

# Wine Quality Classification using Machine Learning

## Overview

The purpose of this project is to build and evaluate machine learning models that can accurately predict the quality of red wine. By analyzing various physicochemical properties, we aim to understand which features are most indicative of wine quality and to develop a robust classifier. This can be particularly useful for vintners seeking to assess and improve wine quality or for retailers recommending wines based on customer preferences.

## Select a dataset.

- `dbutils.fs.ls('/databricks-datasets/')`

```
In [ ]: dbutils.fs.ls('/databricks-datasets/')
```

```
Out[ ]: [FileInfo(path='dbfs:/databricks-datasets/COVID/', name='COVID/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/README.md', name='README.md', size=976, modificationTime=1532502324000),
  FileInfo(path='dbfs:/databricks-datasets/Rdatasets/', name='Rdatasets/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/SPARK_README.md', name='SPARK_README.md', size=3359, modificationTime=1455505834000),
  FileInfo(path='dbfs:/databricks-datasets/adult/', name='adult/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/airlines/', name='airlines/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/amazon/', name='amazon/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/asa/', name='asa/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/atlas_higgs/', name='atlas_higgs/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/bikeSharing/', name='bikeSharing/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/cctvVideos/', name='cctvVideos/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/credit-card-fraud/', name='credit-card-fraud/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/cs100/', name='cs100/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/cs110x/', name='cs110x/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/cs190/', name='cs190/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/data.gov/', name='data.gov/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/definitive-guide/', name='definitive-guide/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/delta-sharing/', name='delta-sharing/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/flights/', name='flights/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/flower_photos/', name='flower_photos/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/flowers/', name='flowers/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/genomics/', name='genomics/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/hail/', name='hail/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/identifying-campaign-effectiveness/', name='identifying-campaign-effectiveness/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/inventory-reports/', name='inventory-reports/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/iot/', name='iot/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/iot-stream/', name='iot-stream/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/learning-spark/', name='learning-spark/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/learning-spark-v2/', name='learning-spark-v2/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/lending-club-loan-stats/', name='lending-club-loan-stats/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/med-images/', name='med-images/', size=0, modificationTime=1721442662906)]
```

```

modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/media/', name='media/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/mnist-digits/', name='mnist-digits/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/news20.binary/', name='news20.binary/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/nyctaxi/', name='nyctaxi/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/nyctaxi-with-zipcodes/', name='nyctaxi-with-zipcodes/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/online_retail/', name='online_retail/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/overlap-join/', name='overlap-join/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/power-plant/', name='power-plant/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/retail-org/', name='retail-org/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/rwe/', name='rwe/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/sai-summit-2019-sf/', name='sai-summit-2019-sf/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/sample_logs/', name='sample_logs/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/samples/', name='samples/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/sfo_customer_survey/', name='sfo_customer_survey/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/sms_spam_collection/', name='sms_spam_collection/', size=0, modificationTime=1721442662906),
  FileInfo(path='dbfs:/databricks-datasets/songs/', name='songs/', size=0, modificationTime=1721442662907),
  FileInfo(path='dbfs:/databricks-datasets/structured-streaming/', name='structured-streaming/', size=0, modificationTime=1721442662907),
  FileInfo(path='dbfs:/databricks-datasets/timeseries/', name='timeseries/', size=0, modificationTime=1721442662907),
  FileInfo(path='dbfs:/databricks-datasets/tpch/', name='tpch/', size=0, modificationTime=1721442662907),
  FileInfo(path='dbfs:/databricks-datasets/travel_recommendations_realtime/', name='travel_recommendations_realtime/', size=0, modificationTime=1721442662907),
  FileInfo(path='dbfs:/databricks-datasets/warmup/', name='warmup/', size=0, modificationTime=1721442662907),
  FileInfo(path='dbfs:/databricks-datasets/weather/', name='weather/', size=0, modificationTime=1721442662907),
  FileInfo(path='dbfs:/databricks-datasets/wiki/', name='wiki/', size=0, modificationTime=1721442662907),
  FileInfo(path='dbfs:/databricks-datasets/wikipedia-datasets/', name='wikipedia-datasets/', size=0, modificationTime=1721442662907),
  FileInfo(path='dbfs:/databricks-datasets/wine-quality/', name='wine-quality/', size=0, modificationTime=1721442662907)]

