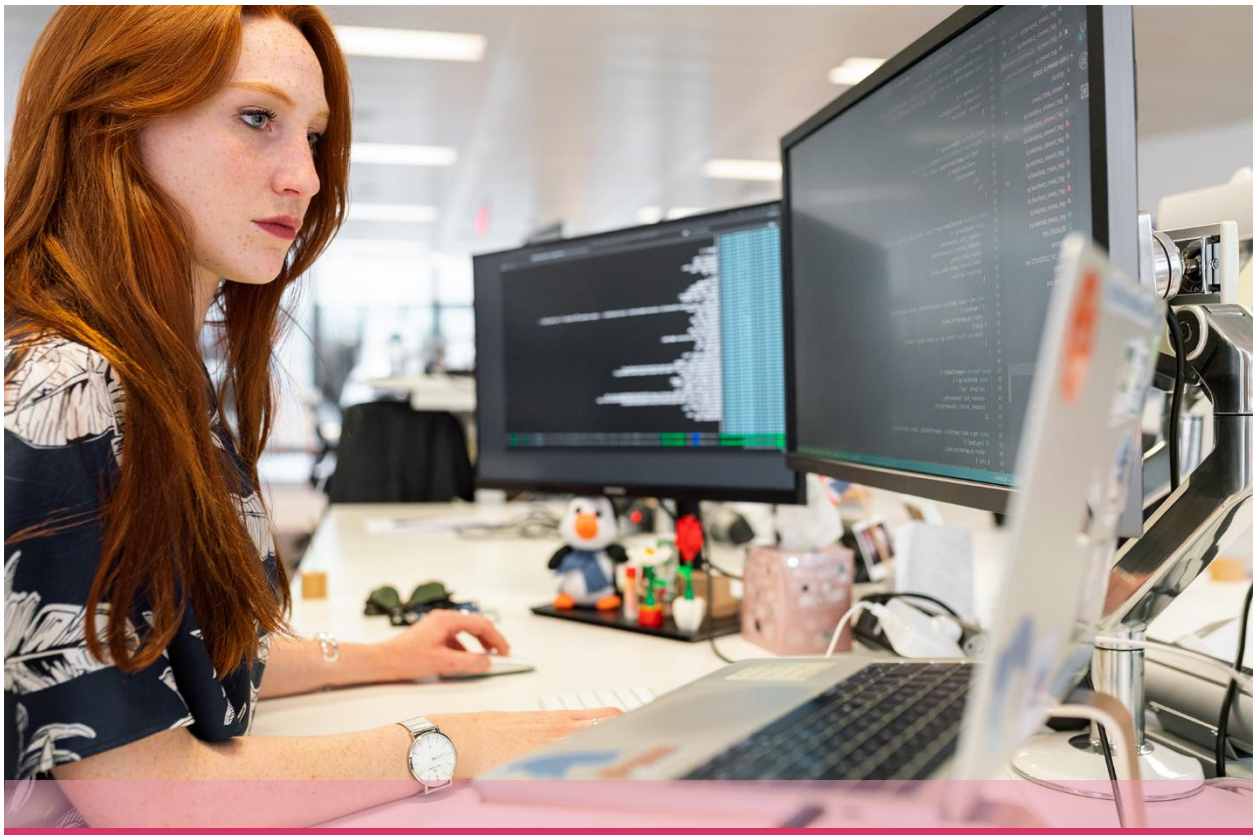


MLOps Fundamentals

Lab Guide

LG-840034-002

skillsoft global knowledge™

Course Information

Copyright

Table of Contents

Lab 1: Initialize an ML Project with GitHub, Virtual Environments, and Pre-Commit Hooks

Lab overview

In this lab, you will:

- Create and set up the GitHub repository
- Create and activate a Python virtual environment
- Configure pre-commit hooks for linting and testing
- Push changes to GitHub with Git tracking

Estimated completion time

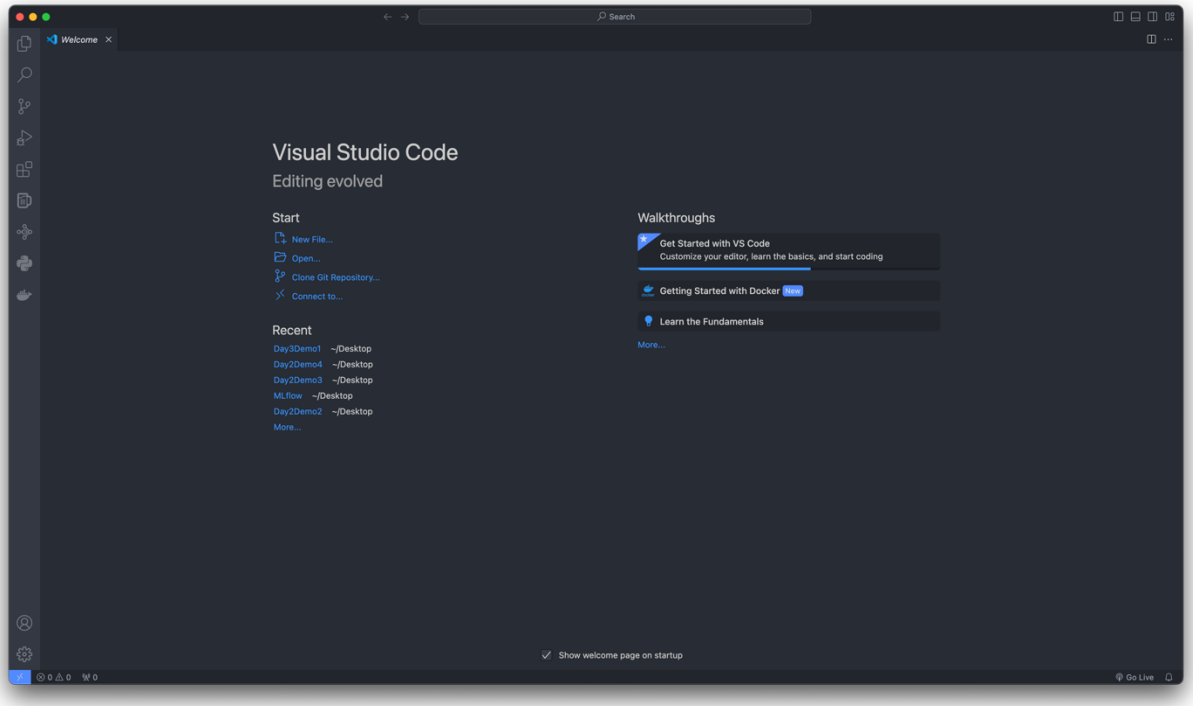
30 minutes

> **Note**
>
> You need a GitHub account to perform this lab.

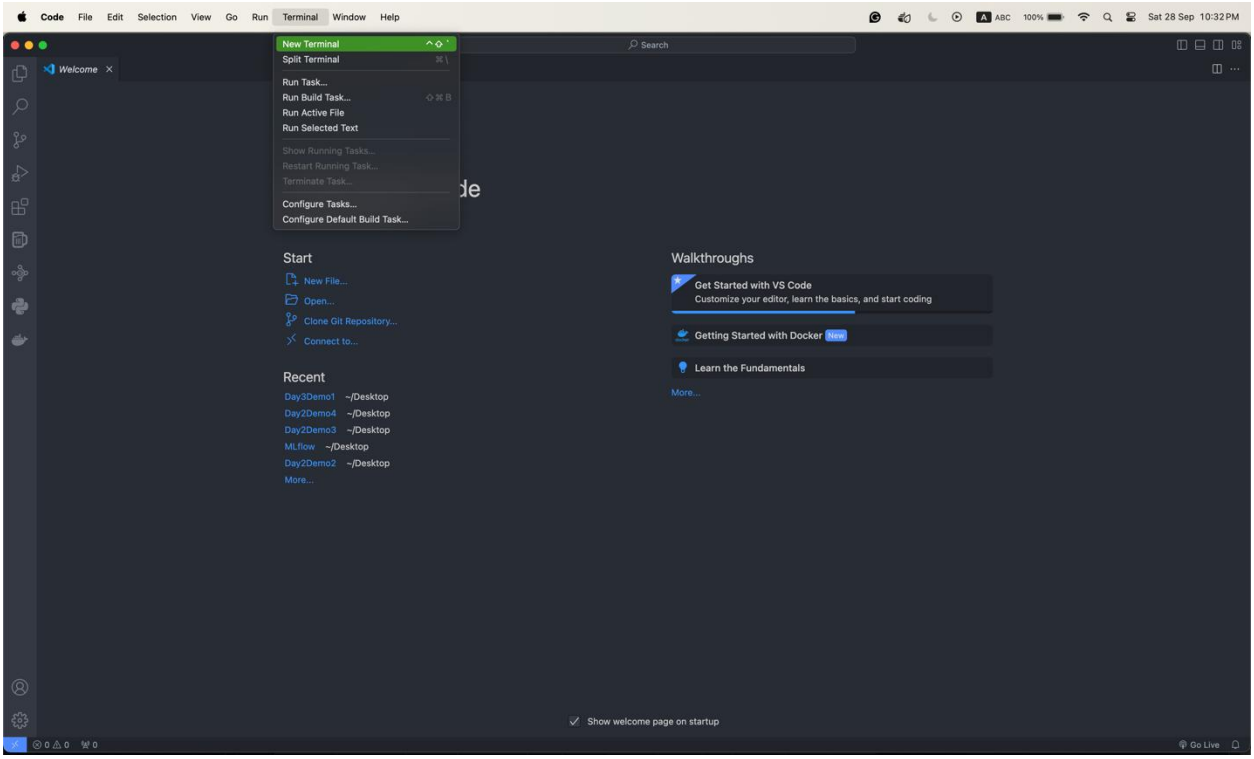
Task 1: Setting up a GitHub repository

In this task, you will learn how to create a new GitHub repository, which will store and manage all the files relevant to this lab. This repository will be the central location for pushing and tracking file changes on GitHub throughout this lab.

1. Open the browser, go to **https://github.com/**, and click the **Sign-in** button on the top right corner of the website.
2. Enter your username, email address, and password, then click the **Sign in** button.
3. Click on the green **New** button to create a new repository.
4. Name your repository (e.g., **mlops-lab01-project**). Enter this description, **This repository demonstrates the setup of an ML project, including GitHub repository initialization, Python virtual environment creation, pre-commit hooks configuration for linting and testing, and Git tracking for version control**. Choose **Public** and initialize it with a **README**.
5. Click on the green **Create Repository button**.
6. Open **Visual Studio Code (VS Code)**.



7. Click on the menu item **Terminal** and then select the sub-menu item **New terminal**.



8. Enter the following command in the terminal with your GitHub username.

```
git clone https://github.com/your-username/mlops-lab01-project.git
```

9. Navigate into the folder by using the following command.

```
cd mlops-lab01-project
```

10. From the VS Code menu, click **File>Open Folder** and select your project folder (C:\Users\student\mlops-lab01-project).

Task 2: Setting up a Python virtual environment

1. Click on the menu item **Terminal** and then select the sub-menu item **New terminal**. Verify that Pip is installed with:

```
pip --version
```

   Verify that virtualenv is installed.

```
virtualenv --version
```

If not, install **virtualenv** using the following command.

```
pip install virtualenv
```

2. In your project folder, run the following command to create the virtual environment.

```
virtualenv venv
```

3. Run the following command to activate the virtual environment.

```
.\venv\Scripts\activate
```

4. Now run the following command to confirm that the virtual environment is activated.

```
python --version
```

The terminal should display the Python version within the virtual environment.

Task 3: Configuring pre-commit hooks for linting and testing

1. While still in the virtual environment, install the pre-commit package using the following command.

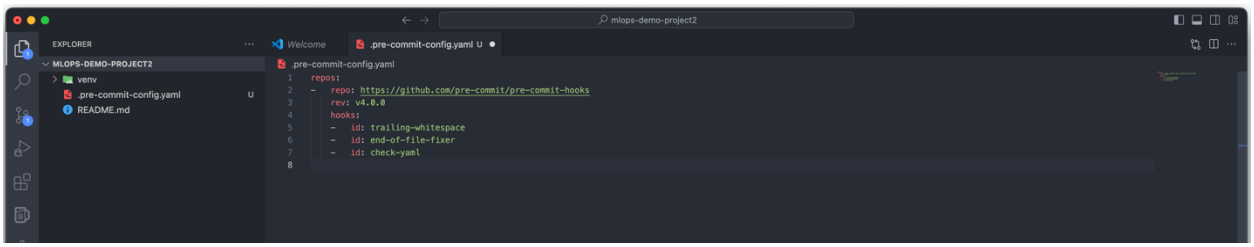
```
pip install pre-commit
```

2. In VS Code, to the right of your project folder - mlops-lab01-project, click the **New File** icon and create a **.pre-commit-config.yaml** file. Type in the name and press **enter**.

> **Caution**
>
> Note the leading **.** (period) in the name!

Type in the contents of the file, as follows:

```
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.0.0
    hooks:
    -   id: trailing-whitespace
    -   id: end-of-file-fixer
    -   id: check-yaml
```



3. Save the file with **File>Save**.
4. From the VS Code, run the following command in the terminal window to install and activate the pre-commit hooks.

```
pre-commit install
```

5. For testing the pre-commit hooks, on your GitHub site, make some minor changes to your project (for example, add extra spaces at the end of a line in the **README.md**) and run the following command to manually trigger the hooks.

```
pre-commit run --all-files
```

   You should see the hooks in action, automatically fixing issues like trailing whitespaces or missing end-of-file newlines.

Task 4: Pushing changes to GitHub with Git tracking

1. Add new files to your repository, create a file in your Project root directory e.g., **myFile.py**, with content of **Hi GitHub from MLOps Training.** In VS Code Explorer, click the **New File** icon to the right of your project folder.
2. For adding this new file to your repository, stage the new changes using this terminal command.

```
git add .
```

3. Run the following command to commit the changes with a message.

```
git commit -m "Check the pre-commit hooks"
```

4. Push the changes to your remote repository using this command.

```
git push origin main
```

5. Go back to your GitHub repository in the browser and refresh the page. You should see your latest commit with all the configured changes.

After review, close the Lab Project folder in VS Code by clicking on **File > Close Folder**.

Lab review

1. What does the command **git push origin main** do?
   A. Stages the changes
   B. Commits the changes locally
   C. Pushes changes to the remote repository
   D. Displays changes in the README.md

STOP

You have successfully completed this lab.

Lab 2: EDA, Feature Engineering, and Data Cleaning

Lab overview

In this lab, you will:

- Perform exploratory data analysis using pandas
- Visualize data distributions using matplotlib and seaborn (histograms, scatter plots, box plots, pair plots, etc.)
- Create new features for better model performance
- Clean the dataset by handling missing values and removing outliers

Estimated completion time

30 minutes

Task 1: Installing required libraries and loading the dataset

In this task, you will learn how to create a new virtual environment for your new project, activate it, and install the required libraries for loading the dataset. Python virtual environments help decouple and isolate Python installs and associated pip packages. This allows end-users to install and manage their own set of packages that are independent of those provided by the system or used by other projects.

1. Create an empty Folder with the name **Lab02** on the desktop of the lab environment.
2. Open the **Visual Studio Code**, click on the **File** menu item, and then click the submenu **Open Folder**. In the browser window, locate the folder that you created in step one and then click **Open**. This will show the folder in the left window of the Visual Studio Code.
3. Now click on the **Terminal** menu item and then click on the sub-menu item, **New Terminal**. This will open a terminal at the bottom to enter commands.
4. Create a new virtual environment for this lab and run the following command in the terminal window to create the virtual environment.

```
virtualenv venv
```

5. Run the following command to activate the virtual environment.

```
.\venv\Scripts\activate
```

6. Now, run the following command to confirm that the virtual environment is activated.

```
python --version
```

7. A new virtual environment setup is completed for this project, to install the required libraries in the venv, run the following command in the terminal window.

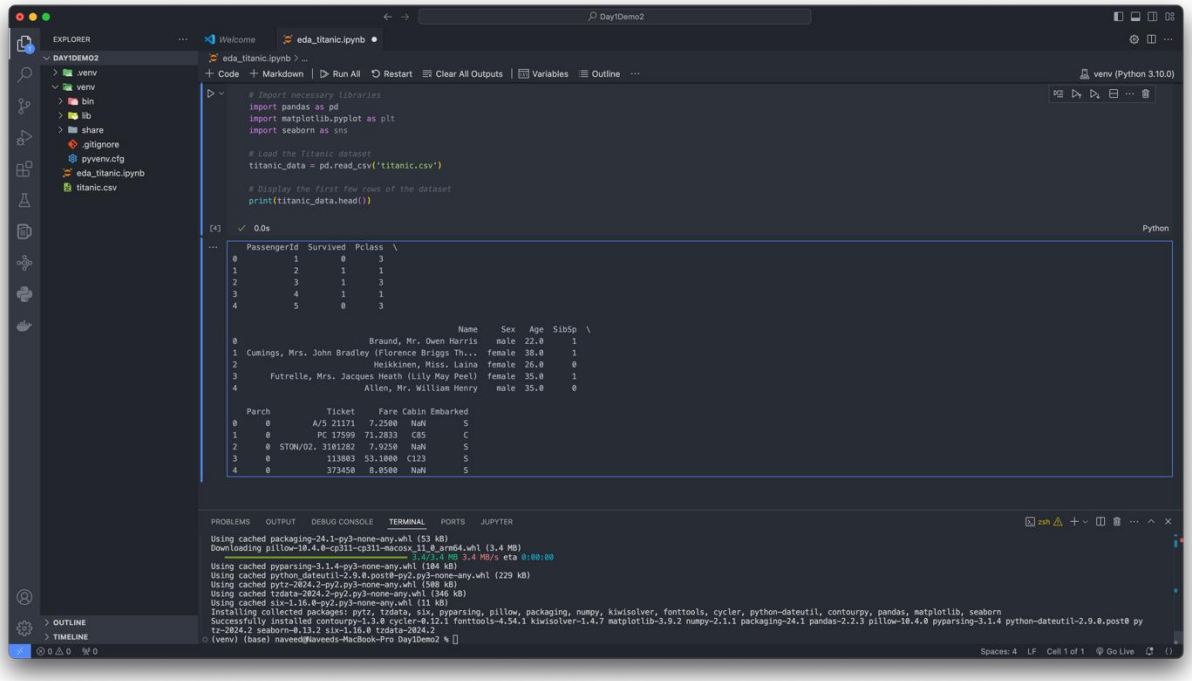
```
pip install pandas matplotlib seaborn ipykernel
```

8. Using Windows File Explorer, copy the dataset file **titanic.csv** into the project folder from the C:\MLOps\Data-Files folder.
9. Copy the **eda_titanic.ipynb** file from the C:\MLOps\Lab-Files\Lab02 folder to the Lab02 Project folder. Both files should then be visible in the VS Code Explorer window.
10. In VS Code, click on the notebook file **eda_titanic.ipynb**, to open the code cells.
11. Read, check and verify that the following code is presented in the first code cell.

```
# Import necessary libraries

import pandas as pd

import matplotlib.pyplot as plt
```

```
import seaborn as sns

# Load the Titanic dataset

titanic_data = pd.read_csv('titanic.csv')

# Display the first few rows of the dataset

print(titanic_data.head())
```

12. Click the **run** button on the left side of the code cell for execution and select the **venv** as the Python environment. After the successful execution of the code, the following output will be displayed.



