Training machine learning models on tabular data: an end-to-end example

This tutorial covers the following steps:

- Import data from your local machine into the Databricks File System (DBFS)
- Visualize the data using Seaborn and matplotlib
- Run a parallel hyperparameter sweep to train machine learning models on the dataset
- Explore the results of the hyperparameter sweep with MLflow
- Register the best performing model in MLflow
- Apply the registered model to another dataset using a Spark UDF
- Set up model serving for low-latency requests

In this example, you build a model to predict the quality of Portugese "Vinho Verde" wine based on the wine's physicochemical properties.

The example uses a dataset from the UCI Machine Learning Repository, presented in *Modeling wine preferences by data mining from physicochemical properties* (https://www.sciencedirect.com/science/article/pii/S0167923609001377?via%3Dihub) [Cortez et al., 2009].

Requirements

This notebook requires Databricks Runtime for Machine Learning.
If you are using Databricks Runtime 7.3 LTS ML or below, you must update the CloudPickle library. To do that, uncomment and run the `%pip install` command in Cmd 2.

```
# This command is only required if you are using a cluster running DBR 7.3 LTS ML or below.
#%pip install --upgrade cloudpickle
```

Import data

In this section, you download a dataset from the web and upload it to Databricks File System (DBFS).

1. Navigate to https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/ (https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/) and download both `winequality-red.csv` and `winequality-white.csv` to your local machine.

2. From this Databricks notebook, select *File* > *Upload Data*, and drag these files to the drag-and-drop target to upload them to the Databricks File System (DBFS).

   Note: if you don't have the *File* > *Upload Data* option, you can load the dataset from the Databricks example datasets. Uncomment and run the last two lines in the following cell.

3. Click *Next*. Some auto-generated code to load the data appears. Select *pandas*, and copy the example code.

4. Create a new cell, then paste in the sample code. It will look similar to the code shown in the following cell. Make these changes:
   - Pass `sep=';'` to `pd.read_csv`
   - Change the variable names from `df1` and `df2` to `white_wine` and `red_wine`, as shown in the following cell.

```
# If you have the File > Upload Data menu option, follow the instructions in
the previous cell to upload the data from your local machine.
# The generated code, including the required edits described in the previous
cell, is shown here for reference.

import pandas as pd

# In the following lines, replace <username> with your username.
white_wine =
pd.read_csv("/dbfs/FileStore/shared_uploads/<username>/winequality_white.csv",
sep=';')
red_wine =
pd.read_csv("/dbfs/FileStore/shared_uploads/<username>/winequality_red.csv",
sep=';')

# If you do not have the File > Upload Data menu option, uncomment and run
these lines to load the dataset.

#white_wine = pd.read_csv("/dbfs/databricks-datasets/wine-quality/winequality-
white.csv", sep=";")
#red_wine = pd.read_csv("/dbfs/databricks-datasets/wine-quality/winequality-
red.csv", sep=";")
```

Merge the two DataFrames into a single dataset, with a new binary feature "is_red" that indicates whether the wine is red or white.

```
red_wine['is_red'] = 1
white_wine['is_red'] = 0

data = pd.concat([red_wine, white_wine], axis=0)

# Remove spaces from column names
data.rename(columns=lambda x: x.replace(' ', '_'), inplace=True)
```

```
data.head()
```

Out[33]:

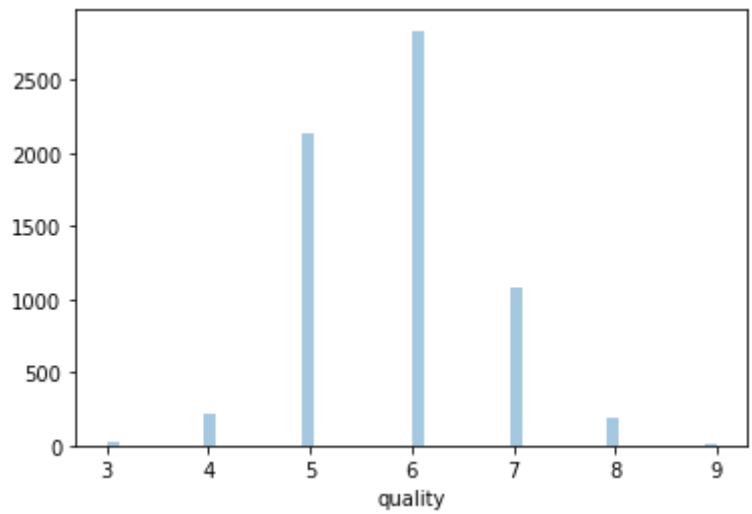
Visualize data

Before training a model, explore the dataset using Seaborn and Matplotlib.

Plot a histogram of the dependent variable, quality.

```
import seaborn as sns
sns.distplot(data.quality, kde=False)
```

```
/databricks/python/lib/python3.8/site-packages/seaborn/distributions.py:2557:
FutureWarning: `distplot` is a deprecated function and will be removed in a fu
ture version. Please adapt your code to use either `displot` (a figure-level f
unction with similar flexibility) or `histplot` (an axes-level function for hi
stograms).
  warnings.warn(msg, FutureWarning)
Out[34]:
```




```
<AxesSubplot:xlabel='quality'>
```

Looks like quality scores are normally distributed between 3 and 9.

Define a wine as high quality if it has quality >= 7.

```
high_quality = (data.quality >= 7).astype(int)
data.quality = high_quality
```

Box plots are useful in noticing correlations between features and a binary label.

```
import matplotlib.pyplot as plt

dims = (3, 4)

f, axes = plt.subplots(dims[0], dims[1], figsize=(25, 15))
axis_i, axis_j = 0, 0
for col in data.columns:
  if col == 'is_red' or col == 'quality':
    continue # Box plots cannot be used on indicator variables
  sns.boxplot(x=high_quality, y=data[col], ax=axes[axis_i, axis_j])
  axis_j += 1
  if axis_j == dims[1]:
    axis_i += 1
    axis_j = 0
```

In the above box plots, a few variables stand out as good univariate predictors of quality.

- In the alcohol box plot, the median alcohol content of high quality wines is greater than even the 75th quantile of low quality wines. High alcohol content is correlated with quality.
- In the density box plot, low quality wines have a greater density than high quality wines. Density is inversely correlated with quality.

Preprocess data

Prior to training a model, check for missing values and split the data into training and validation sets.

```
data.isna().any()
```

```
Out[37]: fixed_acidity           False
volatile_acidity        False
citric_acid             False
residual_sugar          False
chlorides               False
free_sulfur_dioxide     False
total_sulfur_dioxide    False
density                 False
pH                      False
sulphates               False
alcohol                 False
quality                 False
is_red                  False
dtype: bool
```

There are no missing values.

Prepare dataset for training baseline model

Split the input data into 3 sets:

- Train (60% of the dataset used to train the model)
- Validation (20% of the dataset used to tune the hyperparameters)
- Test (20% of the dataset used to report the true performance of the model on an unseen dataset)

```
from sklearn.model_selection import train_test_split

X = data.drop(["quality"], axis=1)
y = data.quality

# Split out the training data
X_train, X_rem, y_train, y_rem = train_test_split(X, y, train_size=0.6,
random_state=123)

# Split the remaining data equally into validation and test
X_val, X_test, y_val, y_test = train_test_split(X_rem, y_rem, test_size=0.5,
random_state=123)
```

Build a baseline model

This task seems well suited to a random forest classifier, since the output is binary and there may be interactions between multiple variables.