```

```

In [ ]: # List the files in the 'sfo_customer_survey' directory
sfo_customer_survey_files = dbutils.fs.ls('dbfs:/databricks-datasets/wine-quality/')
print("Files in sfo_customer_survey directory:")
for file in sfo_customer_survey_files:
    print(file.path)

```

Files in sfo\_customer\_survey directory:

```

dbfs:/databricks-datasets/wine-quality/README.md
dbfs:/databricks-datasets/wine-quality/winequality-red.csv
dbfs:/databricks-datasets/wine-quality/winequality-white.csv

```

```
In [ ]: # Path to the README file
readme_file_path = '/databricks-datasets/wine-quality/README.md'

# Read the README file
readme_df = spark.read.text(readme_file_path)

# Show the contents of the README file
readme_df.show(truncate=False)
```

```
+-----+
+-----+
|value|
+-----+
+-----+
|Wine Quality Data Set|
|=====|
|Two datasets related to red and white variants of the Portuguese "Vinho Verde" wine.|
|
|Provenance|
|=====|
|This data set was obtained from http://archive.ics.uci.edu/ml/datasets/wine+quality.|
|The source of the data is:|
|Paulo Cortez, University of Minho, Guimarães, Portugal, http://www3.dsi.uminho.pt/pcorte|
|A. Cerdeira, F. Almeida, T. Matos and J. Reis, Viticulture Commission of the Vinho Verde|
|Region(CVRVV), Porto, Portugal. @2009|
|
|License and/or Citation|
|=====|
|Example:|
|This data set is licensed under the following license: See citations.|
|
|Applicable citations:|
|Cortez, Paulo (2009). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. I|
|rvine, CA: University of California, School of Information and Computer Science.|
|
|P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis.|
+-----+
+-----+
only showing top 20 rows
```

Question of interest.

My question of interest is whether we can predict the classify a wine based on its physicochemical properties. We are using the Red Wine dataset from the Vinho Verde in northwestern Portugal. This will be a classification problem in which we try to predict a low, medium or high quality.

## Perform EDA on your dataset.

```
In [ ]: # Set your user name in the widgit in the upper left of the screen.  
# This is required so that you can create a folder for yourself!
```

```
# Your User Name Here  
username = dbutils.widgets.get("username")  
save_path = f"dbfs:/tmp/w8/{username}"  
  
silver_path = f"{save_path}/silver"  
  
# View the paths  
print(silver_path)
```

```
dbfs:/tmp/w8/cthirteen/silver
```

```
In [ ]: from pyspark.sql.functions import udf  
from pyspark.sql.types import StringType  
  
# Read in the raw data from the CSV Source  
red_wine = spark.read.csv(  
    "/databricks-datasets/wine-quality/winequality-red.csv",  
    schema="`fixed_acidity` DOUBLE, `volatile_acidity` DOUBLE, `citric_acid` DOUBLE, `re  
)  
  
# Define the categorization functions  
def categorize_quality(score):  
    if score <= 4:  
        return 'Poor'  
    elif score <= 5:  
        return 'Fair'  
    elif score <= 6:  
        return 'Commended'  
    elif score <= 7:  
        return 'Bronze'  
    elif score <= 8:  
        return 'Silver Medal'  
    else:  
        return 'Gold'  
  
categorize_quality_udf = udf(categorize_quality, StringType())  
red_wine = red_wine.withColumn('quality_category', categorize_quality_udf(red_wine['qual  
  
    # Display the updated DataFrame  
#red_wine.display()  
  
# Create our Delta Table in our silversilver_path staging area  
red_wine.write.format('delta').mode('overwrite').option("mergeSchema", "true").save(f"{s
```

fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	total_sulfu
7.4	0.7	0.0	1.9	0.076	11.0	
7.8	0.88	0.0	2.6	0.098	25.0	
7.8	0.76	0.04	2.3	0.092	15.0	
11.2	0.28	0.56	1.9	0.075	17.0	
7.4	0.7	0.0	1.9	0.076	11.0	
7.4	0.66	0.0	1.8	0.075	13.0	
7.9	0.6	0.06	1.6	0.069	15.0	
7.3	0.65	0.0	1.2	0.065	15.0	
7.8	0.58	0.02	2.0	0.073	9.0	

```
In [ ]: # Read the silver red wine data into a SparkDataFrame
red_wine_delta = spark.read.format("delta").load(f"{silver_path}/wine_quality")

display(red_wine_delta)
```

fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	total_sulfu
7.4	0.7	0.0	1.9	0.076	11.0	
7.8	0.88	0.0	2.6	0.098	25.0	
7.8	0.76	0.04	2.3	0.092	15.0	
11.2	0.28	0.56	1.9	0.075	17.0	
7.4	0.7	0.0	1.9	0.076	11.0	
7.4	0.66	0.0	1.8	0.075	13.0	
7.9	0.6	0.06	1.6	0.069	15.0	
7.3	0.65	0.0	1.2	0.065	15.0	
7.8	0.58	0.02	2.0	0.073	9.0	

```
In [ ]: # Check schema and column names
red_wine_delta.printSchema()

root
|-- fixed_acidity: double (nullable = true)
|-- volatile_acidity: double (nullable = true)
|-- citric_acid: double (nullable = true)
|-- residual_sugar: double (nullable = true)
|-- chlorides: double (nullable = true)
|-- free_sulfur_dioxide: double (nullable = true)
|-- total_sulfur_dioxide: double (nullable = true)
|-- density: double (nullable = true)
|-- pH: double (nullable = true)
|-- sulphates: double (nullable = true)
|-- alcohol: double (nullable = true)
|-- quality: double (nullable = true)
|-- quality_category: string (nullable = true)
```

## Explanation of the variables

**fixed acidity:** The amount of non-volatile acids in the wine. These acids do not evaporate easily and contribute to the overall acidity of the wine. Higher levels of fixed acidity can contribute to a wine's crispness and tartness. However, excessively high levels can make the wine taste too sharp or sour. The optimal level depends on the wine style and balance with other components.

**volatile acidity:** The amount of acetic acid in wine, which can lead to an unpleasant vinegar-like taste if too high. Lower volatile acidity is typically preferred for higher-quality wines.

**citric acid:** Provides a fresh flavor to wines and is usually found in small quantities. Wines with higher citric acid might have a more refreshing taste.

**residual\_sugar:** Refers to the amount of sugar remaining in the wine after fermentation, measured in grams per liter. The perception of sweetness in wine is influenced by residual sugar levels. For dry wines, low residual sugar is preferred to maintain balance and allow other flavors to shine. In sweeter wines, higher residual sugar can contribute to a perceived fullness and roundness.

**chlorides:** Represents the amount of salt in the wine. Chloride levels are typically low in wine but can influence its taste and mouthfeel. Higher chlorides might contribute to a salty or briny taste, undesirable in excess.

**free sulfur dioxide:** Measures the free form of sulfur dioxide (SO<sub>2</sub>) in the wine, which acts as an antioxidant and antimicrobial agent. Adequate free sulfur dioxide levels help preserve wine freshness and prevent spoilage.

**total sulfur dioxide:** Indicates the total amount of sulfur dioxide (free + bound forms) in the wine. Too high total sulfur dioxide levels can lead to a pungent aroma and affect taste negatively.

**density:** Represents the density of the wine, which is close to that of water depending on the alcohol and sugar content. Density affects mouthfeel and body. Higher density wines may feel fuller-bodied, while lower density wines can feel lighter. It contributes to the overall texture and perceived quality of the wine.