Task 2: Performing exploratory data analysis (EDA) and visualizing distributions

The main purpose of EDA is to help look at data before making any assumptions. It can help identify obvious errors, as well as better understand patterns within the data, detect outliers or anomalous events, and find interesting relations among the variables.

1. Start with understanding the dataset by summarizing it, looking for missing values and unique values, and checking the basic statistics. Read, check, and verify that the following code is presented in the second code cell.

```
# Summary of the dataset
print("\nDataset Information:")
print(titanic_data.info())

# Check for missing values
print("\nMissing values in each column:")
print(titanic_data.isnull().sum())

# Statistical summary
print("\nStatistical Summary:")
print(titanic_data.describe())
```

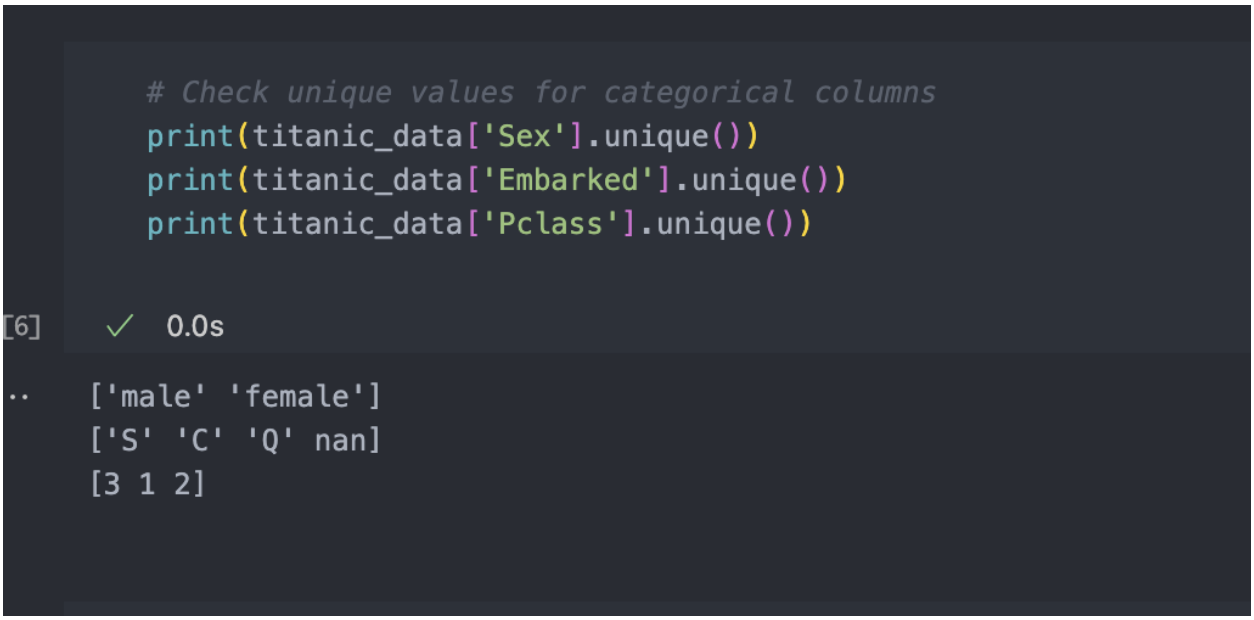
2. Execute this cell. After successful execution, it will provide the basic information about the dataset and display the following output.

```
Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
None

Missing values in each column:
PassengerId      0
...
25%      0.000000    7.910400
50%      0.000000   14.454200
75%      0.000000   31.000000
max      6.000000  512.329200
```

3. The next step is checking the unique values for categorical columns like **Sex**, **Embarked**, and **Pclass**, which should be inspected for unique values. To do this, read, check, and verify that the following code is presented in the next code cell:

```
# Check unique values for categorical columns
print(titanic_data['Sex'].unique())
print(titanic_data['Embarked'].unique())
print(titanic_data['Pclass'].unique())
```

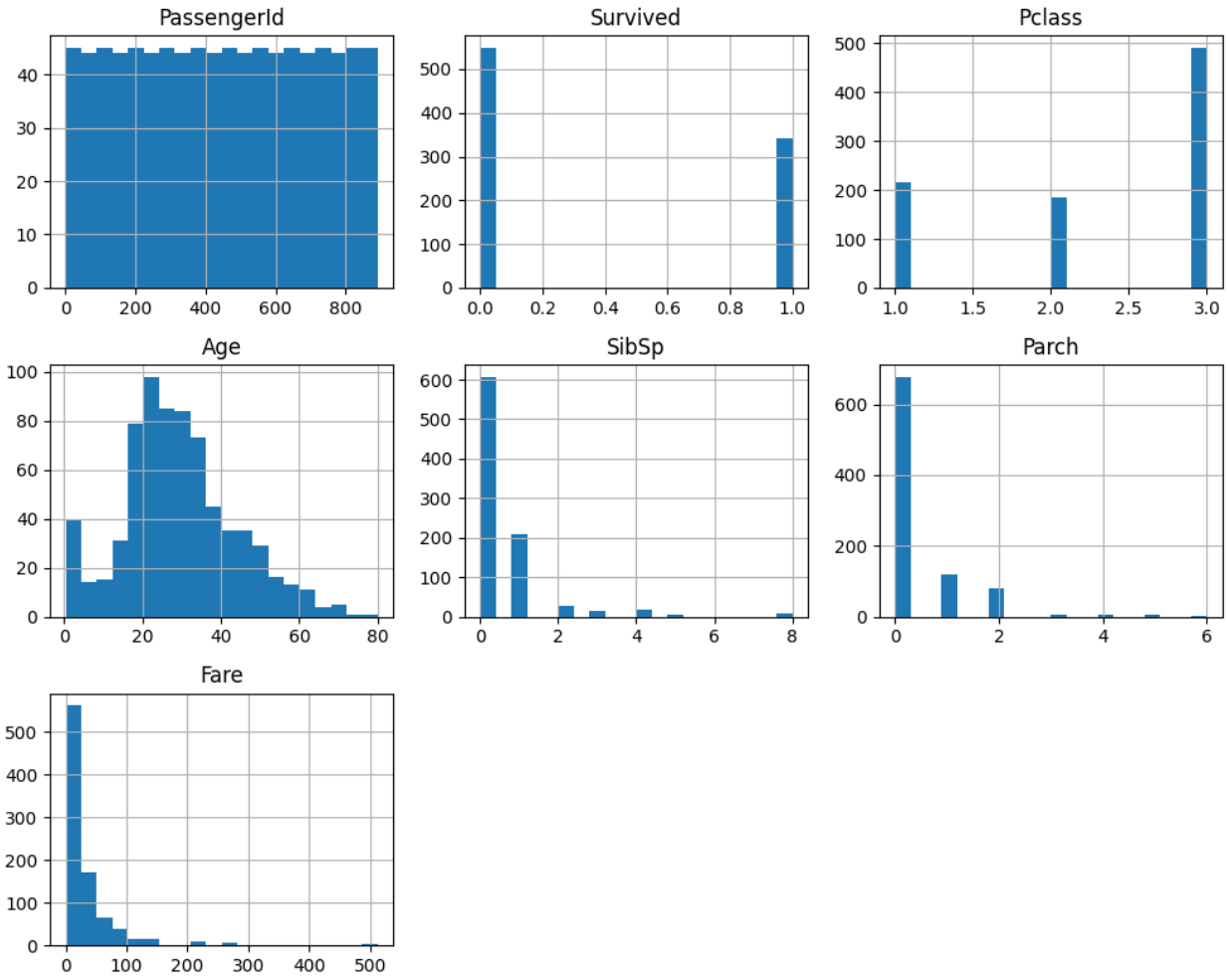
4. Execute this code. The cell will display the unique values for the selected columns from the dataset and display the following output.



5. Visualizing the dataset is an important step in the EDA process. We can visualize and understand the dataset with the help of various plots. Let's start with plotting the histogram for numerical features, as histograms are essential to visualize the distribution of numerical columns like age, fare, etc. Read, check, and verify that the following code is presented in the next code cell.

```
# Plot histogram for numerical columns

titanic_data.hist(bins=20, figsize=(10, 8))

plt.tight_layout()

plt.show()
```

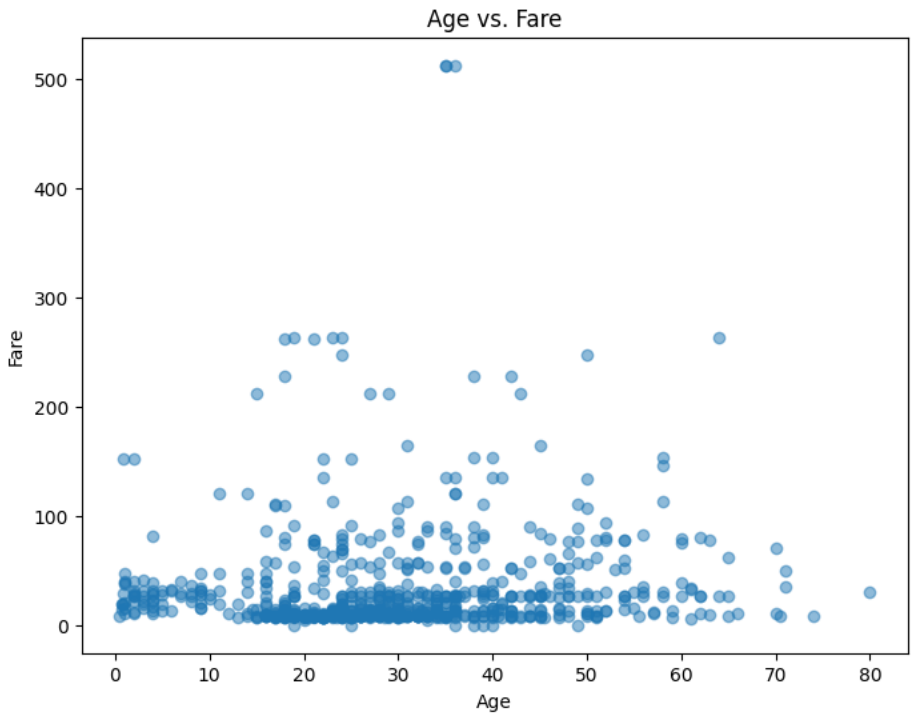
6. Execute the code to show the histograms for numerical features, the output of this step is given below.



7. To find the relationship between different continuous variables, the scatter plots are used in the EDA process. The next code cell is used to generate the scatter plot for age and fare variables in the dataset. Verify the cell contents as follows.

```
# Scatter plot to visualize Age vs. Fare

plt.figure(figsize=(8,6))

plt.scatter(titanic_data['Age'], titanic_data['Fare'], alpha=0.5)

plt.title('Age vs. Fare')

plt.xlabel('Age')

plt.ylabel('Fare')

plt.show()
```

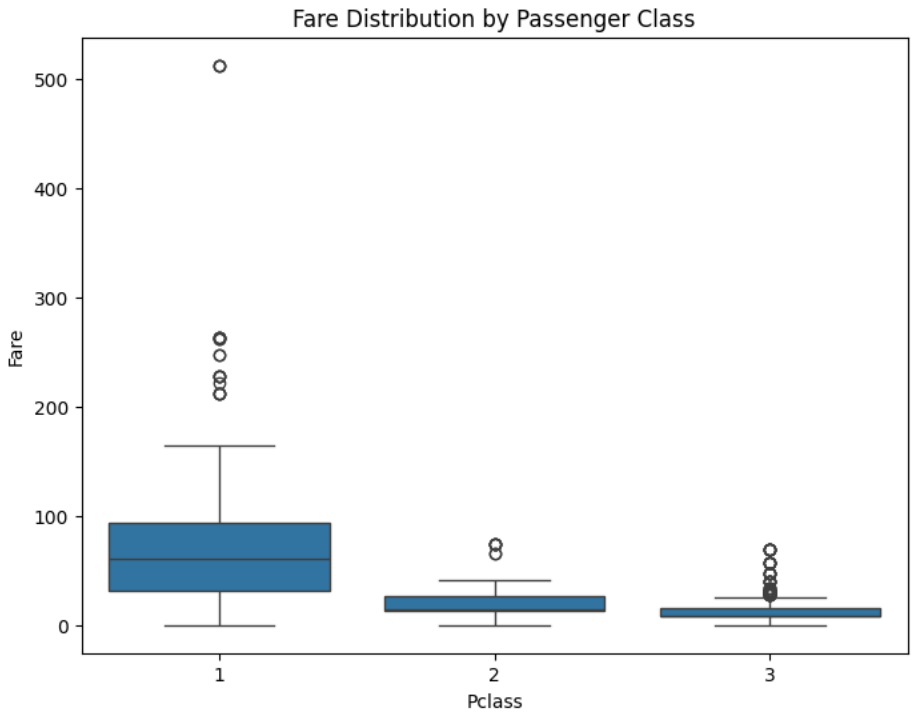
8. Executing the above code cell will provide the following scatter plot as output to show the relationship between the selected columns.



9. Boxplot is another visualization facility to help identify outliers. Let's create a boxplot and show the distribution of fare by passenger class. Read, check, and verify code in the next cell.

```
# Boxplot to compare fare by class

plt.figure(figsize=(8,6))

sns.boxplot(x='Pclass', y='Fare', data=titanic_data)

plt.title('Fare Distribution by Passenger Class')

plt.show()
```

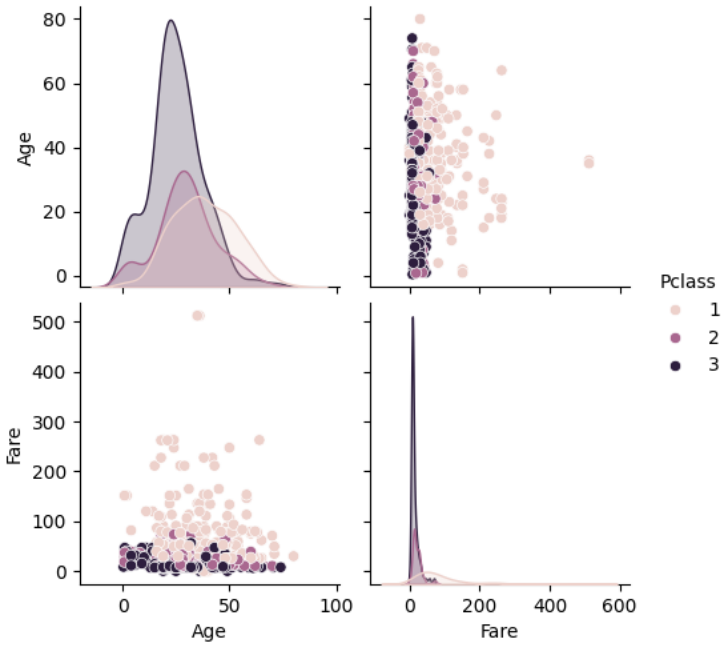
10. Executing the above cell will result in displaying the following boxplot for selected variables in the dataset.



11. The next step in the EDA process is to inspect the relationship between multiple variables in the dataset. Let's select the three variables age, pclass, and fare to find the relationship using the pair plot. Read, check, and verify code in the next cell.

```
# Pair plot for continuous variables

sns.pairplot(titanic_data[['Age', 'Fare', 'Pclass']], hue='Pclass',
diag_kind='kde')

plt.show()
```

12. Executing the above code cell will display the following pair plot to provide the relationship between the selected continuous multiple variables.



Task 3: Feature engineering and data cleaning

Feature engineering aims to prepare an input data set that best fits the machine learning algorithm as well as to enhance the performance of machine learning models.

1. Create new features such as FamilySize by combining existing columns like SibSp (siblings/spouses) and Parch (parents/children). Verify the code in the next cell.

```
# Creating a new feature 'FamilySize'

titanic_data['FamilySize'] = titanic_data['SibSp'] +
titanic_data['Parch'] + 1

# Check the first few rows to confirm the new feature

print(titanic_data[['SibSp', 'Parch', 'FamilySize']].head())
```

2. Execute the above cell. It will create new features in the dataset and display with the head() function. The following is the output of the code.

```
# Creating a new feature 'FamilySize'
titanic_data['FamilySize'] = titanic_data['SibSp'] + titanic_data['Parch'] + 1

# Check the first few rows to confirm the new feature
print(titanic_data[['SibSp', 'Parch', 'FamilySize']].head())
```

[12] ✓ 0.0s

```
   SibSp  Parch  FamilySize
0      1      0           2
1      1      0           2
2      0      0           1
3      1      0           2
4      0      0           1
```

3. We can also extract titles from names as a new categorical feature, let's check and run the code in the next cell.

```
# Extracting title from names

titanic_data['Title'] = titanic_data['Name'].str.extract(' ([A-Za-z]+)', expand=False)


# Check for unique titles

print(titanic_data['Title'].unique())
```

4. Executing the above cell will result in extracting the titles from names and providing the following output.

```
# Extracting title from names
titanic_data['Title'] = titanic_data['Name'].str.extract(' ([A-Za-z]+)\.', expand=False)

# Check for unique titles
print(titanic_data['Title'].unique())
```

[13] ✓ 0.0s

```
['Mr' 'Mrs' 'Miss' 'Master' 'Don' 'Rev' 'Dr' 'Mme' 'Ms' 'Major' 'Lady'
 'Sir' 'Mlle' 'Col' 'Capt' 'Countess' 'Jonkheer']
```

5. We know that columns like Age, Embarked, and Cabin have missing values. Check and run the next code cell that is handling the missing values in the selected columns.

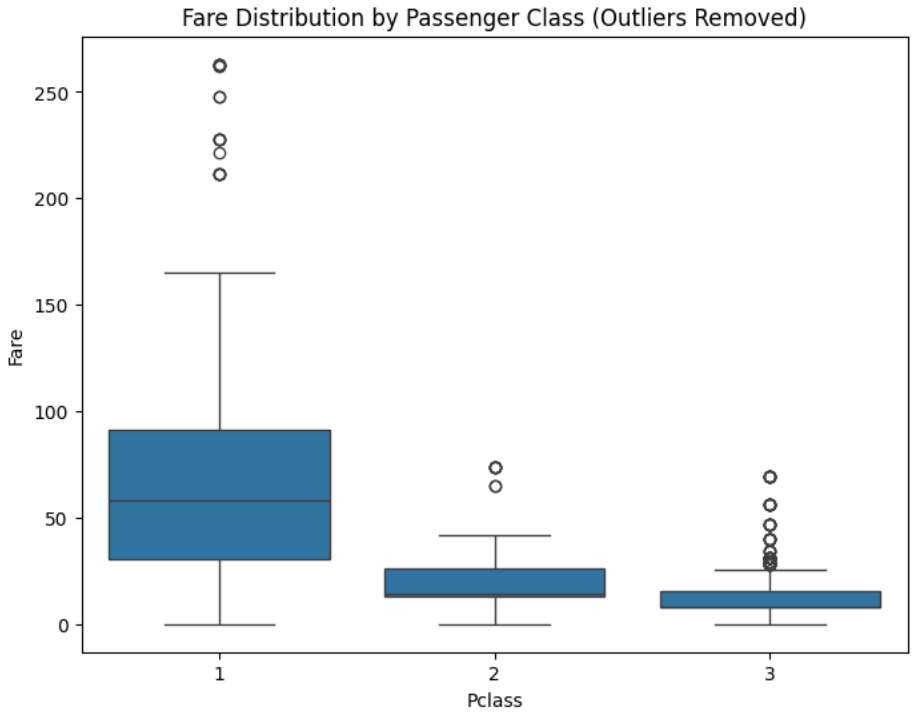
```
# Replace missing values in 'Age' with the median age

titanic_data['Age'] =
titanic_data['Age'].fillna(titanic_data['Age'].median())


# Replace missing values in 'Embarked' with the most common port

titanic_data['Embarked'] =
titanic_data['Embarked'].fillna(titanic_data['Embarked'].mode()[0])


# Drop 'Cabin' column since it has too many missing values

titanic_data.drop(columns=['Cabin'], inplace=True)


# Confirm that missing values have been addressed

print("\nMissing values after cleaning:")

print(titanic_data.isnull().sum())
```

Executing the above code will result in handling the missing values in the dataset.

