Wine Quality Prediction Lab Documentation

This documentation provides a step-by-step guide to a data science lab focused on predicting wine quality using Python. The lab covers data preprocessing, model training, model registration, and batch inference using MLflow and various machine learning libraries.

Prerequisites

Before starting the lab, ensure that you have the following:

- Python environment set up with required libraries installed.
- Datasets: You will need two CSV files, `winequality-white.csv` and `winequality-red.csv`, containing white and red wine data, respectively.

Step 1: Importing Data

In this step, we load the white and red wine datasets using the Pandas library.

```
import pandas as pd

# Load white wine data
white_wine = pd.read_csv("data/winequality-white.csv", sep=";")

# Load red wine data
red_wine = pd.read_csv("data/winequality-red.csv", sep=",")
```

Step 2: Exploring Data

In this step, we'll explore the wine dataset by examining the first few rows of both the white and red wine datasets.

Code:

```
white_wine.head()
red_wine.head()
```

Step 3: Data Preprocessing

In this step, we'll perform data preprocessing tasks to prepare the wine dataset for model training.

Adding Indicator Variable

We add an indicator variable 'is_red' to both datasets to distinguish between white and red wines.

Code:

```
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)
```

Step 4: Data Visualization

In this step, we'll visualize the distribution of the 'quality' variable in the wine dataset.

Code:

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.distplot(data.quality, kde=False);
```

Step 5: Define High-Quality Wines

In this step, we'll define what constitutes a "high-quality" wine based on a quality score threshold.

Code:

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

Explanation:

We create a new binary column high_quality in the data DataFrame.
A wine is considered "high quality" if its 'quality' score is greater than or equal to 7.
We use (data.quality >= 7) to create a boolean mask and then convert it to integers (astype(int)) to represent high-quality wines as 1 and others as 0.
The original 'quality' column is updated with these new values.
Expected Output:
The data DataFrame will have a new 'quality' column indicating whether each wine is of high quality (1) or not (0).

Step 6: Exploratory Data Analysis (EDA)

In this step, we'll perform Exploratory Data Analysis (EDA) by creating box plots to identify potential predictors of wine quality.

Code:

```
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
```

Step 7: Checking for Missing Data

In this step, we'll check for missing data within the wine dataset.

Code:

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

Step 8: Data Splitting

In this step, we'll split the dataset into training, validation, and test sets to prepare for model training and evaluation.

Code:

```
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)
```

Step 9: Building a Baseline Model

In this step, we'll create a baseline model using a random forest classifier and log its performance using MLflow.

Code:

```
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)
```

Explanation:

We create a random forest classifier model using scikit-learn's RandomForestClassifier.
The model is trained on the training data (X_train, y_train).
We log various information using MLflow, including model parameters (n_estimators), the Area Under the ROC Curve (AUC) metric, and the model itself.
A wrapper class SklearnModelWrapper is used to make predictions using predict_proba, which returns class probabilities.
We also define a signature to validate inputs when the model is deployed.
Expected Output:
Model training details and metrics (e.g., AUC) will be logged in the MLflow run.
The trained model will be saved for future use.

Step 10: Feature Importance Analysis

In this step, we analyze feature importance to identify which features have the most impact on predicting wine quality.

Code:

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

Explanation:

We calculate the feature importances using the trained random forest classifier model.
The model.feature_importances_ attribute provides the importance scores for each feature.
We create a DataFrame feature_importances to display the importances along with feature names.
Finally, we sort the DataFrame in descending order to identify the most important features.
Expected Output:
The output will be a table showing the feature importances in descending order, with the most important features at the top.

Step 11: Model Registration in MLflow Model Registry

In this step, we'll register the trained model in the MLflow Model Registry for version tracking and management.

Code:

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

Explanation:

We retrieve the run ID of the MLflow run where the model was trained using mlflow.search_runs.
We specify the desired model name, in this case, "wine_quality."
We use mlflow.register_model to register the model in the Model Registry. The path to the model is constructed using the run ID.
A delay is added to ensure the model registration process is completed.
Expected Output:
The trained model will be registered in the MLflow Model Registry under the specified model name ("wine_quality").

Note:

Model registration in the MLflow Model Registry allows for versioning and tracking of different model versions. It's a crucial step for managing and deploying machine learning models in a production environment.

Step 12: Transitioning Model Version to Production

In this step, we'll transition the newly registered model version to the "Production" stage in the MLflow Model Registry.

Code:

```
from mlflow.tracking import MlflowClient

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

Explanation:

We use the MlflowClient to interact with the MLflow Tracking Server programmatically.
The client.transition_model_version_stage method is used to transition the model version to the "Production" stage.
Expected Output:
The model version will be moved to the "Production" stage in the MLflow Model Registry.

Note:

Transitioning a model version to "Production" indicates that it is ready for use in a production environment. You can now refer to the model using the path "models:/wine_quality/production."
This step is crucial for managing the deployment of machine learning models.