**pH:** Measures the acidity or basicity of the wine on a scale from 0 to 14, with lower values indicating higher acidity. Wines with lower pH levels tend to be crisper and more acidic, enhancing freshness. Higher pH levels can lead to a flatter taste and may indicate microbial instability.

**sulphates:** Adds to the wine's antimicrobial and antioxidant properties. Proper levels of sulphates help maintain wine quality and stability.

**alcohol:** Indicates the alcohol content of the wine, typically measured in percent volume. Alcohol contributes to wine body, texture, and perceived warmth. Well-integrated alcohol levels enhance complexity and balance. High alcohol can dominate flavors, while low alcohol may lack depth.

**quality:** Subjective quality rating of the wine. Higher quality wines typically exhibit balanced acidity, complexity, harmony of flavors, and a pleasing mouthfeel.

```
In [ ]: # For visualizations
        # Read the silver red wine data into a SparkDataFrame
```

```
red_wine_delta = spark.read.format("delta").load(f"{silver_path}/wine_quality")
display(red_wine_delta)
```

fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	total_sulfu
7.4	0.7	0.0	1.9	0.076	11.0	
7.8	0.88	0.0	2.6	0.098	25.0	
7.8	0.76	0.04	2.3	0.092	15.0	
11.2	0.28	0.56	1.9	0.075	17.0	
7.4	0.7	0.0	1.9	0.076	11.0	
7.4	0.66	0.0	1.8	0.075	13.0	
7.9	0.6	0.06	1.6	0.069	15.0	
7.3	0.65	0.0	1.2	0.065	15.0	
7.8	0.58	0.02	2.0	0.073	9.0	

Databricks visualization. Run in Databricks to view.  
Databricks visualization. Run in Databricks to view.  
Databricks visualization. Run in Databricks to view.  
Databricks visualization. Run in Databricks to view.  
Databricks visualization. Run in Databricks to view.  
Databricks visualization. Run in Databricks to view.  
Databricks visualization. Run in Databricks to view.  
Databricks visualization. Run in Databricks to view.  
Databricks visualization. Run in Databricks to view.  
Databricks visualization. Run in Databricks to view.  
Databricks visualization. Run in Databricks to view.

Analyzing the graphs, we see the following based on our summary statistics:

- 'fixed\_acidity' is approximately normal with a slight skew towards higher values. Our mean is 8.32, and the standard deviation is 1.74, with moderate variability around the mean.
- 'volatile\_acidity' is right skewed towards lower values. The mean is around 0.53, and the standard deviation is approximately 0.18, suggesting relatively low variability around the mean.
- 'citric\_acid' is right skewed but with three peaks. The mean value is about 0.27, and the standard deviation is around 0.19, indicating variability in citric acid content.
- 'residual\_sugar' is right skewed with a long tail. The mean value is approximately 2.54, and the standard deviation is about 1.41.
- 'chlorides' is right skewed with a long tail. The mean value is around 0.09, and the standard deviation is approximately 0.05.
- 'free\_sulfur\_dioxide' is right skewed with a long tail as well. The mean value is about 15.87, and the standard deviation is around 10.46, indicating variability in free sulfur dioxide levels.
- 'total\_sulfur\_dioxide' is right skewed with a long tail, with potential outliers at higher values. The mean value is approximately 46.47, and the standard deviation is about 32.90, suggesting variability in total sulfur dioxide levels.
- 'density' is approximately normal with a slight skew towards higher values. The mean value is around 0.9967, and the standard deviation is approximately 0.0019, indicating low variability around the mean.
- 'pH' is normally distributed, and a mean value is about 3.31, with a standard deviation of around 0.15.