6. Outliers can significantly impact statistical measures and machine learning models, leading to misleading conclusions and poor model performance. Remove the outliers in the Fare column by setting a threshold. Verify the following code in the next code cell.

```
# Remove outliers in 'Fare'

titanic_data = titanic_data[titanic_data['Fare'] < 300]


# Plot the updated boxplot for Fare by Passenger Class

plt.figure(figsize=(8,6))

sns.boxplot(x='Pclass', y='Fare', data=titanic_data)

plt.title('Fare Distribution by Passenger Class (Outliers Removed)')

plt.show()
```

7. Executing the above code will remove the outliers and show the following plot.



8. Finally, save the cleaned dataset as a new CSV file for future use. Check and run the code in the next cell.

```
# Save the cleaned dataset to a new CSV file

titanic_data.to_csv('titanic_cleaned.csv', index=False)


print("Cleaned dataset saved as 'titanic_cleaned.csv'")
```

   Executing the above code will create a new csv file in the project folder with the name **titanic_cleaned.csv** that can be used later.

9. Load the cleaned dataset and do some more visualizations. Verify and run the code in the next cell.

```
titanic_cleaned = pd.read_csv('titanic_cleaned.csv')

titanic_cleaned.head()
```

10. Executing the above code cell will result in loading the cleaned dataset file in the new variable **titanic_cleaned**. Following is the output of the above code.

```
titanic_cleaned = pd.read_csv('titanic_cleaned.csv')
titanic_cleaned.head()
```

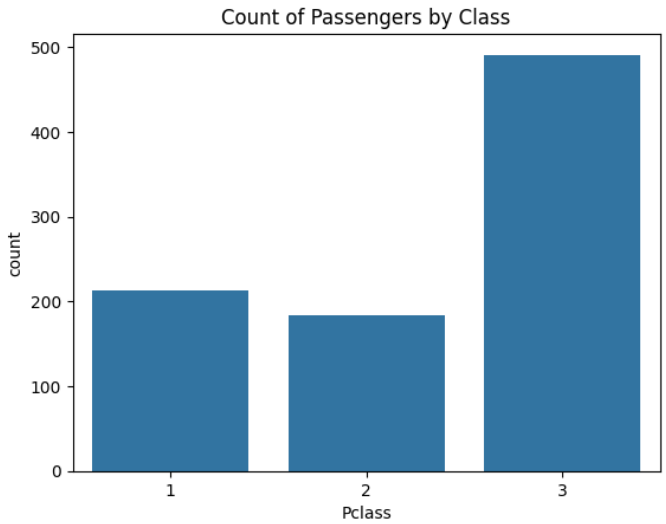
✓ 0.0s

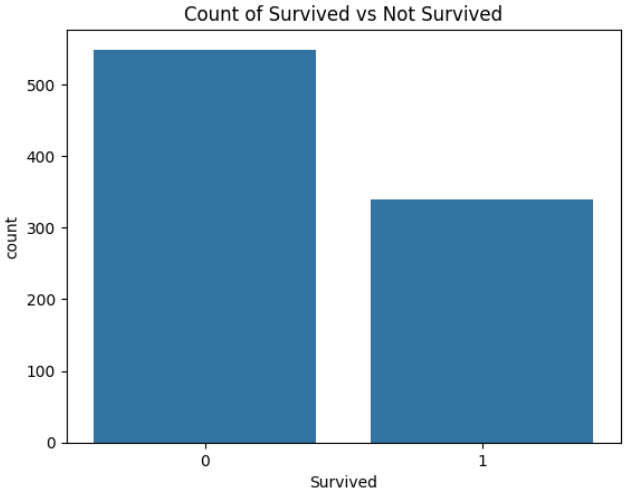
	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Embarked	FamilySize	Title
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	S	2	Mr
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C	2	Mrs
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	S	1	Miss
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	S	2	Mrs
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	S	1	Mr

11. Visualize the cleaned dataset using the countplot, which is useful for visualizing the distribution of categorical features. Check and run the code in the next cell.

```
# Countplot for Pclass

sns.countplot(x='Pclass', data=titanic_cleaned)

plt.title('Count of Passengers by Class')

plt.show()


# Countplot for Survived

sns.countplot(x='Survived', data=titanic_data)

plt.title('Count of Survived vs Not Survived')

plt.show()
```

Executing the above code cell will result in displaying the following plot.





12. Create a violin plot, which is a combination of box plots and KDE (Kernel Density Estimate), useful for visualizing distributions. Finally, check and run the code in the last cell.

```
# Violin plot for Age distribution by survival status
sns.violinplot(x='Survived', y='Age', data=titanic_cleaned)
plt.title('Age Distribution by Survival Status')
plt.show()
```

Executing the above code will display the following plot.

13. After reviewing the code and outputs, close the Lab Project folder in VS Code by clicking on **File > Close Folder**. If asked, save the ipynb file.

Lab review

1. What command is used to save the cleaned dataset as a new CSV file?
   A. titanic_data.save('titanic_cleaned.csv')
   B. titanic_data.to_csv('titanic_cleaned.csv')
   C. titanic_data.write('titanic_cleaned.csv')
   D. titanic_data.export('titanic_cleaned.csv')

STOP

You have successfully completed this lab.

Lab 3: Build and Automate a Basic ML Pipeline

Lab overview

In this lab, you will:

- Automate data preprocessing and feature selection
- Train and evaluate an ML model
- Use metrics to evaluate model performance
- Save the model using joblib

Estimated completion time

30 minutes

Task 1: Preparing data

In this task, you will load the cleaned Titanic dataset, separate features, and the target variable and split the data into training and test sets.

1. Create an empty folder with the name **Lab03** on the desktop of lab environment.
2. Open the **Visual Studio Code**, click on the **File** menu item, and then click the submenu **Open Folder**. In the browser window, locate the folder that you created in step one and then click **Open**. This will show the folder in the left window of the Visual Studio Code.
3. Now click on the **Terminal** menu item and then click on the sub-menu item, **New Terminal**. This will open a terminal at the bottom to enter commands.
4. Now, you will create a new virtual environment for this lab and run the following command in the terminal window to create the virtual environment.

```
virtualenv venv
```

5. Run the following command to activate the virtual environment.

```
.\venv\Scripts\activate
```

6. Now, run the following command to confirm that the virtual environment is activated.

```
python --version
```

7. A new virtual environment setup is completed for this project. To install the required libraries in the venv, run the following command in the terminal window.

```
pip install pandas matplotlib seaborn scikit-learn joblib ipykernel
```

8. Using Windows File Explorer, copy the dataset file **titanic_cleaned.csv** into the project folder from the Desktop/Lab02 Project folder.
9. Now copy the **ml_pipeline.ipynb** file from the C:\MLOps\Lab-Files\Lab03 folder to the Lab03 project folder. Both files should then be visible in the VS Code Explorer window.
10. In VS Code, click on the notebook file **ml_pipeline.ipynb**, to open the code cells.
11. Read, check, and verify that the following code is presented in the first code cell.

```
# Import necessary libraries

import pandas as pd


# Load the cleaned Titanic dataset

data = pd.read_csv('titanic_cleaned.csv')
```

```
# Display the first few rows of the dataset

print("First 5 rows of the dataset:")

display(data.head())
```

12. Click the **run** button on the left side of the code cell for execution and select the recommended Python environment. After the successful execution of the code, the following output will be displayed.

```
# Import necessary libraries
import pandas as pd

# Load the cleaned Titanic dataset
data = pd.read_csv('titanic_cleaned.csv')

# Display the first few rows of the dataset
print("First 5 rows of the dataset:")
display(data.head())
```

✓ 0.0s

```
First 5 rows of the dataset:
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Embarked	FamilySize	Title
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	S	2	Mr
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C	2	Mrs
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	S	1	Miss
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	S	2	Mrs
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	S	1	Mr

13. Separate the dataset into features (**X**) and the target variable (**y**). Check and execute the code in the next cell.

```
# Separate features and target variable

X = data.drop(['Survived'], axis=1)

y = data['Survived']
```

14. Use an 80-20 split, with **80%** of the data being used for training and **20%** for testing. Verify the code in the next cell and run it.

```
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Task 2: Data preprocessing and feature selection

In this task, you will preprocess the data by scaling numerical features and encoding categorical features. We will also perform feature selection to retain the most relevant features.

1. First identify which columns are categorical and which are numerical for preprocessing. Verify and run the following code in the next cell.

```
# Select categorical and numerical columns for preprocessing

categorical_cols = ['Pclass', 'Sex', 'Embarked']

numerical_cols = ['Age', 'Fare', 'FamilySize']
```

2. For preprocessing, use StandardScaler for scaling the numerical columns and OneHotEncoder for encoding categorical columns. Check the contents of the next cell and then run it.

```
from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

# Define preprocessing for numeric features (scaling) and categorical
features (one-hot encoding)

numerical_transformer = StandardScaler()

categorical_transformer = OneHotEncoder(handle_unknown='ignore')

# Create a column transformer for applying transformations

preprocessor = ColumnTransformer(

    transformers=[

        ('num', numerical_transformer, numerical_cols),

        ('cat', categorical_transformer, categorical_cols)

    ])
```

3. For feature selection, use SelectKBest to select the top k features based on the ANOVA F-value between each feature and the target. Check the next code cell contents and execute it.

```
from sklearn.feature_selection import SelectKBest, f_classif

# Feature selection: Select the top k features

feature_selector = SelectKBest(score_func=f_classif, k=8)
```

Task 3: Training a model

In this task, you will build a RandomForest model and train it on the preprocessed and selected features.

1. Use a RandomForestClassifier model for this task. Verify and run the code in the next cell.

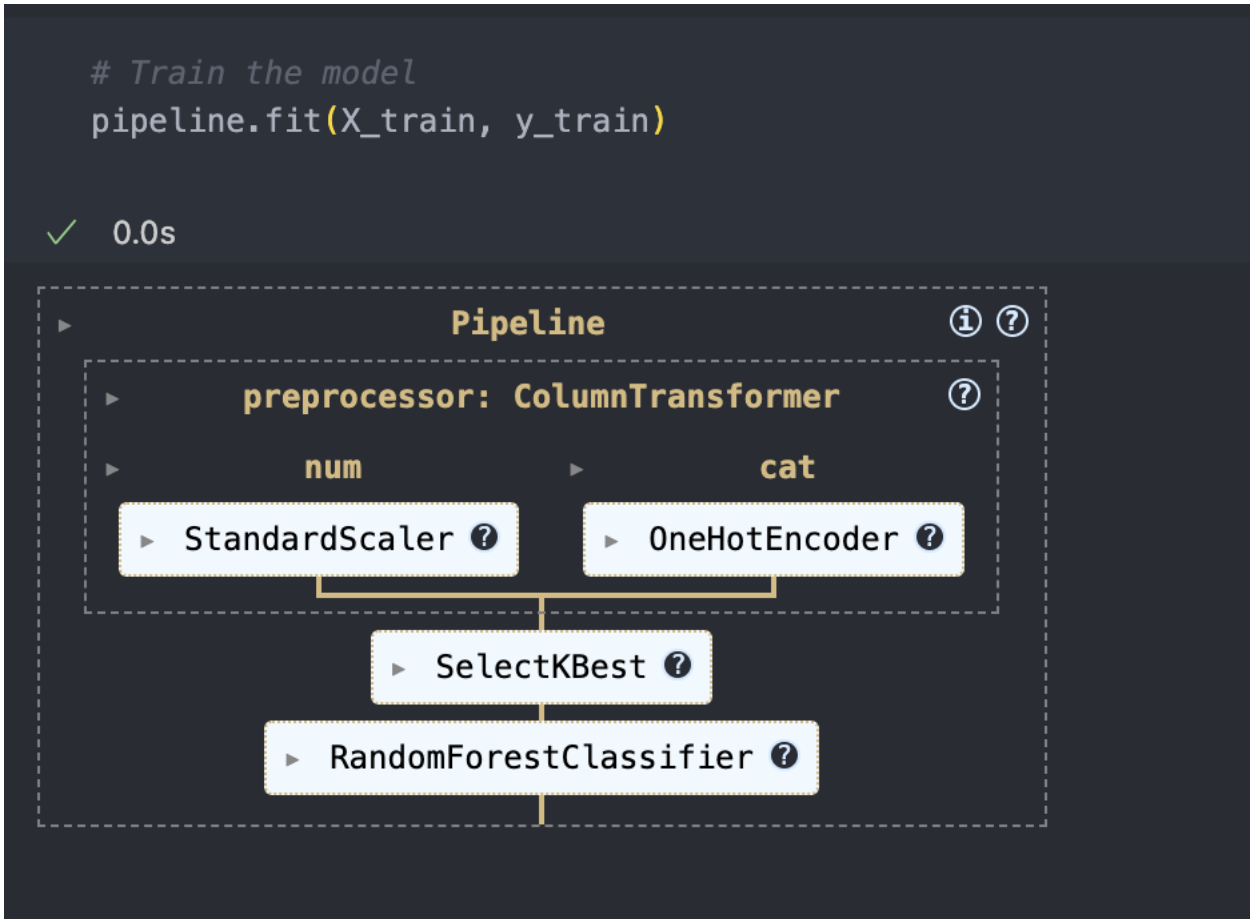
```
from sklearn.ensemble import RandomForestClassifier

# Define the model

model = RandomForestClassifier(random_state=42)
```

2. Combine data preprocessing, feature selection, and the model into a single pipeline. Confirm the next code cell contents and then run it.

```
from sklearn.pipeline import Pipeline

# Create a pipeline that combines the preprocessor, feature selector, and the model

pipeline = Pipeline(steps=[

    ('preprocessor', preprocessor),

    ('feature_selector', feature_selector),

    ('classifier', model)

])
```

3. To train the model on the training data, use the **fit** method with the defined pipeline. Check and run the following content in the next cell.

```
# Train the model

pipeline.fit(X_train, y_train)
```



Task 4: Evaluating and saving a model

This task involves evaluating the trained model using various metrics and saving the model for later use.

1. For the evaluation purpose of the model, use metrics, e.g., **Accuracy**, **Precision**, **Recall**, and **F1-Score**. Verify the content in the next code cell. Run the cell.

```
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score


# Predict on the test set
```

```
y_pred = pipeline.predict(X_test)


# Model evaluation
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)


print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1:.2f}")
```

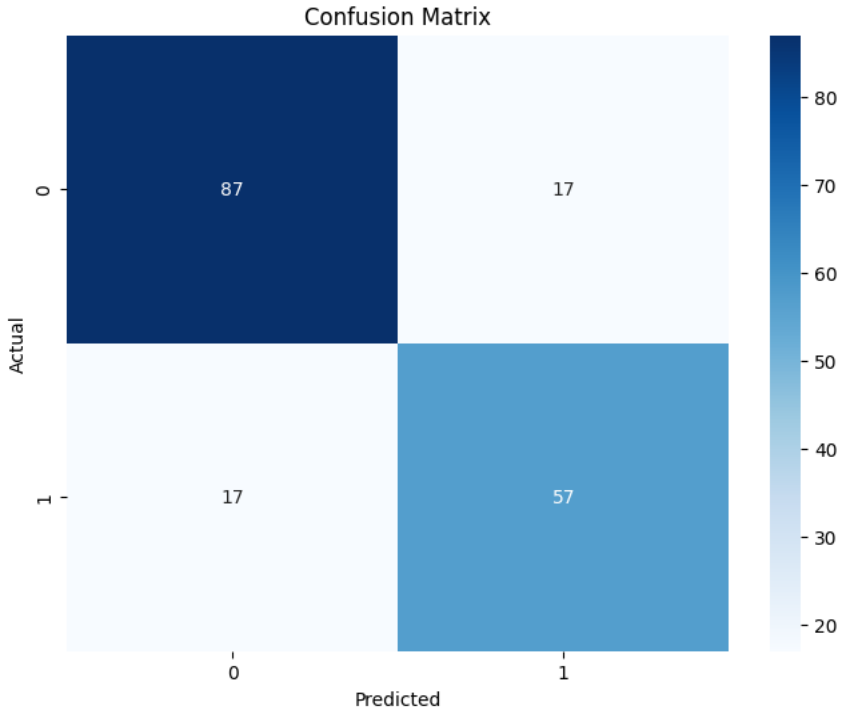
You will see the following output after the execution of the above cell.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Predict on the test set
y_pred = pipeline.predict(X_test)

# Model evaluation
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1:.2f}")

✓ 0.0s
Accuracy: 0.81
Precision: 0.77
Recall: 0.77
F1 Score: 0.77
```

2. You can also visualize the **confusion_matrix** to get a better understanding of the model's performance. Use the following code in the next cell, check and run the code.

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt


# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

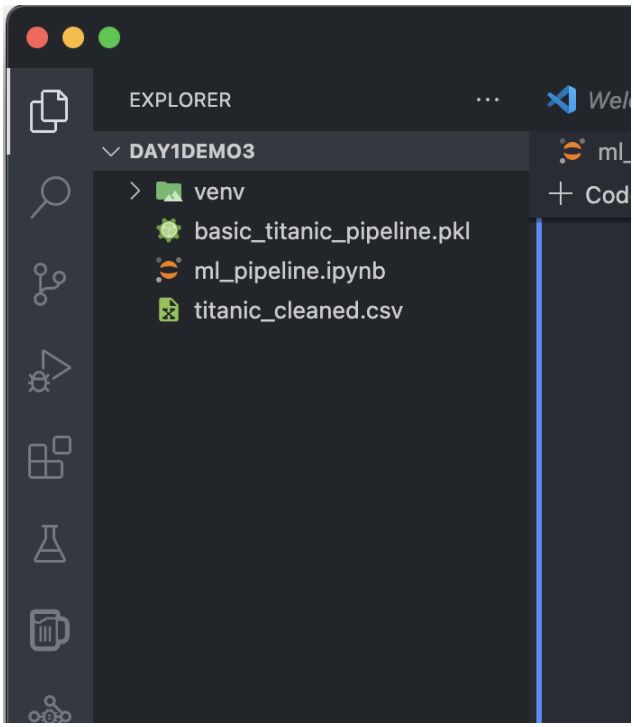
The following output will be displayed after executing the above code cell.



3. Finally, save the trained pipeline using **joblib** for future use. Check the code in the final cell and run it.