The following code builds a simple classifier using scikit-learn. It uses MLflow to keep track of the model accuracy, and to save the model for later use.

```
import mlflow
import mlflow.pyfunc
import mlflow.sklearn
import numpy as np
import sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score
from mlflow.models.signature import infer_signature
from mlflow.utils.environment import _mlflow_conda_env
import cloudpickle
import time

# The predict method of sklearn's RandomForestClassifier returns a binary
classification (0 or 1).
# The following code creates a wrapper function, SklearnModelWrapper, that uses
# the predict_proba method to return the probability that the observation
belongs to each class.

class SklearnModelWrapper(mlflow.pyfunc.PythonModel):
  def __init__(self, model):
    self.model = model

  def predict(self, context, model_input):
    return self.model.predict_proba(model_input)[:,1]

# mlflow.start_run creates a new MLflow run to track the performance of this
model.
# Within the context, you call mlflow.log_param to keep track of the parameters
used, and
# mlflow.log_metric to record metrics like accuracy.
with mlflow.start_run(run_name='untuned_random_forest'):
  n_estimators = 10
  model = RandomForestClassifier(n_estimators=n_estimators,
random_state=np.random.RandomState(123))
  model.fit(X_train, y_train)

  # predict_proba returns [prob_negative, prob_positive], so slice the output
with [:, 1]
  predictions_test = model.predict_proba(X_test)[:,1]
  auc_score = roc_auc_score(y_test, predictions_test)
  mlflow.log_param('n_estimators', n_estimators)
  # Use the area under the ROC curve as a metric.
  mlflow.log_metric('auc', auc_score)
  wrappedModel = SklearnModelWrapper(model)
  # Log the model with a signature that defines the schema of the model's
inputs and outputs.
```

```
  # When the model is deployed, this signature will be used to validate inputs.
  signature = infer_signature(X_train, wrappedModel.predict(None, X_train))

  # MLflow contains utilities to create a conda environment used to serve
models.
  # The necessary dependencies are added to a conda.yaml file which is logged
along with the model.
  conda_env =  _mlflow_conda_env(
        additional_conda_deps=None,
        additional_pip_deps=["cloudpickle=={}".format(cloudpickle.__version__),
"scikit-learn=={}".format(sklearn.__version__)],
        additional_conda_channels=None,
    )
  mlflow.pyfunc.log_model("random_forest_model", python_model=wrappedModel,
conda_env=conda_env, signature=signature)
```

```
/databricks/python/lib/python3.8/site-packages/mlflow/models/signature.py:129:
UserWarning: Hint: Inferred schema contains integer column(s). Integer columns
in Python cannot represent missing values. If your input data contains missing
values at inference time, it will be encoded as floats and will cause a schema
enforcement error. The best way to avoid this problem is to infer the model sc
hema based on a realistic data sample (training dataset) that includes missing
values. Alternatively, you can declare integer columns as doubles (float64) wh
enever these columns may have missing values. See `Handling Integers With Miss
ing Values <https://www.mlflow.org/docs/latest/models.html#handling-integers-w
ith-missing-values>`_ for more details.
  inputs = _infer_schema(model_input)
```

Examine the learned feature importances output by the model as a sanity-check.

```
feature_importances = pd.DataFrame(model.feature_importances_,
index=X_train.columns.tolist(), columns=['importance'])
feature_importances.sort_values('importance', ascending=False)
```

Out[40]:

	importance
alcohol	0.160192
density	0.117415
volatile_acidity	0.093136
chlorides	0.086618
residual_sugar	0.082544
free_sulfur_dioxide	0.080473
pH	0.080212
total_sulfur_dioxide	0.077798
sulphates	0.075780
citric_acid	0.071857
fixed_acidity	0.071841
is_red	0.002134

As illustrated by the boxplots shown previously, both alcohol and density are important in predicting quality.

You logged the Area Under the ROC Curve (AUC) to MLflow. Click **Experiment** at the upper right to display the Experiment Runs sidebar.

The model achieved an AUC of 0.854.

