real-time inference, serving needs to be enabled:



This will take a little while to deploy and once ready look as follows:



To make real-time predictions through the Databricks REST API using the deployed model whose version is going to update automatically each time a new model is created that surpasses the 90 percent success threshold, a Databricks token is required. The token can be created through the User Settings page:



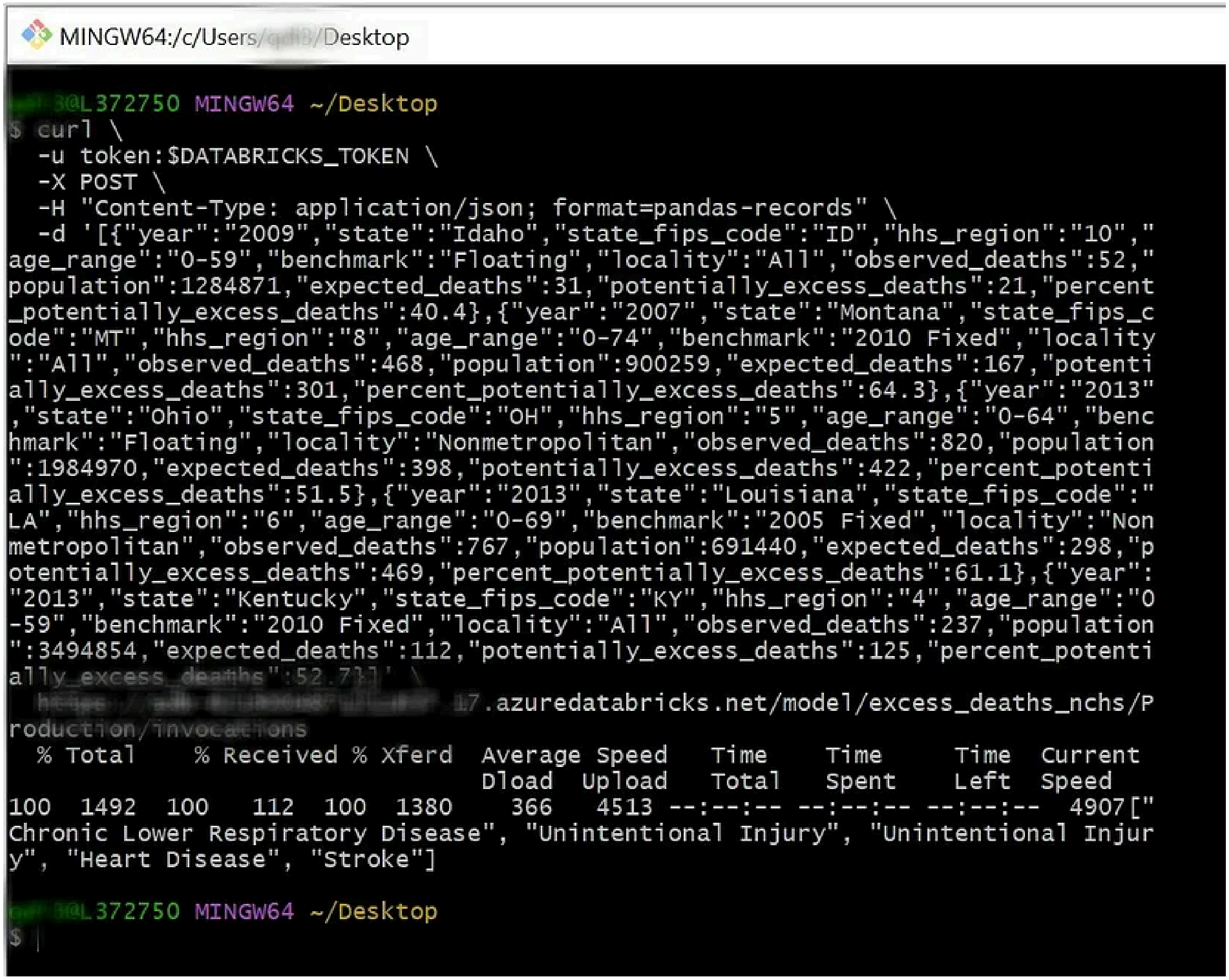
In a terminal, add the token to your environment variables:

Make predictions using cURL and the following example command (you will have to update the model URL at the end to your own):

```
curl \
  -u token:$DATABRICKS_TOKEN \
  -X POST \
  -H "Content-Type: application/json; format=pandas-records" \
  -d
'[{"year":"2009","state":"Idaho","state_fips_code":"ID","hhs_region":
"10","age_range":"0-
59","benchmark":"Floating","locality":"All","observed_deaths":52,"pop
ulation":1284871,"expected_deaths":31,"potentially_excess_deaths":21,
"percent_potentially_excess_deaths":40.4},
{"year":"2007","state":"Montana","state_fips_code":"MT","hhs_region":
"8","age_range":"0-74","benchmark":"2010
Fixed","locality":"All","observed_deaths":468,"population":900259,"ex
pected_deaths":167,"potentially_excess_deaths":301,"percent_potential
ly_excess_deaths":64.3},
{"year":"2013","state":"Ohio","state_fips_code":"OH","hhs_region":"5"
,"age_range":"0-
64","benchmark":"Floating","locality":"Nonmetropolitan","observed_dea
ths":820,"population":1984970,"expected_deaths":398,"potentially_exce
ss_deaths":422,"percent_potentially_excess_deaths":51.5},
{"year":"2013","state":"Louisiana","state_fips_code":"LA","hhs_region
":"6","age_range":"0-69","benchmark":"2005
Fixed","locality":"Nonmetropolitan","observed_deaths":767,"population
":691440,"expected_deaths":298,"potentially_excess_deaths":469,"perce
nt_potentially_excess_deaths":61.1},
{"year":"2013","state":"Kentucky","state_fips_code":"KY","hhs_region"
:"4","age_range":"0-59","benchmark":"2010
Fixed","locality":"All","observed_deaths":237,"population":3494854,"e
xpected_deaths":112,"potentially_excess_deaths":125,"percent_potentia
lly_excess_deaths":52.7}]' \
  https://adb-
```