```
import joblib


# Save the trained pipeline to a file

joblib.dump(pipeline, 'basic_titanic_pipeline.pkl')

print("Trained model saved as 'basic_titanic_pipeline.pkl'")
```

4. After executing this code, a new file will be created in the project folder with the name **basic_titanic_pipeline.pkl** as the saved model for this lab.



Lab review

1. What is the final output of the trained model in this lab?
   A. A .csv file containing predictions
   B. A .pkl file containing the trained pipeline
   C. A .txt file containing model metrics
   D. A .json file containing feature importance

STOP

You have successfully completed this lab.

Challenge Lab 1: Create an End-to-End MLOps Pipeline – From Preprocessing to GitHub Automation with the Iris Dataset

Lab overview

In this lab, you will:

- Set up an ML project with GitHub
- Perform data exploration, preprocessing, and feature engineering
- Automate the ML pipeline using GitHub Actions
- Train and evaluate an ML model within the Pipeline

Estimated completion time

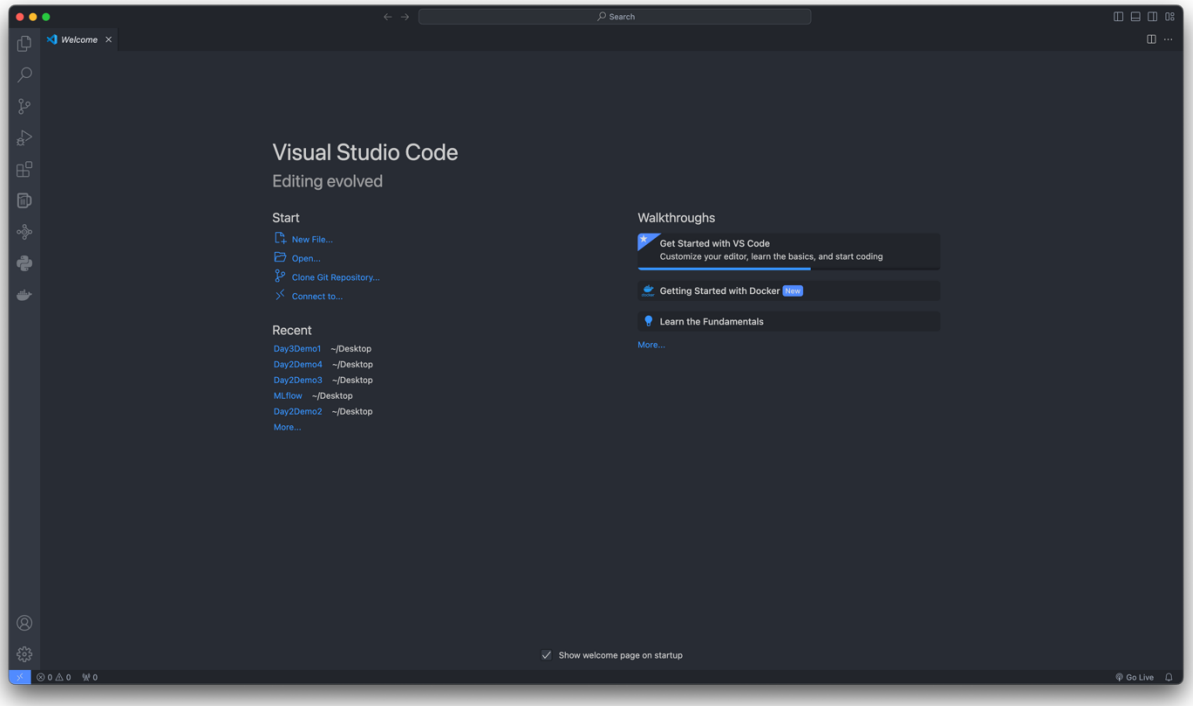
45 minutes

> **Note**
>
> You need a GitHub account to perform this lab.

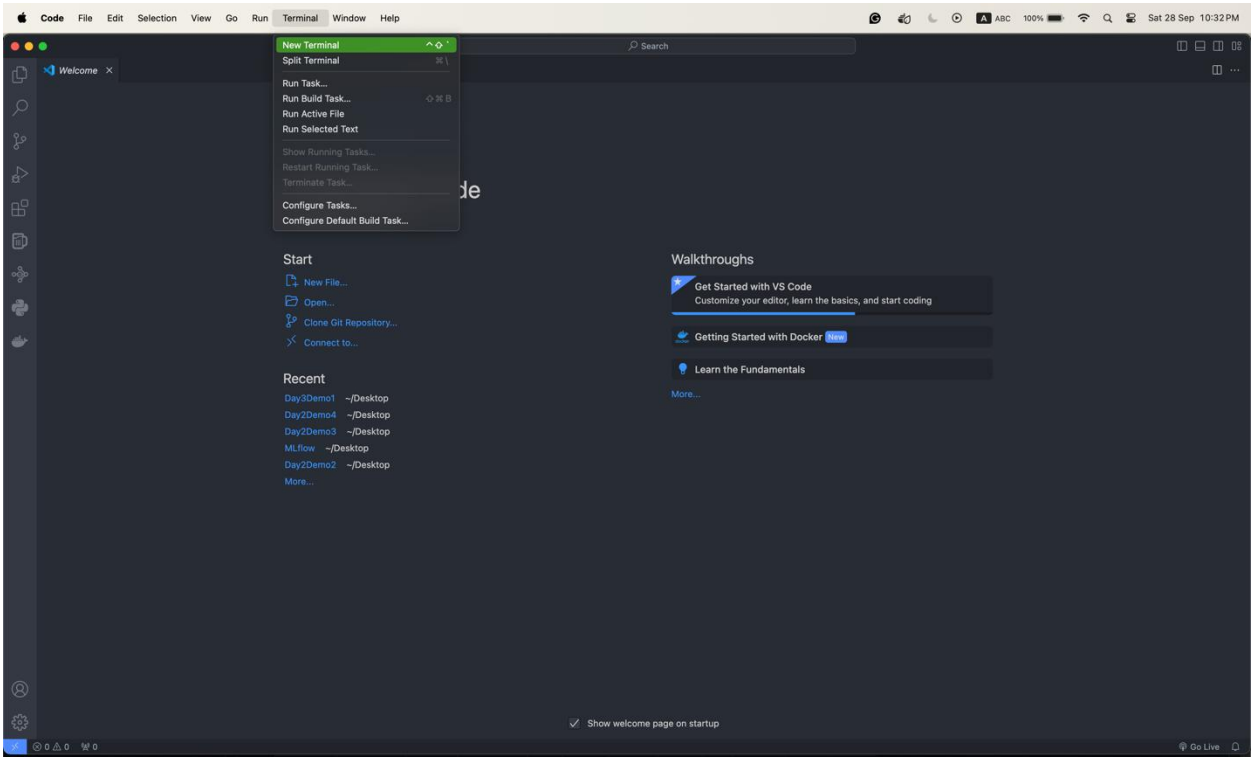
Task 1: Initializing an ML project in GitHub

In this task, you will learn to initialize a machine learning project, set up the necessary environment and dependencies, and ensure code quality by configuring pre-commit hooks for linting.

1. Open the browser, go to **https://github.com/,** and click the **Sign-in** button on the top right corner of the website.
2. Enter your username, email address, and password, then click the **Sign in** button.
3. Click on the green **New** button to create a new repository.
4. Name your repository (e.g., **mlops_Lab**). Enter this description: **This repository demonstrates the setup of an ML project, including GitHub repository initialization, Python virtual environment creation, pre-commit hooks configuration for linting and testing, and Git tracking for version control.** Choose **Public** and initialize it with a **README**.
5. Click on the green **Create Repository** button.
6. Open **VS Code**.



7. Click on the menu item **Terminal** and then select the sub-menu item **New Terminal.**



8. Enter the following command in the terminal using your GitHub username.

```
git clone https://github.com/your-username/mlops_Lab.git
```

9. Navigate into the folder by using the following command.

```
cd mlops_Lab
```

10. Install **virtualenv** if not already installed using the following command.

```
pip install virtualenv
```

11. In your project folder, create a virtual environment, activate it and verify the activation by confirming that the terminal displays the Python version.
12. Using **pip**, install the libraries needed for the project and save them to a **requirements.txt** file in the project folder. The required libraries are:

```
pandas scikit-learn matplotlib seaborn ipykernel
```

13. Commit and push changes to Github with the following commands.

```
git add .

git config --global user.email "you@example.com"

git commit -m "Initial setup with the virtual environment and pre-commit hooks"

git push origin main
```

Task 2: Loading the Iris dataset and performing exploratory data analysis (EDA)

Explore and visualize the dataset using various types of plots to understand the underlying patterns in the data. Perform a detailed analysis of the features and identify potential relationships between variables.

1. Load the dataset and display the first few rows to confirm it's loaded correctly. Open the mlops_Lab folder in VS Code. Create a new IPYNB file (mlops_Lab.ipynb), create a new code cell, enter the following code and execute it.

```
import pandas as pd

import matplotlib

import matplotlib.pyplot as plt

# Load Iris dataset

data = pd.read_csv('https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv')

# Display the first few rows

print(data.head())
```

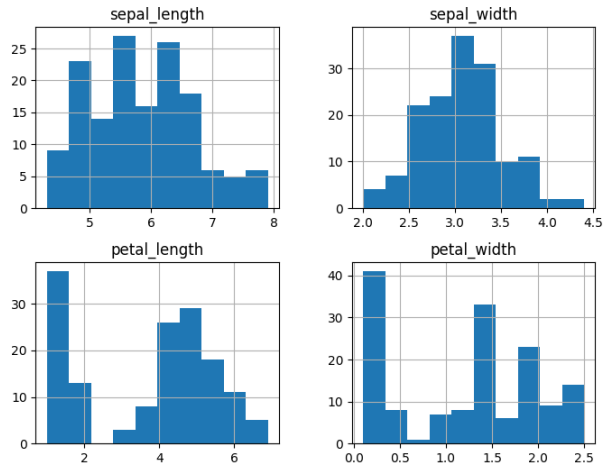
2. Analyze the dataset structure and look for missing values, unique values, and basic statistical summaries. Create a new code cell, enter the following code, and execute it.

```
# Summary of the dataset

print(data.info())

print(data.describe())

print("Missing values in the dataset:\n", data.isnull().sum())
```

3. Use histograms, pair plots, scatter plots, and box plots to understand the data distribution and identify any potential outliers. Create a new code cell, enter the following code, and execute it to visualize the distribution of each numerical feature.

```
data.hist(figsize=(8, 6))

plt.show()
```

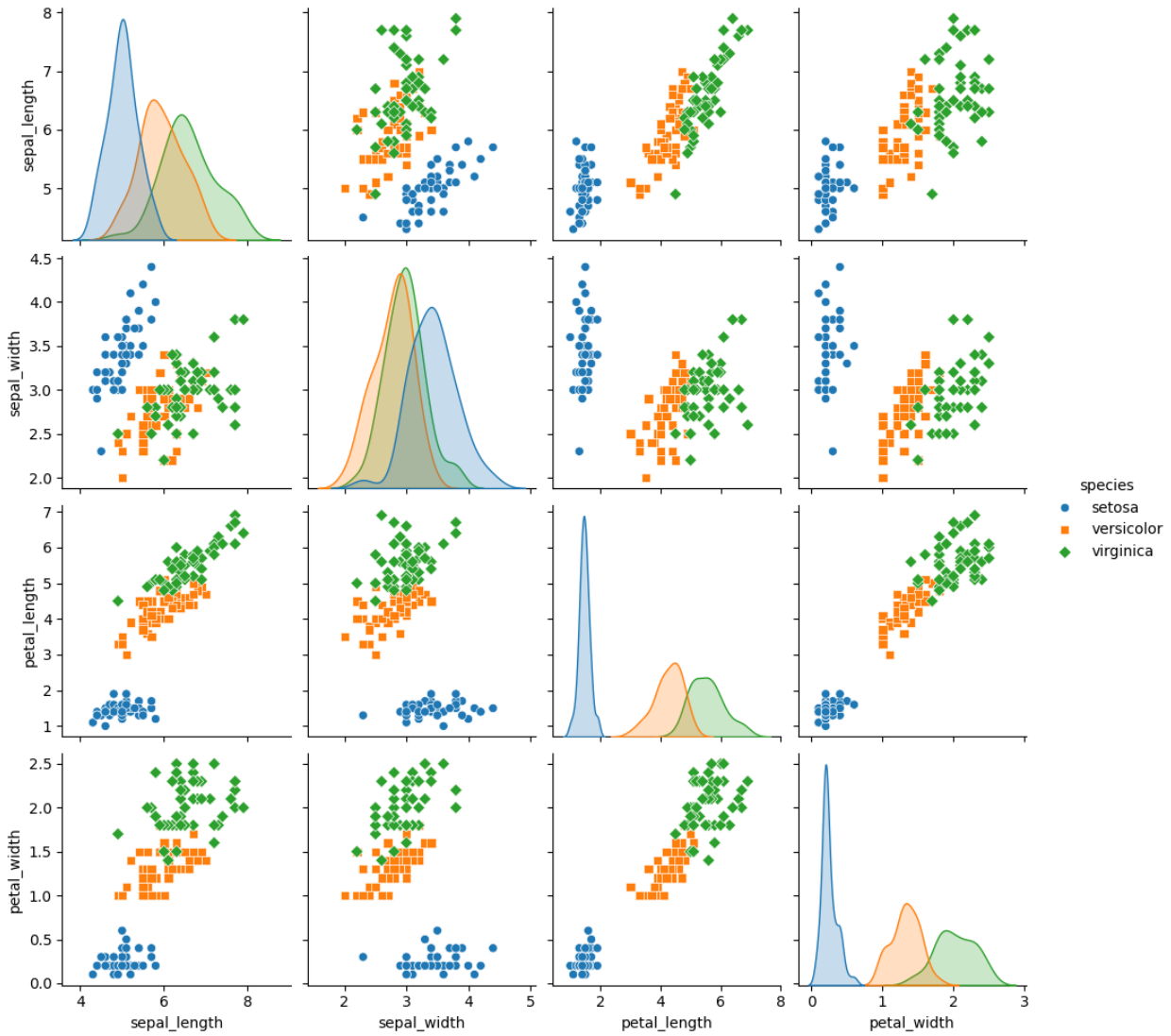
The following plots will be shown as output when you execute the above code.



4. To visualize the relationship between all pairs of features, use the pair plot. Create a new code cell, enter the following code, and execute it.

```
import seaborn as sns

sns.pairplot(data, hue='species', markers=["o", "s", "D"])

plt.show()
```

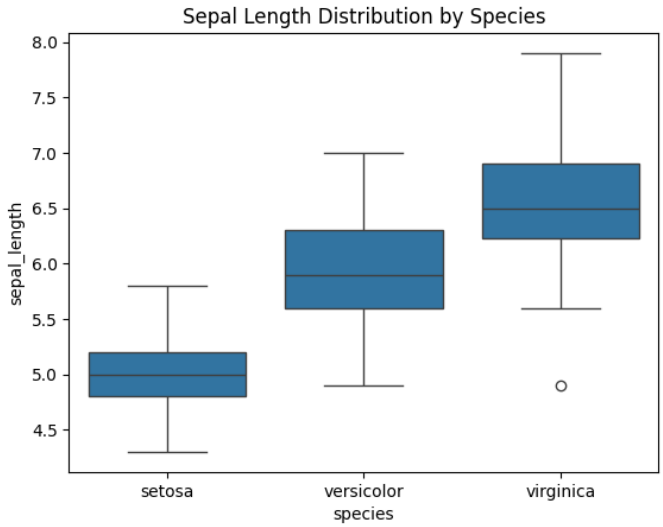
When you execute the above code, the following plot will be displayed as the output.



5. Boxplots are useful for identifying outliers within each species. Create a new code cell, and enter the following code to display the boxplot for selected features.

```
sns.boxplot(x='species', y='sepal_length', data=data)

plt.title('Sepal Length Distribution by Species')

plt.show()
```

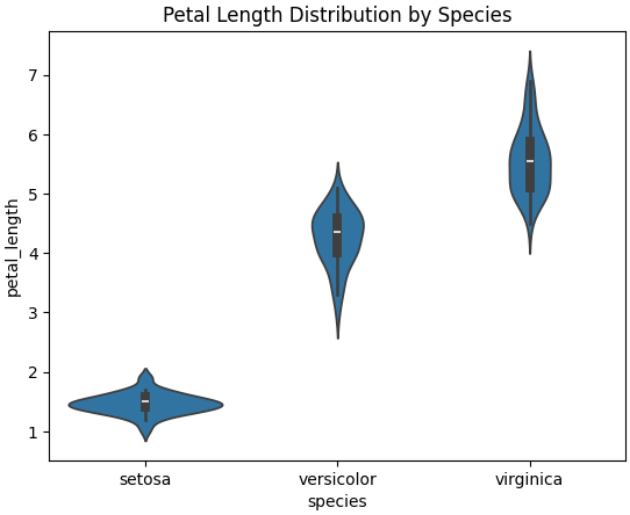
Executing the above code will display the following plot as the output.



6. A violin plot allows comparison of the distribution of features across species. To generate the violin plot for some features in the dataset, create a new code cell and enter the following code in it.

```
sns.violinplot(x='species', y='petal_length', data=data)

plt.title('Petal Length Distribution by Species')

plt.show()
```

Executing the above code will result in displaying the following plot as output.



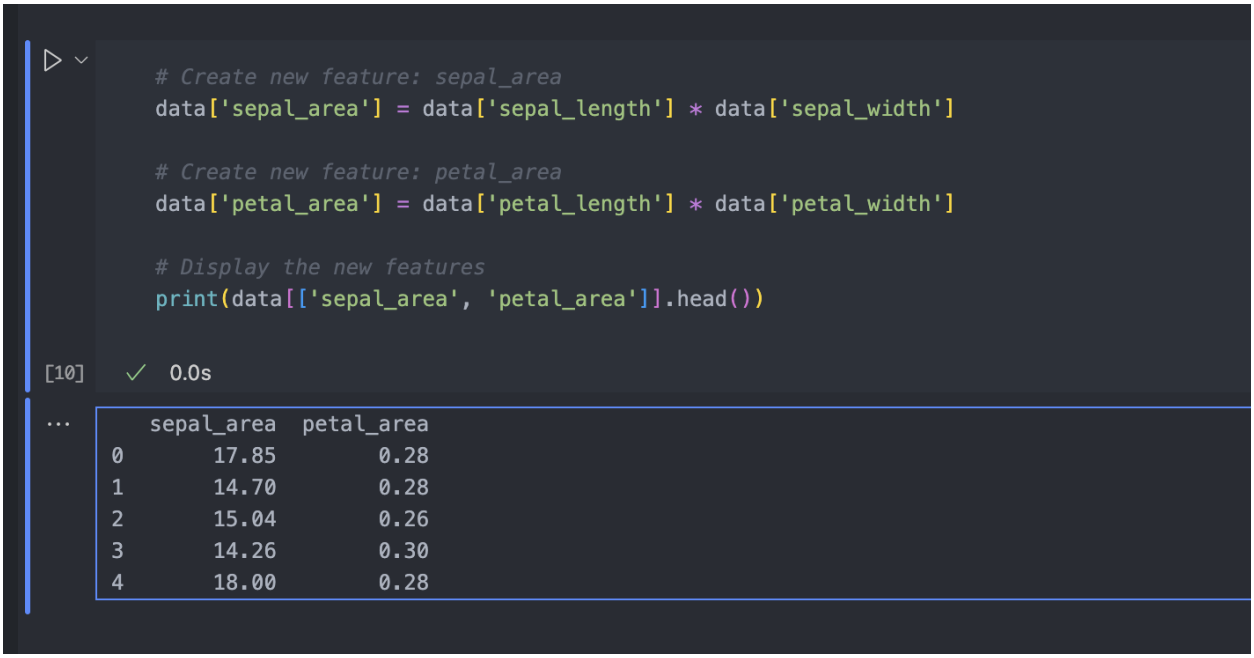
Task 3: Applying data preprocessing techniques such as scaling and feature engineering

Apply appropriate preprocessing techniques, such as scaling numerical features and encoding categorical variables. Additionally, enhance the dataset with new engineered features. Since the Iris dataset doesn't have missing values, you can skip this step.

1. Create new features based on existing ones. For example, create features that represent areas (like sepal area and petal area). Create a new code cell and enter the following code in it.