- 'sulphates' is right skewed with a mean value of approximately 0.66 and a standard deviation of about 0.17.
- 'alcohol' is right skewed, with a mean value of around 10.42 and a standard deviation of approximately 1.07.

```
In [ ]: # Summary statistics
red_wine_delta.describe().display()

# Checking for missing values
from pyspark.sql.functions import col, isnan, when, count
red_wine_delta.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in red_
```

summary	fixed_acidity	volatile_acidity	citric_acid	residual_sugar
count	1599	1599	1599	1599
mean	8.319637273295838	0.5278205128205131	0.2709756097560964	2.5388055034396517
stddev	1.7410963181276948	0.17905970415353525	0.19480113740531824	1.40992805950728
min	4.6	0.12	0.0	0.9
max	15.9	1.58	1.0	15.5

fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	total_sulfu
0	0	0	0	0	0	0

```
In [ ]: # Grouping by 'quality' and aggregating mean values
red_wine_delta.groupBy('quality_category').agg({
    'fixed_acidity': 'mean', 'volatile_acidity': 'mean', 'citric_acid': 'mean', 'residua
    'chlorides': 'mean', 'free_sulfur_dioxide': 'mean', 'total_sulfur_dioxide': 'mean',
    'pH': 'mean', 'sulphates': 'mean', 'alcohol': 'mean'
}).display()
```

quality_category	avg(sulphates)	avg(chlorides)	avg(residual_sugar)	avg(free_sulfu
Fair	0.6209691629955947	0.09273568281938328	2.528854625550658	16.9838472
Poor	0.5922222222222221	0.09573015873015875	2.684920634920635	12.0634920
Silver Medal	0.7677777777777778	0.06844444444444445	2.5777777777777775	13.2777777
Bronze	0.7412562814070353	0.07658793969849244	2.7206030150753793	14.0452261
Commended	0.6753291536050158	0.08495611285266458	2.477194357366772	15.7115987

## Model your data.

To address the question of interest of whether we can classify wine quality based on its physicochemical properties, I explored multiple models and hyperparameters to determine the best approach for classification. Here are the steps taken:

### 1. Data Preparation:

- Utilized the VectorAssembler to transform the feature columns into a single features column.
- Applied StringIndexer to convert string labels into numeric indices for the classification models.

- Split the dataset into training (70%) and testing (30%) subsets.

## 2. Model Experimentation:

- Logistic Regression: Conducted initial experiments with Logistic Regression using 1000 iterations.
- Random Forest Classifier: Trained a Random Forest model with 225 trees.
- Both models were trained and evaluated using PySpark and tracked with MLFlow for experiment management.

## 3. Model Evaluation:

- Employed various evaluation metrics: accuracy, precision, recall, and F1-score.
- Used MulticlassClassificationEvaluator from PySpark for primary metric calculation.
- Converted predictions to Pandas DataFrame to use sklearn's metrics functions for detailed evaluation.

```
In [ ]: import mlflow.sklearn
from pyspark.ml.feature import StringIndexer

# MLFlow can automatically logging your models.
# Support is provided for most of the most popular libraries.
mlflow.sklearn.autolog(log_models=True)

from pyspark.sql.functions import col
from pyspark.ml.feature import VectorAssembler
import matplotlib.pyplot as plt
import seaborn as sns

# Import sklearn
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc
from sklearn.linear_model import LogisticRegression

# Use our 11 measurements
feature_columns = ["fixed_acidity", "volatile_acidity", "citric_acid", "residual_sugar",
]

# removed

#input_data = red_wine_delta[feature_columns]
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
data = assembler.transform(red_wine_delta).select("features", col("quality_category").asLabel())

# StringIndexer to convert the string labels into numeric indices
label_indexer = StringIndexer(inputCol="label", outputCol="indexedLabel")

# Apply StringIndexer to convert 'label' to numeric indices
data = label_indexer.fit(data).transform(data)

# split our dataset into test and training
train_data, test_data = data.randomSplit([0.7, 0.3], seed=1842)

train_data_count = train_data.count()
test_data_count = test_data.count()
print(f"Training data count: {train_data_count}")
print(f"Test data count: {test_data_count}")
```