A random classifier would have an AUC of 0.5, and higher AUC values are better. For more information, see Receiver Operating Characteristic Curve (https://en.wikipedia.org/wiki/Receiver_operating_characteristic#Area_under_the_curve

Register the model in MLflow Model Registry

By registering this model in Model Registry, you can easily reference the model from anywhere within Databricks.

The following section shows how to do this programmatically, but you can also register a model using the UI. See "Create or register a model using the UI" (AWS (https://docs.databricks.com/applications/machine-learning/manage-model-

lifecycle/index.html#create-or-register-a-model-using-the-ui)|Azure (https://docs.microsoft.com/azure/databricks/applications/machine-learning/manage-model-lifecycle/index#create-or-register-a-model-using-the-ui)|GCP (https://docs.gcp.databricks.com/applications/machine-learning/manage-model-lifecycle/index.html#create-or-register-a-model-using-the-ui)).

```
run_id = mlflow.search_runs(filter_string='tags.mlflow.runName =
"untuned_random_forest"').iloc[0].run_id
```

```
# If you see the error "PERMISSION_DENIED: User does not have any permission
level assigned to the registered model",
# the cause may be that a model already exists with the name "wine_quality".
Try using a different name.
model_name = "wine_quality"
model_version = mlflow.register_model(f"runs:/{run_id}/random_forest_model",
model_name)

# Registering the model takes a few seconds, so add a small delay
time.sleep(15)
```

```
Registered model 'wine_quality' already exists. Creating a new version of this
model...
2022/03/08 04:20:15 INFO mlflow.tracking._model_registry.client: Waiting up to
300 seconds for model version to finish creation.                    Model na
me: wine_quality, version 30
Created version '30' of model 'wine_quality'.
```

You should now see the model in the Models page. To display the Models page, click the Models icon in the left sidebar.

Next, transition this model to production and load it into this notebook from Model Registry.

```
from mlflow.tracking import MlflowClient

client = MlflowClient()
client.transition_model_version_stage(
  name=model_name,
  version=model_version.version,
  stage="Production",
)
```

```
Out[43]: <ModelVersion: creation_timestamp=1646713215808, current_stage='Produ
ction', description='', last_updated_timestamp=1646713237353, name='wine_quali
ty', run_id='7fd422b0d80f4b91b74bbaaded2fb79c', run_link='', source='dbfs:/dat
abricks/mlflow-tracking/2706240218630387/7fd422b0d80f4b91b74bbaaded2fb79c/arti
facts/random_forest_model', status='READY', status_message='', tags={}, user_i
d='5813470939533708', version='30'>
```

The Models page now shows the model version in stage "Production".

You can now refer to the model using the path "models:/wine_quality/production".

```
model = mlflow.pyfunc.load_model(f"models:/{model_name}/production")

# Sanity-check: This should match the AUC logged by MLflow
print(f'AUC: {roc_auc_score(y_test, model.predict(X_test))}')
```

```
AUC: 0.8540300975814177
```

Experiment with a new model

The random forest model performed well even without hyperparameter tuning.