```
# Create a new feature: sepal_area

data['sepal_area'] = data['sepal_length'] * data['sepal_width']

# Create a new feature: petal_area

data['petal_area'] = data['petal_length'] * data['petal_width']

# Display the new features

print(data[['sepal_area', 'petal_area']].head())
```

Executing the above code cell will result in creating the features sepal_area and petal_area.



2. Scale the numerical features to ensure consistent ranges for model training. To do this, create a new code cell and enter the following code in it.

```
from sklearn.preprocessing import StandardScaler

# Select numerical columns

numerical_cols = ['sepal_length', 'sepal_width', 'petal_length',
'petal_width', 'sepal_area', 'petal_area']

# Apply Standard Scaling

scaler = StandardScaler()

data[numerical_cols] = scaler.fit_transform(data[numerical_cols])

# Display scaled features

print(data[numerical_cols].head())
```

Executing the above code cell will result in the following output.

```
from sklearn.preprocessing import StandardScaler

# Select numerical columns
numerical_cols = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'sepal_area', 'petal_area']

# Apply Standard Scaling
scaler = StandardScaler()
data[numerical_cols] = scaler.fit_transform(data[numerical_cols])

# Display scaled features
print(data[numerical_cols].head())

[11]  ✓ 3.6s

   sepal_length  sepal_width  petal_length  petal_width  sepal_area  \
0     -0.900681     1.019004     -1.340227    -1.315444    0.008098
1     -1.143017    -0.131979     -1.340227    -1.315444   -0.932024
2     -1.385353     0.328414     -1.397064    -1.315444   -0.830551
3     -1.506521     0.098217     -1.283389    -1.315444   -1.063343
4     -1.021849     1.249201     -1.340227    -1.315444    0.052866

   petal_area
0   -1.174041
1   -1.174041
2   -1.178299
3   -1.169783
4   -1.174041
```

Task 4: Building an automated ML pipeline using scikit-learn and setting up GitHub Actions to automate the model training and evaluation pipeline

This task aims to set up a machine learning pipeline that automates data preprocessing and model training. Integrate this pipeline with GitHub Actions to automate the workflow.

1. For defining the ML pipeline, use scikit-learn's Pipeline to chain together the preprocessing and model training steps. To do this, create a new code cell, enter the following code, and execute it.

```
from sklearn.pipeline import Pipeline

from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import train_test_split


# Split data into features and target

X = data.drop('species', axis=1)

y = data['species']


# Split into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)


# Build an ML pipeline with preprocessing and model training steps

pipeline = Pipeline(steps=[

    ('classifier', RandomForestClassifier(n_estimators=100,
random_state=42))

])
```

```
# Train the pipeline

pipeline.fit(X_train, y_train)
```

2. Test the trained pipeline on the test data and evaluate its performance using accuracy and other metrics. Create a new code cell, enter the following code, and execute it.

```
from sklearn.metrics import accuracy_score, classification_report


# Predict on test set

y_pred = pipeline.predict(X_test)


# Evaluate the model

accuracy = accuracy_score(y_test, y_pred)

print(f"Model Accuracy: {accuracy:.2f}")


# Classification report

print(classification_report(y_test, y_pred))
```

Executing the above code will result in providing the results of precision, recall, f1-score, and support.

```
Model Accuracy: 1.00
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

3. Create a file **ml_pipeline.py** and paste all the Python code from the **ipynb** file to it. Set up GitHub Actions to automate the ML pipeline. Create a **.github/workflows/main.yml** file to automate the pipeline on push events and add the following code to it.

```
name: ML Pipeline Automation

on: [push]

jobs:

  build:

    runs-on: ubuntu-latest


    steps:

    - name: Checkout code

      uses: actions/checkout@v2


    - name: Set up Python

      uses: actions/setup-python@v2

      with:

        python-version: 3.9


    - name: Install dependencies

      run: |

        python -m venv venv

        source venv/bin/activate

        pip install -r requirements.txt


    - name: Run pipeline

      run: |
```

```
source venv/bin/activate

python ml_pipeline.py4.
```

4. Commit and push to Github, run the following commands in the terminal.

```
git add .

git commit -m "Added automated ML pipeline and GitHub Actions"

git push origin main
```

5. Verify that your GitHub repository now has the files upload.

> **Note**
>
> Ensure that you logout of your GitHub account on completion of this exercise.

Lab review

1. What is the purpose of the requirements.txt file?
   A. To store project data
   B. To list and manage project dependencies
   C. To configure GitHub Actions workflows
   D. To automate the ML pipeline

> **STOP**
>
> You have successfully completed this lab.

Lab 4: Select, Implement, and Evaluate Algorithms

Lab overview

In this lab, you will:

- Explore the dataset, preprocessing, and feature engineering
- Implement multiple machine learning algorithms
- Evaluate the models using various metrics
- Compare the result to identify the best-performing model

Estimated completion time

30 minutes

> **Note**
>
> You need to install pandas, matplotlib, joblib, scikit-learn and seaborn

Task 1: Loading data and exploratory data analysis (EDA)

This task aims to load the dataset and conduct an exploratory data analysis (EDA) to understand the data's structure, distribution, and key characteristics.

1. Create an empty folder with the name **Lab04** in the lab environment.
2. Open the **Visual Studio Code**, click the **File** menu item, and then click the submenu **Open Folder**. In the browser window, locate the folder that you created in step one and then click **Open**. This will show the folder in the left window of the Visual Studio Code.
3. Now click on the **Terminal** menu item and then click on the sub-menu item, **New Terminal**. This will open a terminal at the bottom to enter commands.
4. Create a new virtual environment for this lab and run the following command in the terminal window to create the virtual environment.

```
virtualenv venv
```

5. Run the following command to activate the virtual environment.

```
.\venv\Scripts\activate
```

6. Now, run the following command to confirm that the virtual environment is activated.

```
python --version
```

7. A new virtual environment setup is completed for this project. To install the required libraries in the venv, run the following command in the terminal window.

```
pip install pandas matplotlib seaborn scikit-learn joblib ipykernel
```

8. Using Windows File Explorer, copy the dataset file **winequality-red.csv** into the Lab04 project folder, from the C:\MLOps\Data-Files folder.
9. Now copy the **Lab04.ipynb** file from the C:\MLOps\Lab-Files\Lab04 folder to the Lab04 project folder. Both files should then be visible in the VS Code Explorer window.
10. Import the necessary libraries for data manipulation, visualization, and machine learning. Click on the **Lab04.ipynb** notebook file to open the code cells. Read, check, and verify that the following code is present in the first code cell, then click the **run** button on the left side of the code cell to execute the code.

```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns
```

```
from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler
```

11. Check the code in the next cell. This will load the dataset and display its first few rows, in order to to understand its structure.

```
# Load the Wine Quality dataset

data = pd.read_csv('winequality-red.csv')

# Display the first few rows of the dataset

print(data.head())

# Display information about the dataset

print(data.info())
```

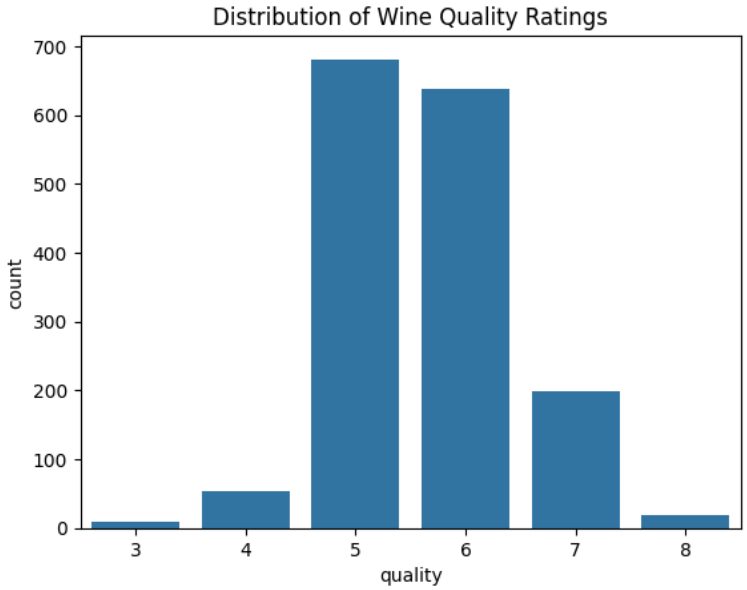
Executing the above code will display the information about the dataset that contains physicochemical tests as features and quality as the target.

```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality
0      9.4        5
1      9.8        5
2      9.8        5
3      9.8        6
4      9.4        5
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
```

12. The next cell code contents uses countplot to visualise the distribution of the target variable quality in order to check for imbalances in the dataset.

```
# Plot the distribution of the target variable

sns.countplot(x='quality', data=data)

plt.title('Distribution of Wine Quality Ratings')

plt.show()
```

Executing the above code cell will display the count plot as shown below.



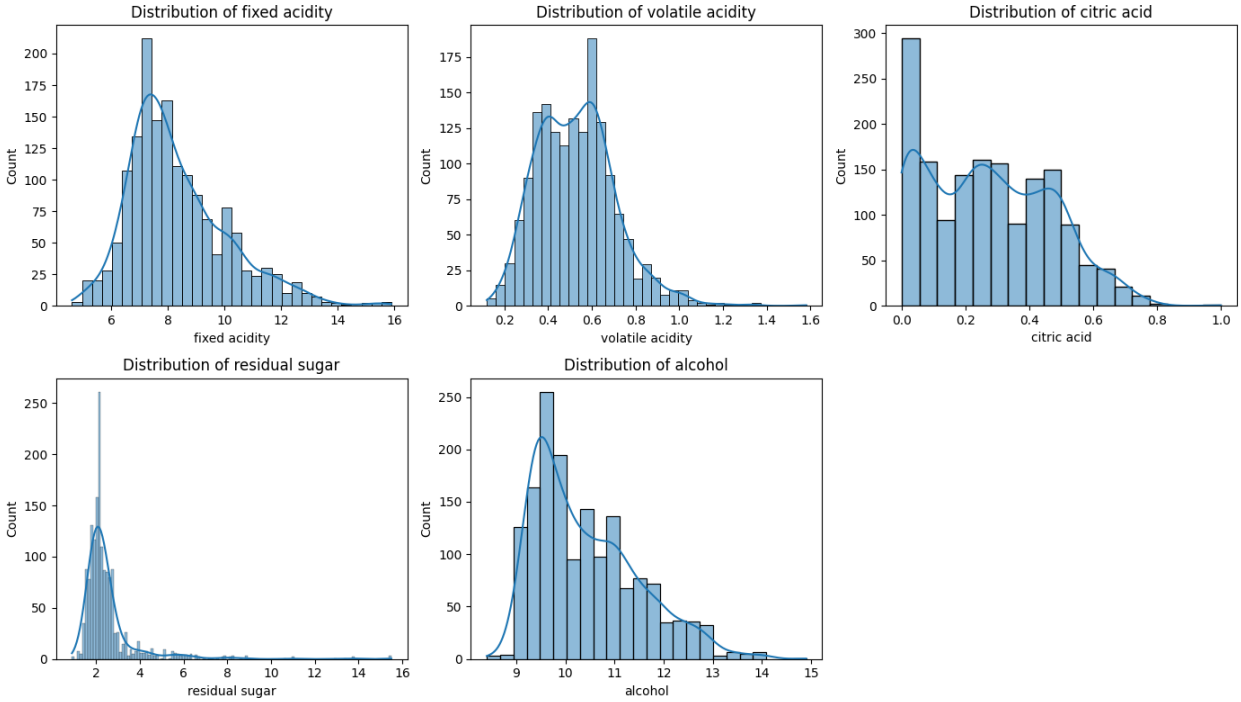
The target variable quality is imbalanced. Most wines have a quality rating of 5 or 6, while fewer wines are rated 3, 4, or 8.

13. To explore the feature distribution, visualize the distribution of some key features to better understand their characteristics. Verify the code in the next cell.

```
# Plot the distribution of important features

features = ['fixed acidity', 'volatile acidity', 'citric acid',
'residual sugar', 'alcohol']

plt.figure(figsize=(14, 8))

for i, feature in enumerate(features, 1):

    plt.subplot(2, 3, i)

    sns.histplot(data[feature], kde=True)

    plt.title(f'Distribution of {feature}')
```

```
plt.tight_layout()

plt.show()
```

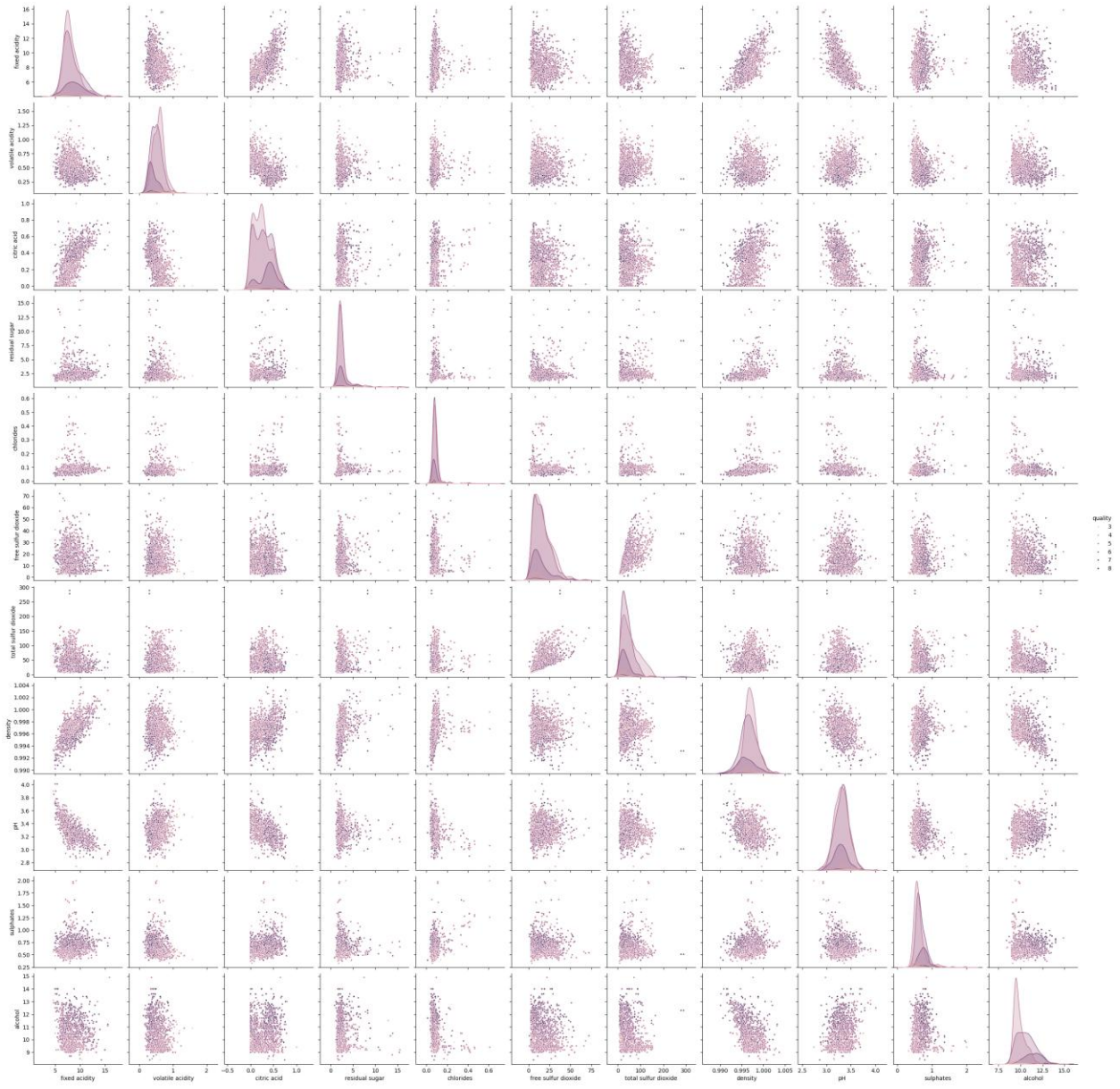
Executing the above code will display the following histogram distributions of various features.



14. For multivariate exploration, use the pairplot, which helps in visualizing the relationships between multiple features and the target variable. Check and run the code in the next cell.

```
# Pairplot to visualize relationships between features and the target

sns.pairplot(data, diag_kind='kde', hue='quality', markers='.')

plt.show()
```

Executing the above code cell will result in the following plots. This shows the potential relationship between features and the target variables, as well as how the distribution overlaps between different quality ratings.