XXX.azuredatabricks.net/model/excess_deaths_nchs/Production/invocations

The results will be returned as a list:



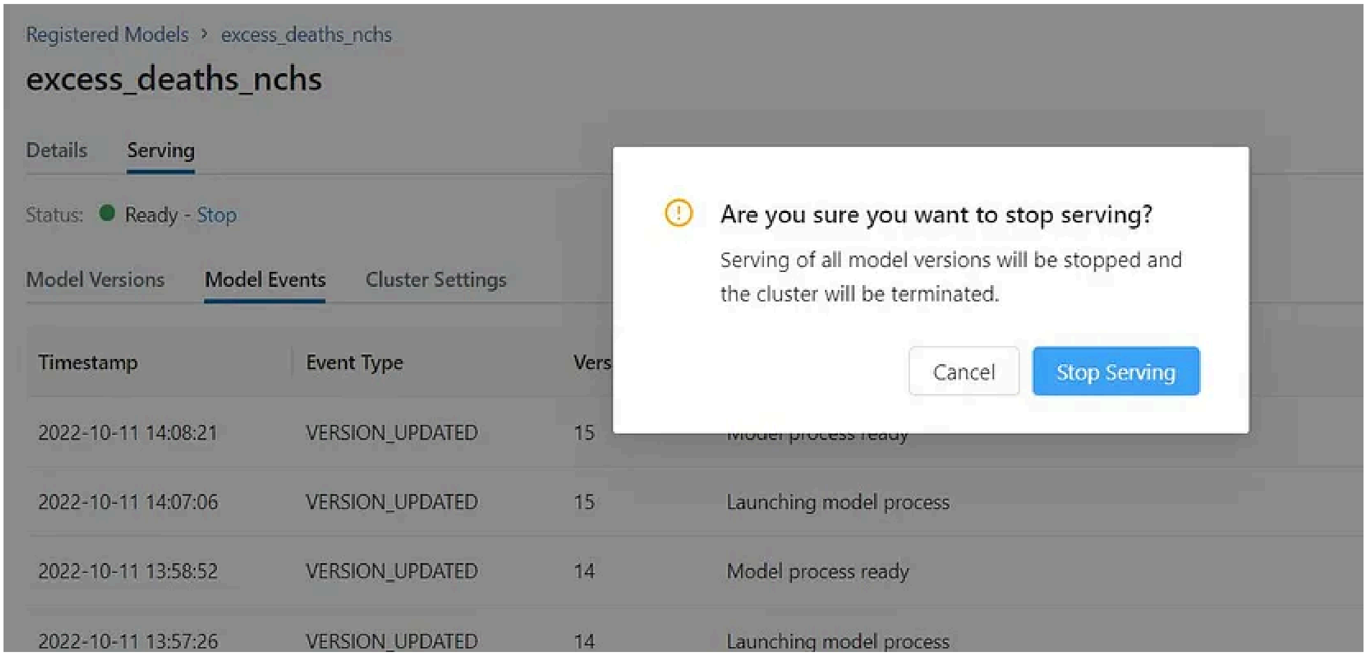
Predictions can also be made directly in the browser or using Python.

Call the model

Browser | Curl | Python

```
import os
import requests
import numpy as np
import pandas as pd
import json

def create_tf_serving_json(data):
  return {'inputs': {name: data[name].tolist() for name in data.keys()} if isinstance(data, dict) else data.tolist()}

def score_model(dataset):
  url = 'https://[illegible].azuredatabricks.net/model/excess_deaths_nchs/15/invocations'
  headers = {'Authorization': f'Bearer {os.environ.get("DATABRICKS_TOKEN")}', 'Content-Type': 'application/json'}
  ds_dict = dataset.to_dict(orient='split') if isinstance(dataset, pd.DataFrame) else create_tf_serving_json(dataset)
  data_json = json.dumps(ds_dict, allow_nan=True)
  response = requests.request(method='POST', headers=headers, url=url, data=data_json)
  if response.status_code != 200:
    raise Exception(f'Request failed with status {response.status_code}, {response.text}')
  return response.json()
```

A version of a model with the status 'Production' cannot be deleted before the status has been changed to e.g. 'Archived', but the model server can be stopped anytime the model is not needed for real-time serving:



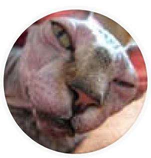
If you need it again, just enable serving once again.

As I set it up above with the scheduled Databricks Workflow, the data-ingest-to-model-deployment notebook would now run on a weekly basis, download potentially updated source data from the CDC website, build a new XGBoost model with the new data, and test it on new data as well. Only if the predictions on the holdout data are better than 90 percent correct the new model will be deployed to production. In any case I would receive an email each time the process runs and if it succeeded or failed, and all the created model artifacts would be available for inspection and troubleshooting in case the predictive power of the model declines below the success threshold, or any other issues or problems arise.

Bottom Line

Databricks provides all the tools to develop an end-to-end Machine Learning solution with relative ease. There are certainly limitations (e.g. currently Databricks AutoML only works on tabular data, IDE support to be able to develop locally is limited and difficult to set up, one still has to create and manage and wait on Spark clusters to become ready, just to name a few…), but overall it is a nice environment with a pleasant UI to build and manage Machine Learning code and artifacts.

Machine Learning Databricks Mlops Python Spark



Written by Thomas Jaensch

WFH, staring at code

More from Thomas Jaensch