The following code uses the xgboost library to train a more accurate model. It runs a parallel hyperparameter sweep to train multiple models in parallel, using Hyperopt and SparkTrials. As before, the code tracks the performance of each parameter configuration with MLflow.

```
from hyperopt import fmin, tpe, hp, SparkTrials, Trials, STATUS_OK
from hyperopt.pyll import scope
from math import exp
import mlflow.xgboost
import numpy as np
import xgboost as xgb

search_space = {
  'max_depth': scope.int(hp.quniform('max_depth', 4, 100, 1)),
  'learning_rate': hp.loguniform('learning_rate', -3, 0),
  'reg_alpha': hp.loguniform('reg_alpha', -5, -1),
  'reg_lambda': hp.loguniform('reg_lambda', -6, -1),
  'min_child_weight': hp.loguniform('min_child_weight', -1, 3),
  'objective': 'binary:logistic',
  'seed': 123, # Set a seed for deterministic training
}

def train_model(params):
  # With MLflow autologging, hyperparameters and the trained model are
automatically logged to MLflow.
  mlflow.xgboost.autolog()
  with mlflow.start_run(nested=True):
    train = xgb.DMatrix(data=X_train, label=y_train)
    validation = xgb.DMatrix(data=X_val, label=y_val)
    # Pass in the validation set so xgb can track an evaluation metric. XGBoost
terminates training when the evaluation metric
    # is no longer improving.
    booster = xgb.train(params=params, dtrain=train, num_boost_round=1000,\
                        evals=[(validation, "validation")],
early_stopping_rounds=50)
    validation_predictions = booster.predict(validation)
    auc_score = roc_auc_score(y_val, validation_predictions)
    mlflow.log_metric('auc', auc_score)

    signature = infer_signature(X_train, booster.predict(train))
    mlflow.xgboost.log_model(booster, "model", signature=signature)

    # Set the loss to -1*auc_score so fmin maximizes the auc_score
    return {'status': STATUS_OK, 'loss': -1*auc_score, 'booster':
booster.attributes()}

# Greater parallelism will lead to speedups, but a less optimal hyperparameter
sweep.
# A reasonable value for parallelism is the square root of max_evals.
spark_trials = SparkTrials(parallelism=10)
```

```
# Run fmin within an MLflow run context so that each hyperparameter
configuration is logged as a child run of a parent
# run called "xgboost_models" .
with mlflow.start_run(run_name='xgboost_models'):
  best_params = fmin(
    fn=train_model,
    space=search_space,
    algo=tpe.suggest,
    max_evals=96,
    trials=spark_trials,
  )
```

```
Hyperopt with SparkTrials will automatically track trials in MLflow. To view t
he MLflow experiment associated with the notebook, click the 'Runs' icon in th
e notebook context bar on the upper right. There, you can view all runs.
To view logs from trials, please check the Spark executor logs. To view execut
or logs, expand 'Spark Jobs' above until you see the (i) icon next to the stag
e from the trial job. Click it and find the list of tasks. Click the 'stderr'
link for a task to view trial logs.

  0%|          | 0/96 [00:00<?, ?trial/s, best loss=?]
  2%|▏         | 2/96 [00:10<08:03,  5.14s/trial, best loss: -0.88233709456443
1]
  3%|▎         | 3/96 [00:14<07:14,  4.68s/trial, best loss: -0.88233709456443
1]
  4%|▍         | 4/96 [00:17<06:19,  4.13s/trial, best loss: -0.88465002078169
97]
  6%|▋         | 6/96 [00:21<04:33,  3.04s/trial, best loss: -0.88465002078169
97]
  7%|▋         | 7/96 [00:23<04:05,  2.76s/trial, best loss: -0.88465002078169
97]
  9%|▉         | 9/96 [00:30<04:31,  3.13s/trial, best loss: -0.89339180431335
73]
```