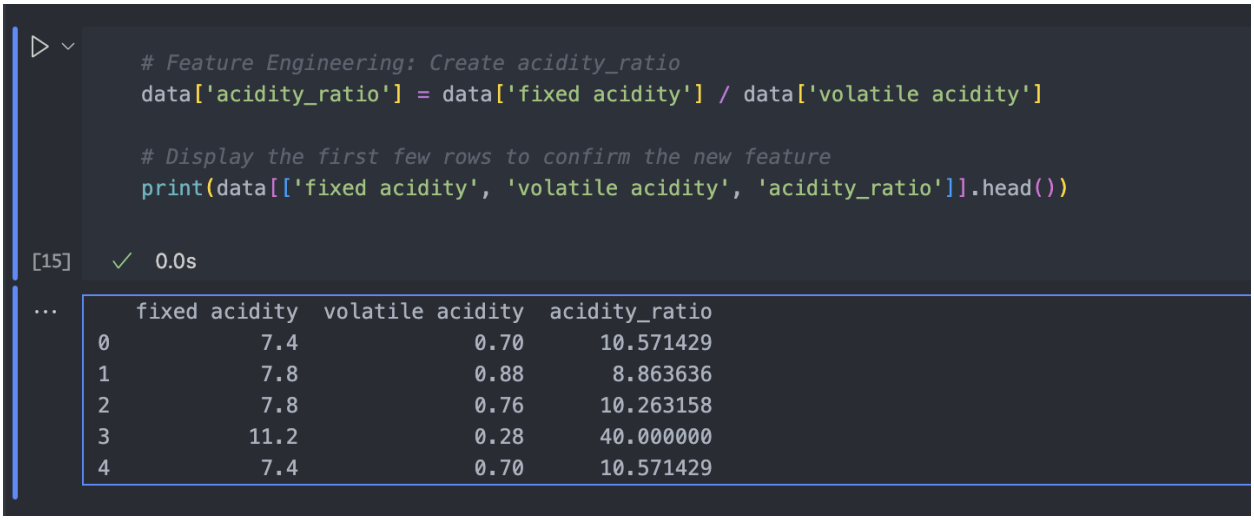
Task 2: Data preprocessing and feature engineering

This task focuses on scaling numeric features, handling imbalanced data, and feature engineering to improve the performance of our models.

1. Create a new feature acidity_ratio to represent the balance between fixed and volatile acidity. Check and verify the following contents if the next code cell.

```
# Feature Engineering: Create acidity_ratio

data['acidity_ratio'] = data['fixed acidity'] / data['volatile
acidity']

# Display the first few rows to confirm the new feature

print(data[['fixed acidity', 'volatile acidity',
'acidity_ratio']].head())
```

Executing the above code will result in the following output.



2. Scale the numeric features using StandardScaler to ensure that all features have consistent ranges, using the code in the next cell.

```
# List of numeric features to scale

numeric_features = ['fixed acidity', 'volatile acidity', 'citric
acid', 'residual sugar',

                    'chlorides', 'free sulfur dioxide', 'total sulfur
dioxide',
```

```
                    'density', 'pH', 'sulphates', 'alcohol',
'acidity_ratio']

# Apply StandardScaler to numeric features

scaler = StandardScaler()

data[numeric_features] = scaler.fit_transform(data[numeric_features])


# Display scaled features

print(data[numeric_features].head())
```

Executing the above code will result in the following output.

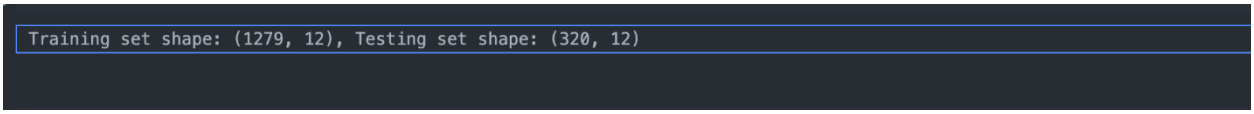
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0      -0.528360          0.961877    -1.391472       -0.453218  -0.243707
1      -0.298547          1.967442    -1.391472        0.043416   0.223875
2      -0.298547          1.297065    -1.186070       -0.169427   0.096353
3       1.654856         -1.384443     1.484154       -0.453218  -0.264960
4      -0.528360          0.961877    -1.391472       -0.453218  -0.243707

   free sulfur dioxide  total sulfur dioxide   density        pH  sulphates  \
0            -0.466193             -0.379133  0.558274  1.288643  -0.579207
1             0.872638              0.624363  0.028261 -0.719933   0.128950
2            -0.083669              0.229047  0.134264 -0.331177  -0.048089
3             0.107592              0.411500  0.664277 -0.979104  -0.461180
4            -0.466193             -0.379133  0.558274  1.288643  -0.579207

    alcohol  acidity_ratio
0 -0.960246      -0.826647
1 -0.584777      -1.010830
2 -0.584777      -0.859894
3 -0.584777       2.347183
4 -0.960246      -0.826647
```

3. Split the data into training and testing sets.

```
# Define features and target variable

X = data.drop('quality', axis=1)

y = data['quality']

# Split the data into train and test sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

print(f"Training set shape:", X_train.shape, f"Testing set shape:",
X_test.shape)
```

After executing the above code cell we will get the X as the training dataset and Y as the testing dataset having the following shapes.



Task 3: Implementing multiple algorithms

In this task, you will implement various machine learning algorithms and evaluate their performance using different metrics.

1. Define functions to evaluate models using common metrics like accuracy, precision, recall, and F1 score.

```
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score, confusion_matrix


# Function to evaluate the models

def evaluate_model(y_true, y_pred):

    accuracy = accuracy_score(y_true, y_pred)

    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', zero_division=0)

    return accuracy, precision, recall, f1


# Function to print the confusion matrix

def print_confusion_matrix(y_true, y_pred):

    cm = confusion_matrix(y_true, y_pred)

    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
```

```
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()
```

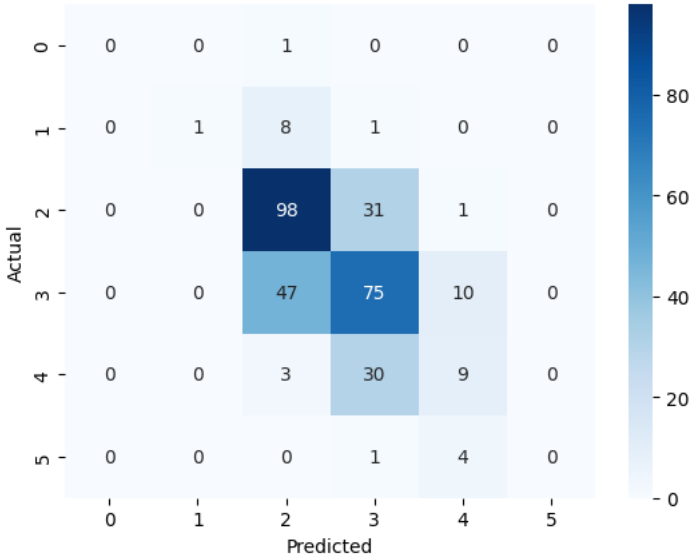
2. Now, start with the first model: Logistic Regression.

```
from sklearn.linear_model import LogisticRegression


# Initialize Logistic Regression
lr_model = LogisticRegression(max_iter=200)


# Train the model
lr_model.fit(X_train, y_train)
# Predict on the test set
lr_predictions = lr_model.predict(X_test)
# Evaluate the Logistic Regression model
lr_accuracy, lr_precision, lr_recall, lr_f1 = evaluate_model(y_test,
lr_predictions)
print(f"Logistic Regression - Accuracy: {lr_accuracy:.2f}, Precision:
{lr_precision:.2f}, Recall: {lr_recall:.2f}, F1 Score: {lr_f1:.2f}")
# Confusion matrix
print_confusion_matrix(y_test, lr_predictions)
```

Executing the above code will train the model on the training dataset, provide the results on different metrics, and display the confusion matrix.

Results: **Accuracy**: 0.57, **Precision**: 0.56, **Recall**: 0.57, **F1 Score**: 0.55

Confusion Matrix:



3. Now, train the second model: **Decision Tree Classifier**.

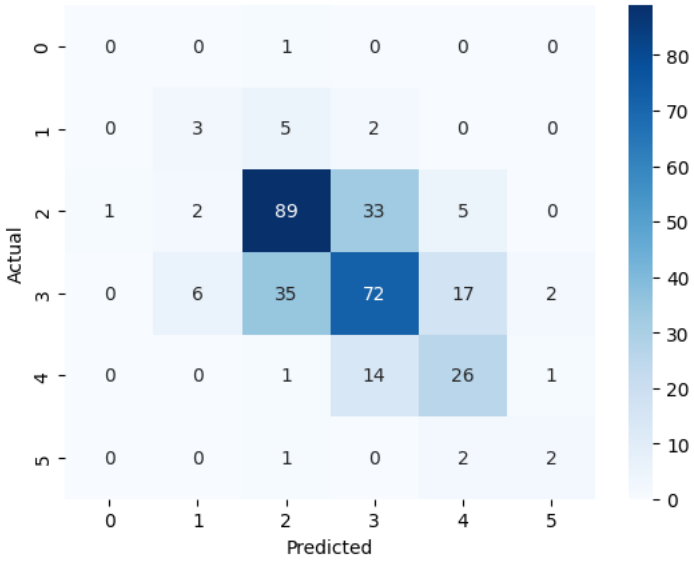
```
from sklearn.tree import DecisionTreeClassifier


# Initialize Decision Tree

dt_model = DecisionTreeClassifier(random_state=42)

# Train the model

dt_model.fit(X_train, y_train)

# Predict on the test set

dt_predictions = dt_model.predict(X_test)

# Evaluate the Decision Tree model

dt_accuracy, dt_precision, dt_recall, dt_f1 = evaluate_model(y_test,
dt_predictions)

print(f"Decision Tree - Accuracy: {dt_accuracy:.2f}, Precision:
{dt_precision:.2f}, Recall: {dt_recall:.2f}, F1 Score: {dt_f1:.2f}")
```

```
# Confusion matrix

print_confusion_matrix(y_test, dt_predictions)
```

Executing the above code will train the model on the training dataset, provide the results on different metrics, and display the confusion matrix.

Results: Accuracy: 0.60, Precision: 0.60, Recall: 0.60, F1 Score: 0.60

Confusion Matrix:



4. Train the third model: Random Forest.

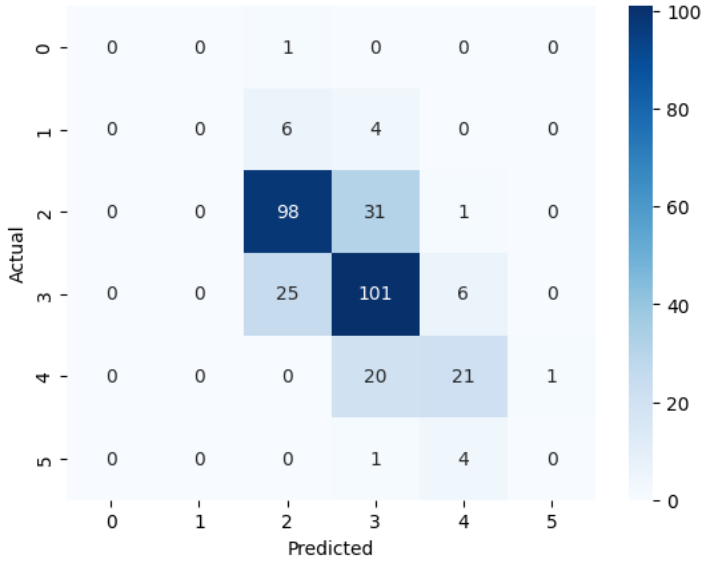
```
from sklearn.ensemble import RandomForestClassifier

# Initialize Random Forest

rf_model = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the model

rf_model.fit(X_train, y_train)

# Predict on the test set

rf_predictions = rf_model.predict(X_test)

# Evaluate the Random Forest model

rf_accuracy, rf_precision, rf_recall, rf_f1 = evaluate_model(y_test,
rf_predictions)

print(f"Random Forest - Accuracy: {rf_accuracy:.2f}, Precision:
{rf_precision:.2f}, Recall: {rf_recall:.2f}, F1 Score: {rf_f1:.2f}")
```

```
# Confusion matrix

print_confusion_matrix(y_test, rf_predictions)
```

Executing the above code will train the model on the training dataset, provide the results on different metrics, and display the confusion matrix.

Results: Accuracy: 0.69, Precision: 0.66, Recall: 0.69, F1 Score: 0.67

Confusion Matrix:



5. Train the fourth model: Naive Bayes.

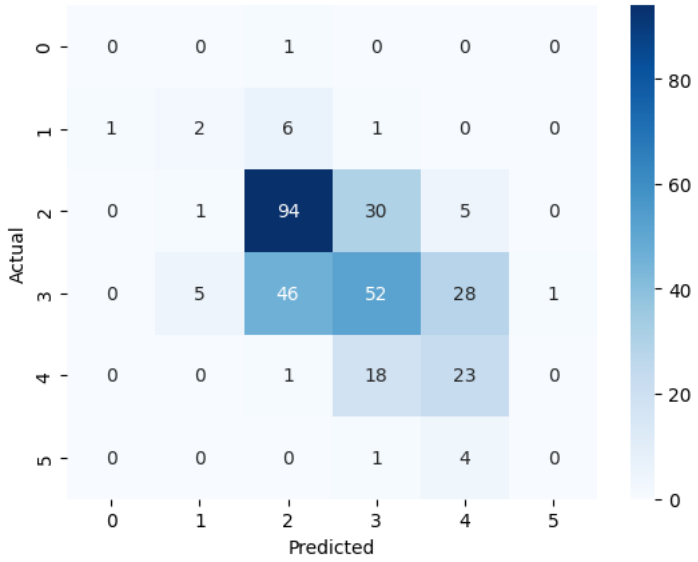
```
from sklearn.naive_bayes import GaussianNB

# Initialize Naive Bayes

nb_model = GaussianNB()

# Train the model

nb_model.fit(X_train, y_train)

# Predict on the test set

nb_predictions = nb_model.predict(X_test)

# Evaluate the Naive Bayes model

nb_accuracy, nb_precision, nb_recall, nb_f1 = evaluate_model(y_test,
nb_predictions)

print(f"Naive Bayes - Accuracy: {nb_accuracy:.2f}, Precision:
{nb_precision:.2f}, Recall: {nb_recall:.2f}, F1 Score: {nb_f1:.2f}")
```

```
# Confusion matrix

print_confusion_matrix(y_test, nb_predictions)
```

Executing the above code will train the model on the training dataset, provide the results on different metrics, and display the confusion matrix.

Results: Accuracy: 0.53, Precision: 0.53, Recall: 0.53, F1 Score: 0.52

Confusion Matrix:



6. Train the fifth model: Support Vector Machine (SVM).

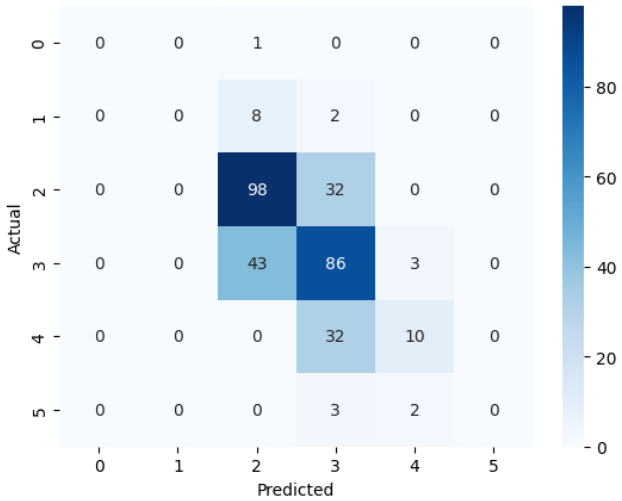
```
from sklearn.svm import SVC

# Initialize SVM

svm_model = SVC(kernel='rbf', random_state=42)

# Train the model

svm_model.fit(X_train, y_train)

# Predict on the test set

svm_predictions = svm_model.predict(X_test)

# Evaluate the SVM model

svm_accuracy, svm_precision, svm_recall, svm_f1 =
evaluate_model(y_test, svm_predictions)
```

```
print(f"SVM - Accuracy: {svm_accuracy:.2f}, Precision:
{svm_precision:.2f}, Recall: {svm_recall:.2f}, F1 Score:
{svm_f1:.2f}")

# Confusion matrix

print_confusion_matrix(y_test, svm_predictions)
```

Executing the above code will train the model on the training dataset, provide the results on different metrics, and display the confusion matrix.

Results: Accuracy: 0.61, Precision: 0.58, Recall: 0.61, F1 Score: 0.58

Confusion Matrix:



7. Train the sixth model: K-Nearest Neighbors (KNN).

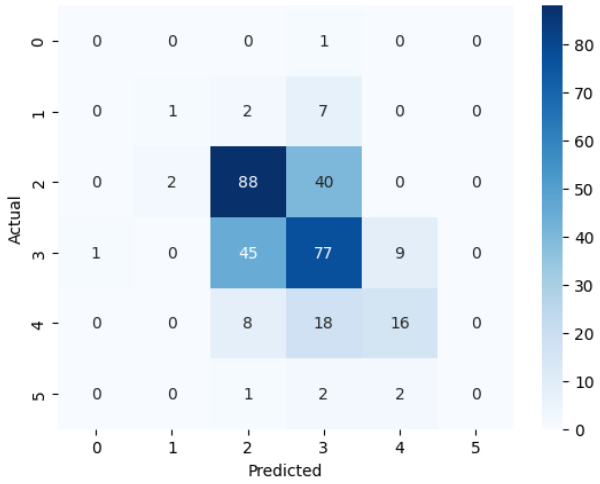
```
from sklearn.neighbors import KNeighborsClassifier

# Initialize KNN

knn_model = KNeighborsClassifier(n_neighbors=5)

# Train the model

knn_model.fit(X_train, y_train)

# Predict on the test set

knn_predictions = knn_model.predict(X_test)

# Evaluate the KNN model

knn_accuracy, knn_precision, knn_recall, knn_f1 =
evaluate_model(y_test, knn_predictions)
```

```
print(f"KNN - Accuracy: {knn_accuracy:.2f}, Precision:
{knn_precision:.2f}, Recall: {knn_recall:.2f}, F1 Score:
{knn_f1:.2f}")

# Confusion matrix

print_confusion_matrix(y_test, knn_predictions)
```

Executing the above code will train the model on the training dataset, provide the results on different metrics, and display the confusion matrix.

Results: Accuracy: 0.57, Precision: 0.56, Recall: 0.57, F1 Score: 0.56

Confusion Matrix:



8. Train the seventh model: Gradient Boosting.

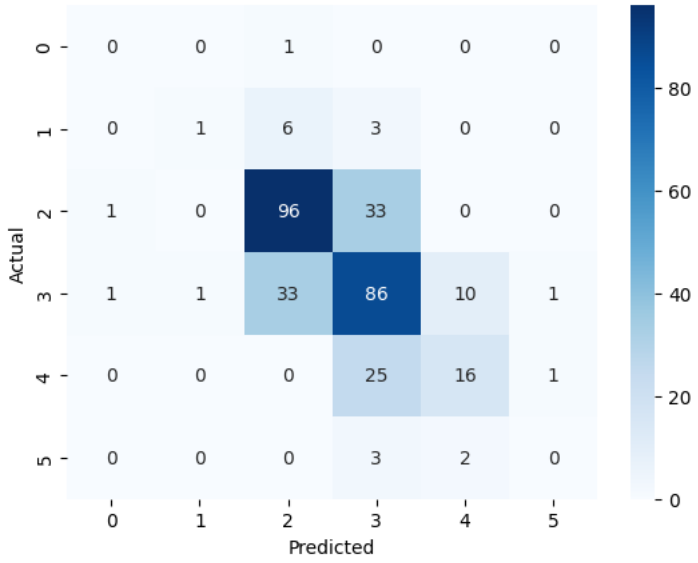
```
from sklearn.ensemble import GradientBoostingClassifier

# Initialize Gradient Boosting

gb_model = GradientBoostingClassifier(random_state=42)

# Train the model

gb_model.fit(X_train, y_train)

# Predict on the test set

gb_predictions = gb_model.predict(X_test)

# Evaluate the Gradient Boosting model

gb_accuracy, gb_precision, gb_recall, gb_f1 = evaluate_model(y_test,
gb_predictions)
```

```
print(f"Gradient Boosting - Accuracy: {gb_accuracy:.2f}, Precision:
{gb_precision:.2f}, Recall: {gb_recall:.2f}, F1 Score: {gb_f1:.2f}")

# Confusion matrix

print_confusion_matrix(y_test, gb_predictions)
```

Executing the above code will train the model on the training dataset, provide the results on different metrics, and display the confusion matrix.

Results: Accuracy: 0.62, Precision: 0.61, Recall: 0.62, F1 Score: 0.61

Confusion Matrix:



Task 4: Comparing evaluation metrics

This task aims to compare the performance of all the models you have implemented based on their evaluation metrics (accuracy, precision, recall, F1 score). By summarizing the performance of each model, you will identify the best-performing model and determine which one is most suitable for the given problem.

In this task, you will also understand how to choose a model based on specific metrics, considering the nature of the dataset and the problem.

When comparing models, it's essential to evaluate multiple performance metrics rather than relying on a single metric like accuracy. Here's why each metric is important.

- Accuracy

  Measures the overall correctness of the model's predictions.

  However, in imbalanced datasets (like this one), accuracy might be misleading as it can be high even if the model performs poorly on the minority classes.

- Precision

  Measures how many of the model's positive predictions are correct.

  This is particularly important when false positives are costly (e.g., misclassifying a low-quality wine as high-quality).

- Recall

  Measures how many actual positives were correctly identified by the model.

  This is critical when missing out on true positives is more costly (e.g., failing to identify a good-quality wine).

- F1 Score

  The harmonic mean of precision and recall. This is often the preferred metric when there's a class imbalance, as it balances the trade-off between precision and recall.

How to select the best model

To select the best model, consider the following.

- Balanced metric

  If your dataset is imbalanced (as in this case), accuracy might not reflect the model's true performance. In this case, the F1 score becomes more important as it accounts for both false positives and false negatives.

- Precision vs. recall

  If the task prioritizes minimizing false positives (e.g., in medical diagnoses where a false alarm could be costly), you should focus on precision.

  If the task prioritizes minimizing false negatives (e.g., in fraud detection, where missing a fraud case is more harmful), recall is more important.

- Domain context

  For this wine quality dataset, the objective might be to ensure that both high-quality and low-quality wines are correctly classified. Therefore, a model with a high F1 score will likely be a good choice.

- Computational complexity

  Models like Random Forest and XGBoost generally provide strong performance, but they are computationally expensive. If the goal is to deploy a model that balances speed and

performance, Logistic Regression or Decision Tree might be more suitable for real-time systems, even if they slightly compromise accuracy.

1. Verify and then run the code in the final cell, that summarizes the performance of all models.