Step 13: Model Inference and Evaluation

In this step, we'll load the production version of the model from the MLflow Model Registry and perform batch inference and evaluation.

Code:

```
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))}')
```

Explanation:

We load the production version of the model from the MLflow Model Registry using mlflow.pyfunc.load_model.
We perform batch inference on the test data (X_test) using the loaded model.
We calculate and print the Area Under the ROC Curve (AUC) score to assess the model's performance.
Expected Output:
The AUC score will be printed, providing an evaluation of the model's performance on the test data.

Note:

Loading the production model version allows us to make predictions on new data.
The AUC score is used here as an example metric for model evaluation. Depending on the problem, other evaluation metrics may be more appropriate.

Step 14: Batch Inference with the Deployed Model

In this step, we'll perform batch inference using the deployed model to make predictions on a dataset stored in a Delta table.

Code:

```
import mlflow.pyfunc


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

```
new_data = spark.read.format("csv").load(table_path)


# Apply the model to the new data
predictions = new_data.withColumn("prediction", apply_model_udf(*new_data.columns))
```

Explanation:

We use the mlflow.pyfunc.spark_udf function to create a Spark User-Defined Function (UDF) that applies the deployed production model to the input data.
We read the new data from a Delta table using spark.read.format("csv").load(table_path).
We apply the model to the new data using the UDF and create a new DataFrame predictions that includes the predictions.
Expected Output:
The predictions DataFrame will contain the input data along with a new column "prediction" that contains the model's predictions.

Note:

Batch inference is the process of using a trained machine learning model to make predictions on a batch of data. In this step, we apply the deployed model to new data stored in a Delta table to make predictions at scale.

Step 15: Batch Inference Result Verification

In this step, we'll verify and inspect the results of the batch inference performed using the deployed model.

Code:

```
predictions.show()
```

Step 16: Serving the Model for Real-Time Inference

In this step, we'll serve the model to enable real-time inference using MLflow's model serving capabilities.

Code (Command Line):

```
# Serve the model using the MLflow Model Serving
mlflow models serve -m models:/${model_name}/production -h 0.0.0.0 -p 5001
```

Explanation:

We use the mlflow models serve command to start serving the model.

```
-m models:/${model_name}/production specifies the model to be served. Adjust the path accordingly
to match your model name and stage.
-h 0.0.0.0 specifies the host on which the model will be served.
-p 5001 specifies the port on which the model will be available for real-time inference.
```

Expected Output:

The model will be served and ready to accept real-time inference requests.

Note:

Serving a model allows you to make real-time predictions by sending requests to the model's API endpoint.
Ensure that the MLflow Model Server is correctly configured and running before serving the model.
Real-time serving is useful for integrating machine learning models into production systems, applications, or APIs.

Step 17: Performing Real-Time Inference with the Deployed Model

In this step, we'll perform real-time inference by sending requests to the deployed model's API endpoint.

Code (Python):

```
import requests
import json

url = 'http://localhost:5001/invocations'  # Replace with the actual endpoint URL

data_dict = {"dataframe_split": X_test.to_dict(orient='split')}

response = requests.post(url, json=data_dict)
predictions = response.json()

print(predictions)
```

Explanation:

We use the requests library to send a POST request to the deployed model's API endpoint.
The URL should be set to the correct endpoint where the model is served.
We prepare the input data in the desired format and send it as JSON in the request.
The response contains the model's predictions, which we extract using response.json().
Finally, we print the predictions.
Expected Output:
The output will be the model's predictions for the input data sent in the request.

Note:

Real-time inference allows you to use the deployed model to make predictions on new data as it becomes available.
Ensure that the model serving endpoint is running and accessible before making real-time inference requests.
Replace the endpoint URL with the actual URL where your model is served.

Step 18: Cleaning Up and Conclusion

In this final step, we'll wrap up the lab and perform any necessary clean-up tasks.

Clean-Up Tasks:

- **Stop the Model Serving**: If you've started the model serving process, make sure to stop it when you're done with real-time inference. You can do this by stopping the MLflow Model Serving process or using appropriate commands.
- **Close Resources**: Ensure that any resources or connections used during the lab are properly closed or released.
- **Save Documentation**: Save this lab documentation for future reference or sharing with others.

Conclusion:

In this lab, we've covered various aspects of the machine learning lifecycle, including data preparation, model training, evaluation, deployment, and real-time inference. Here are the key takeaways:

- Data preparation is essential for training and evaluating machine learning models. Cleaning, transforming, and splitting the data are crucial steps.
- Model training involves selecting an appropriate algorithm, training the model, and evaluating its performance using relevant metrics.
- Model deployment involves registering the model, transitioning it to the production stage, and serving it for real-time inference.
- Real-time inference allows you to use the deployed model to make predictions on new data as it arrives.

By following these steps, you can effectively develop and deploy machine learning models for various applications.

Thank you for completing this lab!