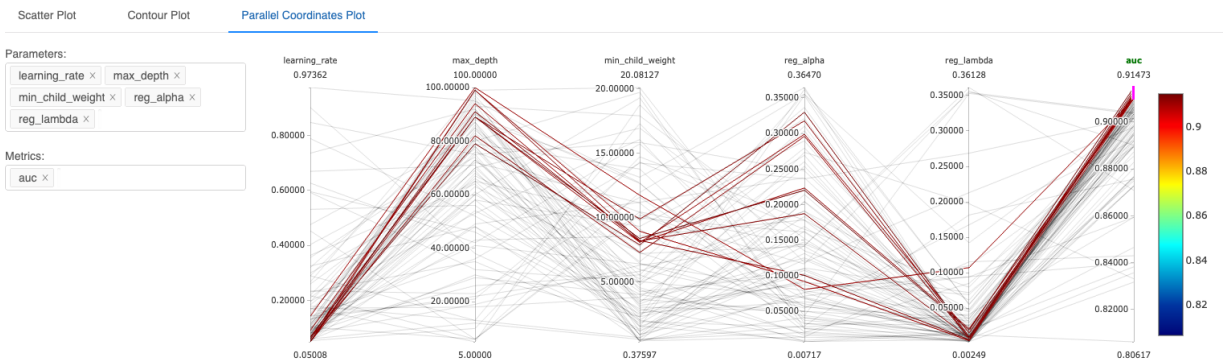
Use MLflow to view the results

Open up the Experiment Runs sidebar to see the MLflow runs. Click on Date next to the down arrow to display a menu, and select 'auc' to display the runs sorted by the auc metric. The highest auc value is 0.90.

MLflow tracks the parameters and performance metrics of each run. Click the External Link icon at the top of the Experiment Runs sidebar to navigate to the MLflow Runs Table.

Now investigate how the hyperparameter choice correlates with AUC. Click the "+" icon to expand the parent run, then select all runs except the parent, and click "Compare". Select the Parallel Coordinates Plot.

The Parallel Coordinates Plot is useful in understanding the impact of parameters on a metric. You can drag the pink slider bar at the upper right corner of the plot to highlight a subset of AUC values and the corresponding parameter values. The plot below highlights the highest AUC values:



Notice that all of the top performing runs have a low value for reg_lambda and learning_rate.

You could run another hyperparameter sweep to explore even lower values for these parameters. For simplicity, that step is not included in this example.

You used MLflow to log the model produced by each hyperparameter configuration. The following code finds the best performing run and saves the model to Model Registry.

```
best_run = mlflow.search_runs(order_by=['metrics.auc DESC']).iloc[0]
print(f'AUC of Best Run: {best_run["metrics.auc"]}')
```

```
AUC of Best Run: 0.9003632949000169
```

Update the production `wine_quality` model in MLflow Model Registry

Earlier, you saved the baseline model to Model Registry with the name `wine_quality`. Now that you have a created a more accurate model, update `wine_quality`.

```
new_model_version = mlflow.register_model(f"runs:/{best_run.run_id}/model",
model_name)

# Registering the model takes a few seconds, so add a small delay
time.sleep(15)
```

```
Registered model 'wine_quality' already exists. Creating a new version of this
model...
2022/03/08 04:24:04 INFO mlflow.tracking._model_registry.client: Waiting up to
300 seconds for model version to finish creation.                    Model na
me: wine_quality, version 31
Created version '31' of model 'wine_quality'.
```

Click **Models** in the left sidebar to see that the `wine_quality` model now has two versions.

The following code promotes the new version to production.

```
# Archive the old model version
client.transition_model_version_stage(
  name=model_name,
  version=model_version.version,
  stage="Archived"
)

# Promote the new model version to Production
client.transition_model_version_stage(
  name=model_name,
  version=new_model_version.version,
  stage="Production"
)
```

```
Out[48]: <ModelVersion: creation_timestamp=1646713443971, current_stage='Produ
ction', description='', last_updated_timestamp=1646713465720, name='wine_quali
ty', run_id='ef703dd69da445d69ecf8f56be34354c', run_link='', source='dbfs:/dat
abricks/mlflow-tracking/2706240218630387/ef703dd69da445d69ecf8f56be34354c/arti
facts/model', status='READY', status_message='', tags={}, user_id='58134709395
33708', version='31'>
```

Clients that call load_model now receive the new model.

```
# This code is the same as the last block of "Building a Baseline Model". No
change is required for clients to get the new model!
model = mlflow.pyfunc.load_model(f"models:/{model_name}/production")
print(f'AUC: {roc_auc_score(y_test, model.predict(X_test))}')
```

```
AUC: 0.9058945462164752
```

The auc value on the test set for the new model is 0.90. You beat the baseline!

Batch inference

There are many scenarios where you might want to evaluate a model on a corpus of new data. For example, you may have a fresh batch of data, or may need to compare the performance of two models on the same corpus of data.