```
# Compare model performance

results = {

    'Model': ['Logistic Regression', 'Decision Tree', 'Random Forest',
'Naive Bayes',

              'SVM', 'KNN', 'Gradient Boosting'],

    'Accuracy': [lr_accuracy, dt_accuracy, rf_accuracy, nb_accuracy,
svm_accuracy, knn_accuracy, gb_accuracy],

    'Precision': [lr_precision, dt_precision, rf_precision,
nb_precision, svm_precision, knn_precision, gb_precision],

    'Recall': [lr_recall, dt_recall, rf_recall, nb_recall, svm_recall,
knn_recall, gb_recall],

    'F1 Score': [lr_f1, dt_f1, rf_f1, nb_f1, svm_f1, knn_f1, gb_f1]

}


results_df = pd.DataFrame(results)

print(results_df)
```

You can get the performance of all the models by executing the above code. Following is the output.

```
                Model  Accuracy  Precision    Recall  F1 Score
0  Logistic Regression  0.571875   0.558236  0.571875  0.548083
1        Decision Tree  0.600000   0.602388  0.600000  0.600148
2        Random Forest  0.687500   0.657749  0.687500  0.669065
3          Naive Bayes  0.534375   0.526443  0.534375  0.524199
4                  SVM  0.606250   0.581788  0.606250  0.577640
5                  KNN  0.568750   0.555510  0.568750  0.555958
6    Gradient Boosting  0.621875   0.613890  0.621875  0.610037
```

Lab review

1. What does the confusion matrix visualize in the evaluation step?
   A. Correlation between features
   B. True positives, false positives, true negatives, and false negatives
   C. Accuracy of the model predictions
   D. Distribution of numerical features

STOP

You have successfully completed this lab.

Lab 5: Experiment Tracking and Model Management with MLflow

Lab overview

In this lab, you will:

- Set up MLflow for tracking experiments
- Track metrics and parameters for different models
- Compare the results of multiple model runs and visualize them in MLflow's UI
- Save and retrieve models for deployment

Estimated completion time

30 minutes

Task 1: Setting up MLflow for experiment tracking

This task aims to configure MLflow in your project so that you can log and track machine learning experiments.

1. Create an empty folder with the name **Lab05** in the lab environment.
2. Open the **Visual Studio Code**, click the **File** menu item, and then click the submenu **Open Folder**. In the browser window, locate the folder **Lab05** and then click **Open**. This will show the folder in the left window of the Visual Studio Code.
3. Click on the **Terminal** menu item and then click on the sub-menu item, **New Terminal**. This will open a terminal at the bottom to enter commands.
4. Using Windows File Explorer, copy the dataset file **winequality-red.csv** into the Lab05 project folder, from the C:\MLOps\Data-Files folder.
5. Now copy the **Lab05.ipynb** file from the C:\MLOps\Lab-Files\Lab05 folder to the Lab05 project folder. Both files should then be visible in the VS Code Explorer window.
6. Now, create a new virtual environment for this lab and run the following command in the terminal window to create the virtual environment.

```
virtualenv venv
```

7. Run the following command to activate the virtual environment.

```
.\venv\Scripts\activate
```

8. Now, run the following command to confirm that the virtual environment is activated.

```
python --version
```

9. A new virtual environment setup is completed for this project. To install the required libraries in the venv, run the following command in the terminal window.

```
pip install mlflow scikit-learn pandas matplotlib ipykernel
```

10. Import the necessary data manipulation, visualization, and machine learning libraries. Click the **Lab05.ipynb** file to open the notebook file. Ensure that the following code is displayed in the first cell and then run the code.

```
# Import necessary Libraries

import pandas as pd

import mlflow

import mlflow.sklearn
```

```
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, precision_score, f1_score

from mlflow.models.signature import infer_signature
```

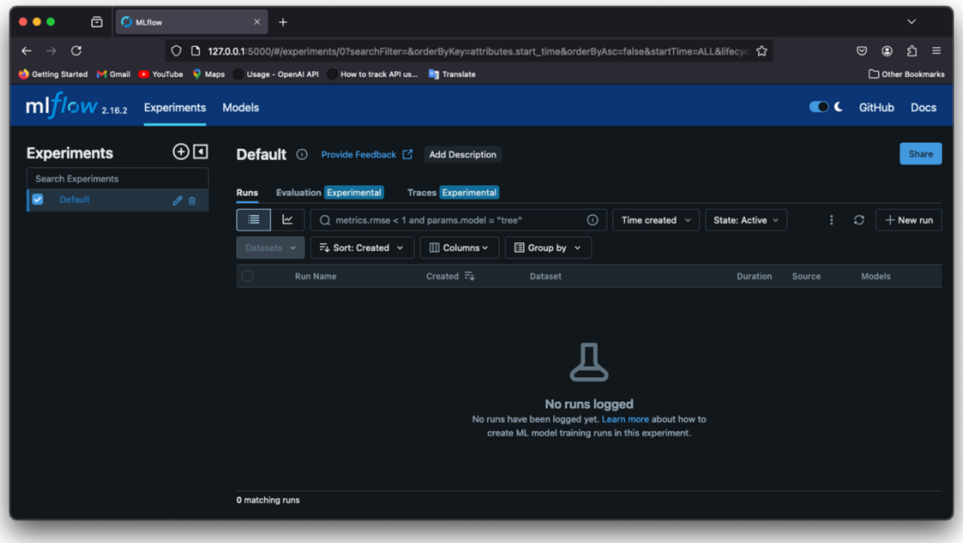
11. The next step is to load the wine quality dataset with the following code.

```
# Load the Wine Quality dataset

df = pd.read_csv("winequality-red.csv")

# Define features and target

X = df.drop('quality', axis=1)

y = df['quality']

# Split the dataset

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Set the experiment name

mlflow.set_experiment("Wine Quality Experiment-MLFlow22")
```

12. To visualize the tracking, run MLflow UI from the command line.

```
mlflow ui
```

This will start the MLflow UI on http://localhost:5000, where you can visualize the experiment logs.



Task 2: Tracking and logging experiment parameters and metrics

In this task, you will track the performance of multiple models and their respective parameters, to decide which model performs best for the given dataset.

1. Choose the two best-performing models from demo-4, Random Forest and Gradient Boosting, and use the same wine quality dataset to split it into training and testing sets. Train the two models, Random Forest and Gradient Boosting, and log the experiment details using MLflow.

```
# Ensure MLflow is tracking with proper logs

def train_and_log_model(model, model_name):

    # End any active run

    if mlflow.active_run():

        mlflow.end_run()


    with mlflow.start_run(run_name=model_name):

        # Log model parameters
```

```
        mlflow.log_params(model.get_params())


        # Train the model
        model.fit(X_train, y_train)


        # Predict and evaluate
        y_pred = model.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred,
average='weighted', zero_division=1)
        f1 = f1_score(y_test, y_pred, average='weighted')


        # Log metrics
        mlflow.log_metrics({
            "accuracy": accuracy,
            "precision": precision,
            "f1_score": f1
        })


        # Define an example input for model signature
        input_example = X_test.iloc[:1]
        signature = infer_signature(X_test, y_pred)


        # Log the model with signature and input example
        mlflow.sklearn.log_model(
            sk_model=model,
            artifact_path="model",
```

```
            signature=signature,
            input_example=input_example
        )

        # Console log for confirmation
        print(f"Logged {model_name} model with accuracy:
{accuracy:.4f}, precision: {precision:.4f}, f1_score: {f1:.4f}")

# Train and log the Random Forest and Gradient Boosting models
rf_model = RandomForestClassifier(n_estimators=100, max_depth=10)
train_and_log_model(rf_model, "Random Forest")

gb_model = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.1)
train_and_log_model(gb_model, "Gradient Boosting")
```

Executing the above code will display the results on the console and also update the MLflow UI.

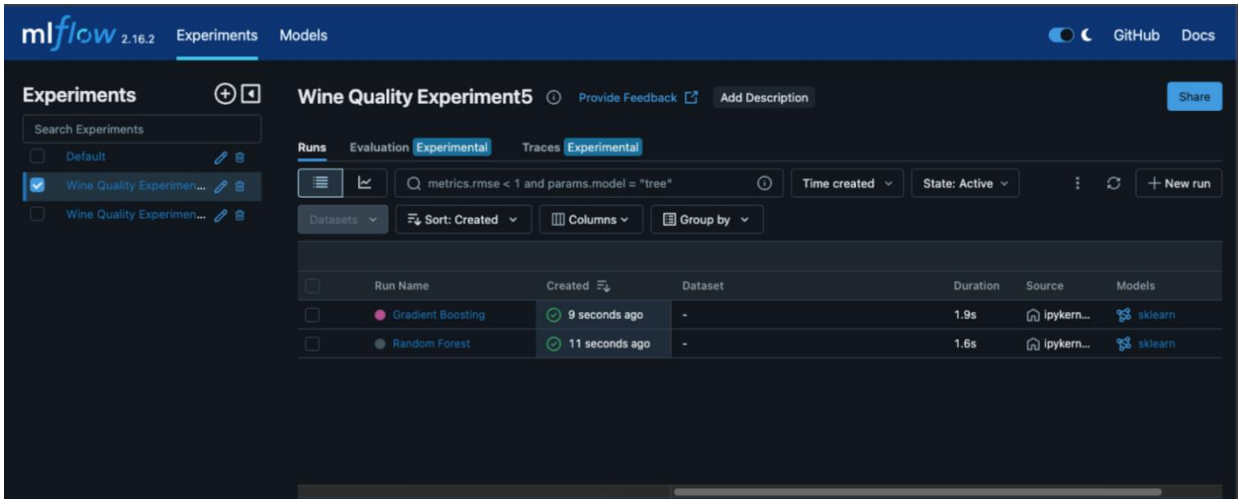
Task 3: Visualizing model performance in MLflow UI

Use MLflow's UI to compare the performance of the models you've trained. This will help you decide which model performs better based on metrics like accuracy, precision, and F1-score.

1. Use MLflow's UI to compare metrics of multiple runs. Open MLflow's UI by running the following command.

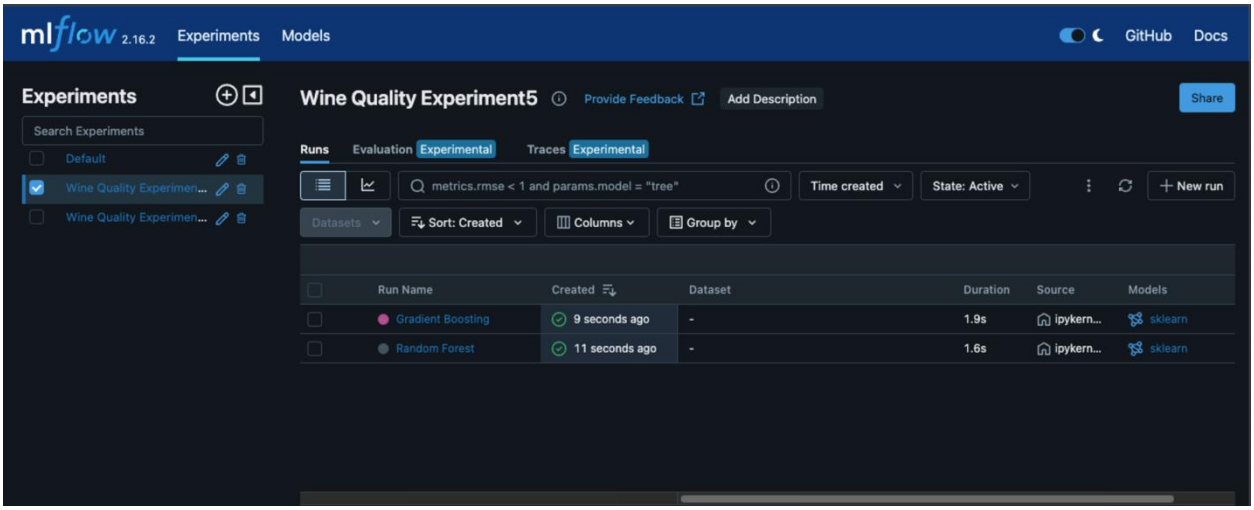
```
mlflow ui
```

Refresh the browser tab if the MLflow UI is already running or navigate to http://localhost:5000. You will be able to compare each model's metrics side by side and see which model performs better.

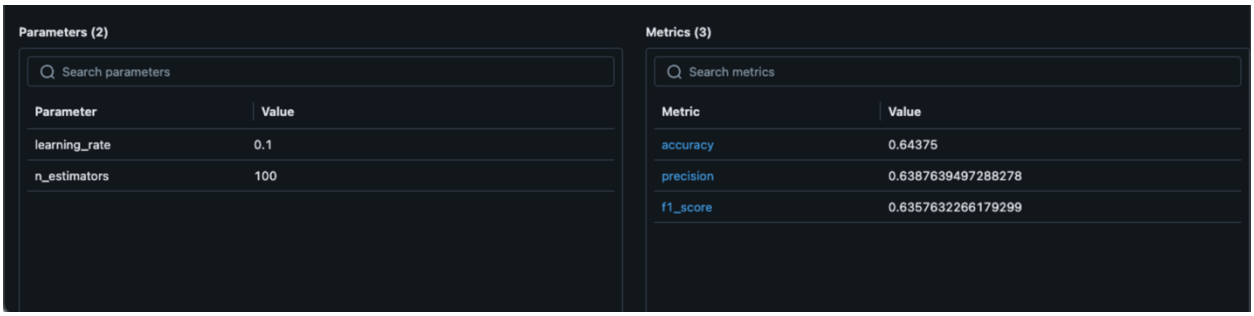


Plot graphs in MLflow's UI for visual comparison. MLflow UI allows you to generate comparison graphs automatically. You can visualize the accuracy and F1 scores for each run directly.

2. Click on the Gradient Boosting model, it will display all the information and results.



3. Scroll down to see the parameters and metrics.



4. Click on the **Model metrics** tab to see the plots on different metrics.

5. Explore all the options available on the MLflow UI to see the information.

In this demo, you learned how to set up MLflow for experiment tracking, log model parameters and metrics, and compare models using the MLflow UI.

MLflow is an excellent tool for managing the full lifecycle of machine learning models, from training to deployment. This approach allows for easy comparison of models, helping you choose the one that best meets your performance criteria. The Random Forest and Gradient Boosting models were tracked and compared in this lab using MLflow's capabilities.

Lab review

1. Which command is used to start the MLflow UI?
   A. mlflow start
   B. mlflow run
   C. mlflow ui
   D. mlflow track

STOP

You have successfully completed this lab.

Lab 6: Model Performance Evaluation and Comparison

Lab overview

In this lab, you will:

- Set up an environment for comprehensive model evaluation and comparison
- Implement a range of evaluation metrics
- Visualize model performance with plots and charts
- Interpret results to analyze trade-offs between models for better decision-making

Estimated completion time

25 minutes

Task 1: Setting up the environment and loading the data

This task aims to set up the working environment, import necessary libraries, and load the dataset into the workspace. This step lays the foundation for the entire lab.

1. Create an empty folder with the name **Lab06** on the desktop of the lab environment.
2. Using Windows File Explorer, copy the dataset file **winequality-red.csv** into the Lab06 project folder, from the C:\MLOps\Data-Files folder.
3. Now copy the **Lab06.ipynb** file from the C:\MLOps\Lab-Files\Lab06 folder to the Lab06 project folder. Both files should then be visible in the VS Code Explorer window.
4. Open the **Visual Studio Code**, click the **File** menu item, and then click the submenu **Open Folder**. In the browser window, locate the folder **Lab06** and then click **Open**. This will show the folder in the left window of the Visual Studio Code.
5. Now click on the **Terminal** menu item and then click on the sub-menu item, **New Terminal**. This will open a terminal at the bottom to enter commands.
6. Now, create a new virtual environment for this lab and run the following command in the terminal window to create the virtual environment.

```
virtualenv venv
```

7. Run the following command to activate the virtual environment.

```
.\venv\Scripts\activate
```

8. Now, run the following command to confirm that the virtual environment is activated.

```
python --version
```

9. A new virtual environment setup is completed for this project, to install the required libraries in the venv, run the following command in the terminal window.

```
pip install mlflow scikit-learn pandas matplotlib seaborn ipykernel
```

10. Import the necessary data manipulation, visualization, and machine learning libraries. Click the **Lab06.ipynb** file to open notebook file. Ensure that the following code is displayed in the first cell, then run the code.