Training data count: 1141  
Test data count: 458

```
In [ ]: from pyspark.ml.classification import LogisticRegression
        from pyspark.ml.evaluation import MulticlassClassificationEvaluator
        from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc

        # Set the experiment where I want to track my training
        mlflow.set_experiment(experiment_id="2930558423655646")

        # Start an MLflow run; the "with" keyword ensures we'll close the run even if this cell
        with mlflow.start_run() as run:

            # Train a Logistic Regression model using PySpark with 1000 iterations
            lr = LogisticRegression(featuresCol='features', labelCol='indexedLabel', maxIter=100)
            lr_model = lr.fit(train_data)

            # Make predictions on the test data
            predictions = lr_model.transform(test_data)

            # Calculate metrics using PySpark
            evaluator = MulticlassClassificationEvaluator(labelCol="indexedLabel", predictionCol="prediction")
            accuracy = evaluator.evaluate(predictions, {evaluator.metricName: "accuracy"})

            # Log metrics
            mlflow.log_metric("accuracy", accuracy)

            # For confusion matrix and other metrics, you need to convert predictions to Pandas
            # and then use sklearn's metrics functions
            predictions_pd = predictions.select("indexedLabel", "prediction").toPandas()
            y_true = predictions_pd['indexedLabel']
            y_pred = predictions_pd['prediction']

            # Calculate sklearn metrics
            precision = precision_score(y_true, y_pred, average='weighted', zero_division=0)
            recall = recall_score(y_true, y_pred, average='weighted')
            f1 = f1_score(y_true, y_pred, average='weighted')

            # Log sklearn metrics
            mlflow.log_metric("precision", precision)
            mlflow.log_metric("recall", recall)
            mlflow.log_metric("f1_score", f1)

            # End the run
            mlflow.end_run()
```

```
In [ ]: from pyspark.ml.classification import RandomForestClassifier
        from pyspark.ml.evaluation import MulticlassClassificationEvaluator
        from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc
        import mlflow
        import numpy as np

        # Set the experiment where you want to track your training
        mlflow.set_experiment(experiment_id="2930558423655646")

        # Start an MLflow run; the "with" keyword ensures we'll close the run even if this cell
        with mlflow.start_run() as run:
```

```

# Train a Random Forest model using PySpark
rf = RandomForestClassifier(featuresCol='features', labelCol='indexedLabel', numTree
rf_model = rf.fit(train_data)

# Make predictions on the test data
predictions = rf_model.transform(test_data)

# Calculate metrics using PySpark
evaluator = MulticlassClassificationEvaluator(labelCol="indexedLabel", predictionCol
accuracy = evaluator.evaluate(predictions, {evaluator.metricName: "accuracy"})

# Log metrics
mlflow.log_metric("accuracy", accuracy)

# For confusion matrix and other metrics, you need to convert predictions to Pandas
# and then use sklearn's metrics functions
predictions_pd = predictions.select("indexedLabel", "prediction").toPandas()
y_true = predictions_pd['indexedLabel']
y_pred = predictions_pd['prediction']

# Calculate sklearn metrics
precision = precision_score(y_true, y_pred, average='weighted', zero_division=0)
recall = recall_score(y_true, y_pred, average='weighted')
f1 = f1_score(y_true, y_pred, average='weighted')

# Log sklearn metrics
mlflow.log_metric("precision", precision)
mlflow.log_metric("recall", recall)
mlflow.log_metric("f1_score", f1)

# Calculate and log confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)
conf_matrix_file = "confusion_matrix.txt"
np.savetxt(conf_matrix_file, conf_matrix, fmt='%d')
mlflow.log_artifact(conf_matrix_file)

mlflow.end_run()

```

```

In [ ]: id = 2930558423655646
max_results = 1000 # Maximum number of rows to retrieve
runs = mlflow.search_runs(experiment_ids=[id], max_results=max_results)

# Display dataframe in cell output
display(runs)

```