The following code evaluates the model on data stored in a Delta table, using Spark to run the computation in parallel.

```
# To simulate a new corpus of data, save the existing X_train data to a Delta
table.
# In the real world, this would be a new batch of data.
spark_df = spark.createDataFrame(X_train)
# Replace <username> with your username before running this cell.
table_path = "dbfs:/<username>/delta/wine_data"
# Delete the contents of this path in case this cell has already been run
dbutils.fs.rm(table_path, True)
spark_df.write.format("delta").save(table_path)
```

Load the model into a Spark UDF, so it can be applied to the Delta table.

```
import mlflow.pyfunc

apply_model_udf = mlflow.pyfunc.spark_udf(spark,
f"models:/{model_name}/production")
```

```
# Read the "new data" from Delta
new_data = spark.read.format("delta").load(table_path)


display(new_data)
```

	fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides
1	7.2	0.23	0.39	2.3	0.033
2	6.4	0.24	0.27	1.5	0.04
3	7.4	0.24	0.29	10.1	0.05
4	8.6	0.37	0.65	6.4	0.08
5	5.9	0.32	0.39	3.3	0.114
6	7	0.42	0.35	1.6	0.088
7	6.7	0.31	0.44	6.7	0.054

Truncated results, showing first 1000 rows.

```
from pyspark.sql.functions import struct

# Apply the model to the new data
udf_inputs = struct(*(X_train.columns.tolist()))

new_data = new_data.withColumn(
  "prediction",
  apply_model_udf(udf_inputs)
)


# Each row now has an associated prediction. Note that the xgboost function
does not output probabilities by default, so the predictions are not limited to
the range [0, 1].
display(new_data)
```

	fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides
1	7.2	0.23	0.39	2.3	0.033
2	6.4	0.24	0.27	1.5	0.04
3	7.4	0.24	0.29	10.1	0.05
4	8.6	0.37	0.65	6.4	0.08
5	5.9	0.32	0.39	3.3	0.114
6	7	0.42	0.35	1.6	0.088
7	6.7	0.31	0.44	6.7	0.054

Truncated results, showing first 1000 rows.

Model serving

To productionize the model for low latency predictions, use MLflow Model Serving (AWS (https://docs.databricks.com/applications/mlflow/model-serving.html)|Azure (https://docs.microsoft.com/azure/databricks/applications/mlflow/model-serving)|GCP (https://docs.gcp.databricks.com/applications/mlflow/model-serving.html)) to deploy the model to an endpoint.

The following code illustrates how to issue requests using a REST API to get predictions from the deployed model.

You need a Databricks token to issue requests to your model endpoint. You can generate a token from the User Settings page (click Settings in the left sidebar). Copy the token into the next cell.

```
import os
os.environ["DATABRICKS_TOKEN"] = "<YOUR_TOKEN>"
```

Click **Models** in the left sidebar and navigate to the registered wine model. Click the serving tab, and then click **Enable Serving**.

Then, under **Call The Model**, click the **Python** button to display a Python code snippet to issue requests. Copy the code into this notebook. It should look similar to the code in the next cell.

You can use the token to make these requests from outside Databricks notebooks as well.

```
# Replace with code snippet from the model serving page
import os
import requests
import pandas as pd

def score_model(dataset: pd.DataFrame):
  url = 'https://<DATABRICKS_URL>/model/wine_quality/Production/invocations'
  headers = {'Authorization': f'Bearer {os.environ.get("DATABRICKS_TOKEN")}'}
  data_json = dataset.to_dict(orient='split')
  response = requests.request(method='POST', headers=headers, url=url,
json=data_json)
  if response.status_code != 200:
    raise Exception(f'Request failed with status {response.status_code},
{response.text}')
  return response.json()
```

The model predictions from the endpoint should agree with the results from locally evaluating the model.

```
  Command skipped
```