```
# Import necessary libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt
```

```
import seaborn as sns


# Import scikit-learn libraries for model building and evaluation

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier

from sklearn.metrics import (

    accuracy_score, precision_score, recall_score, f1_score,

    roc_auc_score, roc_curve, confusion_matrix, precision_recall_curve

)
```

11. Load the Wine Quality dataset and explore its structure to understand the data. Review and run the next cell contents.

```
# Load the dataset

df = pd.read_csv('winequality-red.csv')


# Explore the first few rows

print(df.head())

print(df.info())
```

12. Preprocess and split data into training and testing sets in this step. Verify the contents and run the following code cell.

```
# Check for missing values

print(df.isnull().sum())

# Feature Engineering (Creating a binary classification for wine
quality)

df['quality_label'] = df['quality'].apply(lambda x: 1 if x >= 6 else
0)

# Drop original 'quality' column

df.drop('quality', axis=1, inplace=True)
```

```
# Preview the dataset

print(df.head())
```

You will handle missing values and create a new binary classification column (quality_label) based on wine quality scores to make the task a binary classification problem. This simplifies model evaluation later.

13. Split the data into training and test sets to evaluate model performance on unseen data.

```
# Define features (X) and target (y)

X = df.drop('quality_label', axis=1)

y = df['quality_label']

# Split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

Task 2: Training models and evaluate performance

The purpose of this task is to build two machine learning models (Random Forest and Gradient Boosting), evaluate their performance using multiple metrics, and log the metrics for comparison.

1. First, train a Random Forest Classifier, which is robust and handles both overfitting and underfitting issues well.

```
# Train Random Forest Classifier

rf_model = RandomForestClassifier(n_estimators=100, max_depth=10,
random_state=42)

rf_model.fit(X_train, y_train)

# Predict on test data

y_pred_rf = rf_model.predict(X_test)
```

   Random Forest is chosen for its strong performance on tabular datasets. You will fit the model on the training data and predict the test data to assess its generalization.

2. Next, train a Gradient Boosting model to compare it with the Random Forest model.

```
# Train Gradient Boosting Classifier

gb_model = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.1, random_state=42)

gb_model.fit(X_train, y_train)

# Predict on test data

y_pred_gb = gb_model.predict(X_test)
```

Gradient Boosting is another powerful model that often outperforms Random Forest for certain types of datasets. You will compare the two models to see which one performs better.

3. Evaluate both models using several key metrics: accuracy, precision, recall, F1-score, confusion matrix, ROC, and AUC. Review and run the code cell contents.

```
# Evaluate Random Forest

rf_accuracy = accuracy_score(y_test, y_pred_rf)

rf_precision = precision_score(y_test, y_pred_rf)

rf_recall = recall_score(y_test, y_pred_rf)

rf_f1 = f1_score(y_test, y_pred_rf)


# Evaluate Gradient Boosting

gb_accuracy = accuracy_score(y_test, y_pred_gb)

gb_precision = precision_score(y_test, y_pred_gb)

gb_recall = recall_score(y_test, y_pred_gb)

gb_f1 = f1_score(y_test, y_pred_gb)


# Print results for both models

print(f"Random Forest - Accuracy: {rf_accuracy:.4f}, Precision:
{rf_precision:.4f}, Recall: {rf_recall:.4f}, F1: {rf_f1:.4f}")

print(f"Gradient Boosting - Accuracy: {gb_accuracy:.4f}, Precision:
{gb_precision:.4f}, Recall: {gb_recall:.4f}, F1: {gb_f1:.4f}")
```

Evaluating models using various metrics helps you understand their performance from multiple perspectives (e.g., precision for false positives, recall for false negatives). This provides a well-rounded view of model effectiveness.

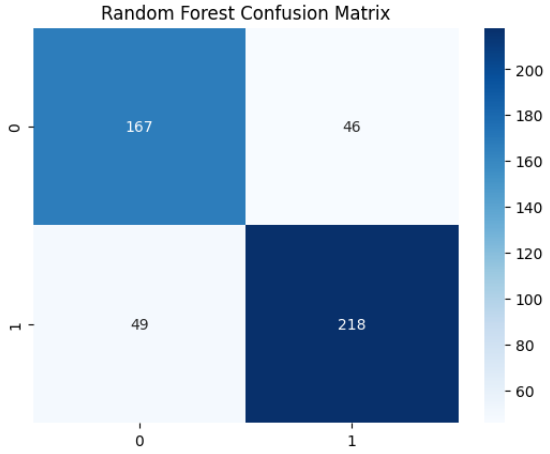
Task 3: Visualizing model performance

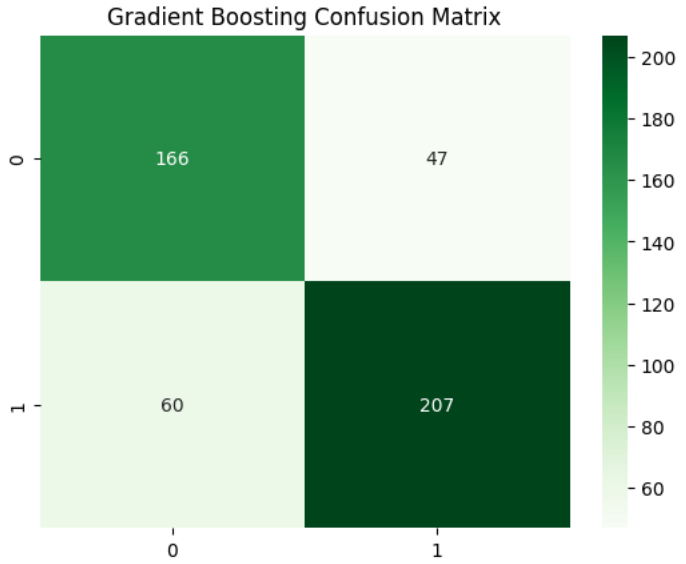
The objective here is to use multiple visualizations (confusion matrix, ROC curve, precision-recall curve) to analyze the model results and highlight performance differences between the models.

1. Visualize the confusion matrix for both models using heatmaps.

```
# Confusion Matrix for Random Forest
rf_cm = confusion_matrix(y_test, y_pred_rf)
sns.heatmap(rf_cm, annot=True, fmt='d', cmap='Blues')
plt.title('Random Forest Confusion Matrix')
plt.show()

# Confusion Matrix for Gradient Boosting
gb_cm = confusion_matrix(y_test, y_pred_gb)
sns.heatmap(gb_cm, annot=True, fmt='d', cmap='Greens')
plt.title('Gradient Boosting Confusion Matrix')
plt.show()
```

Executing the above code cell will generate the following output.





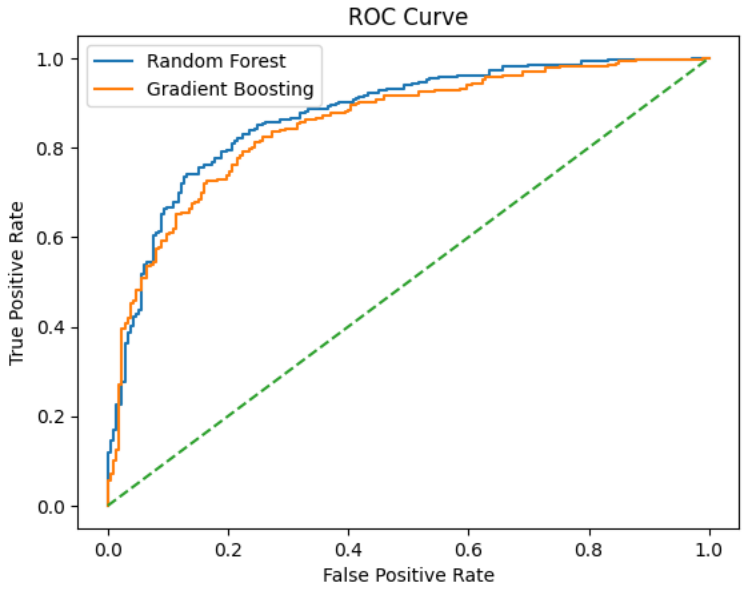
Confusion matrices provide detailed insight into true positives, false positives, and false negatives, helping to evaluate where the model is making mistakes.

2. Compare the models' ability to discriminate between classes using ROC and precision-recall curves.

```
# ROC Curve for Random Forest
rf_prob = rf_model.predict_proba(X_test)[:, 1]
rf_fpr, rf_tpr, _ = roc_curve(y_test, rf_prob)
plt.plot(rf_fpr, rf_tpr, label='Random Forest')

# ROC Curve for Gradient Boosting
gb_prob = gb_model.predict_proba(X_test)[:, 1]
gb_fpr, gb_tpr, _ = roc_curve(y_test, gb_prob)
plt.plot(gb_fpr, gb_tpr, label='Gradient Boosting')

# Plot ROC Curves
plt.plot([0, 1], [0, 1], linestyle='--')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
```

```
plt.ylabel('True Positive Rate')

plt.legend()

plt.show()
```

Executing the above code cell will show the following graph.



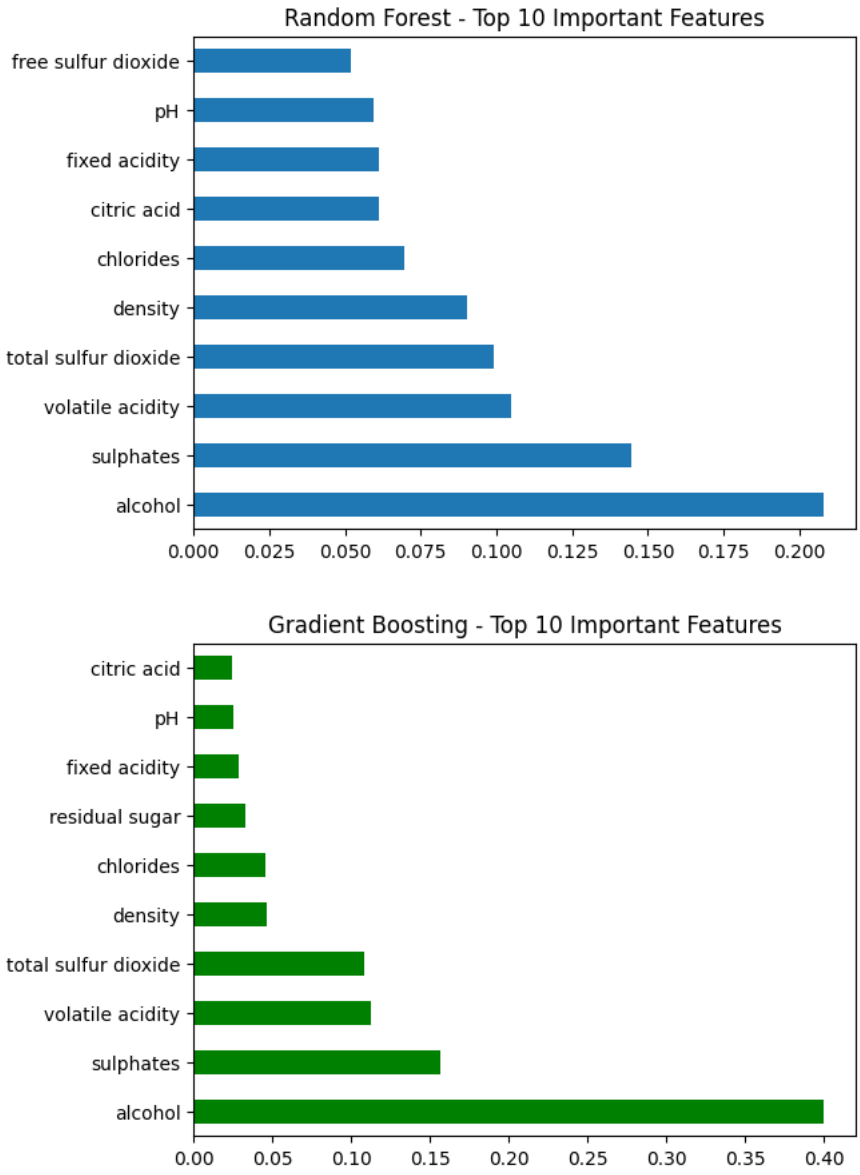
ROC curves illustrate the trade-off between true positive and false positive rates, while precision-recall curves focus on precision in imbalanced datasets. This step is critical for a deeper understanding of model performance.

3. Models like Random Forest and Gradient Boosting feature importance plots help to understand which features contribute the most to the predictions.

```
# Feature importance for Random Forest

rf_feature_importance = pd.Series(rf_model.feature_importances_,
index=X.columns)

rf_feature_importance.nlargest(10).plot(kind='barh', title='Random
Forest - Top 10 Important Features')

plt.show()


# Feature importance for Gradient Boosting

gb_feature_importance = pd.Series(gb_model.feature_importances_,
index=X.columns)
```

```
gb_feature_importance.nlargest(10).plot(kind='barh', title='Gradient
Boosting - Top 10 Important Features', color='green')

plt.show()
```

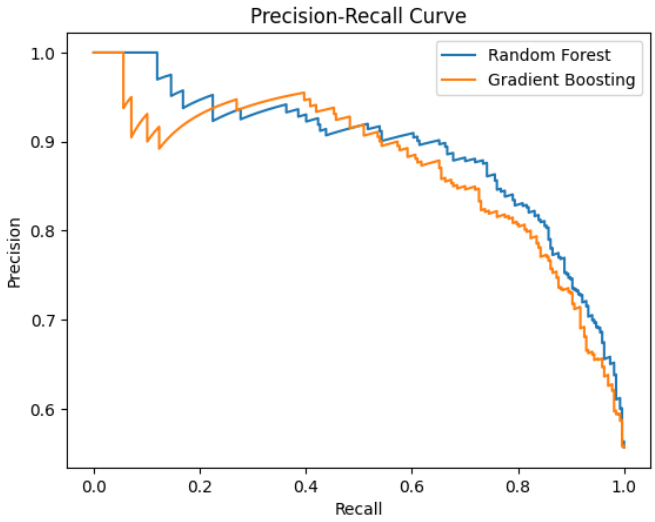
Executing the above code cell will display the following graphs.



This bar plot ranks the features based on their importance in the model. It provides a clear view of which features contribute the most to the prediction, helping to understand the decision-making process of the model.

4. The precision-recall curve is particularly useful for imbalanced datasets, where precision and recall are more informative than accuracy.

```
# Precision-Recall Curve for Random Forest
rf_prob = rf_model.predict_proba(X_test)[:, 1]
rf_precision, rf_recall, _ = precision_recall_curve(y_test, rf_prob)
plt.plot(rf_recall, rf_precision, label='Random Forest')

# Precision-Recall Curve for Gradient Boosting
gb_prob = gb_model.predict_proba(X_test)[:, 1]
gb_precision, gb_recall, _ = precision_recall_curve(y_test, gb_prob)
plt.plot(gb_recall, gb_precision, label='Gradient Boosting')

# Plot Precision-Recall Curves
plt.title('Precision-Recall Curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```



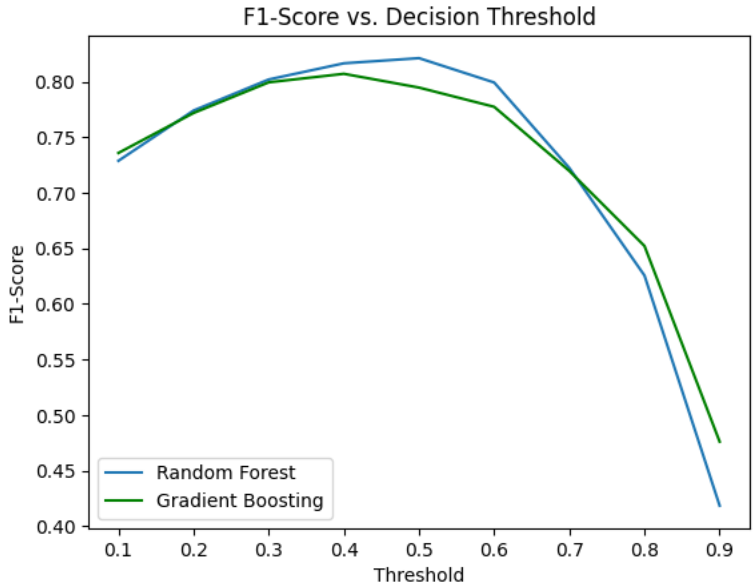
The precision-recall curve visualizes the trade-off between precision and recall. This plot is particularly useful when the dataset is imbalanced, as it focuses on how well the model handles false positives and false negatives.

5. Plot the F1 score vs threshold. This plot shows how the F1-score varies as the decision threshold changes, helping to understand where the best balance between precision and recall lies.

```
# F1-Score vs. Threshold for Random Forest

thresholds = np.arange(0.1, 1, 0.1)

f1_scores_rf = []

f1_scores_gb = []


for threshold in thresholds:

    rf_pred_threshold = (rf_prob >= threshold).astype(int)

    gb_pred_threshold = (gb_prob >= threshold).astype(int)


    f1_rf = f1_score(y_test, rf_pred_threshold)

    f1_gb = f1_score(y_test, gb_pred_threshold)


    f1_scores_rf.append(f1_rf)

    f1_scores_gb.append(f1_gb)


# Plot F1-Score vs Threshold

plt.plot(thresholds, f1_scores_rf, label='Random Forest')

plt.plot(thresholds, f1_scores_gb, label='Gradient Boosting',
color='green')

plt.xlabel('Threshold')

plt.ylabel('F1-Score')

plt.title('F1-Score vs. Decision Threshold')
```

```
plt.legend()
plt.show()
```

Executing the above code cell will provide us the following plot.



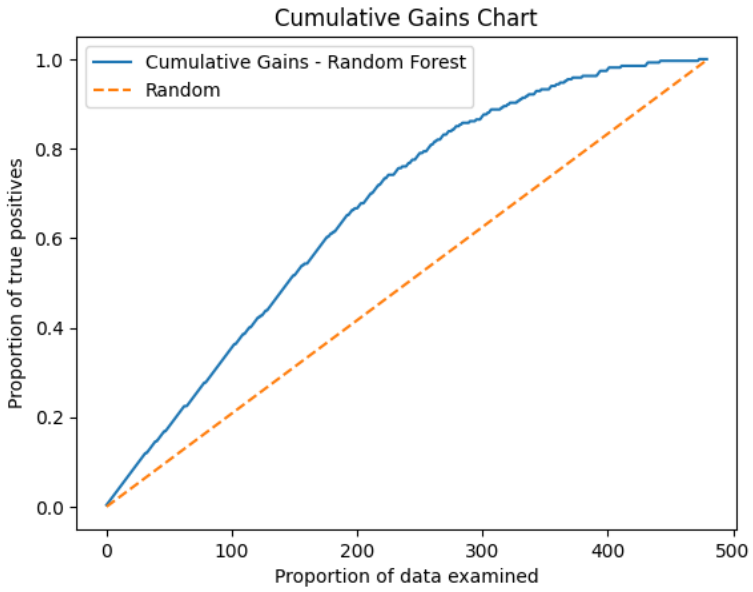
This plot demonstrates how the F1-score changes with different decision thresholds, helping to find the optimal threshold that provides the best trade-off between precision and recall.

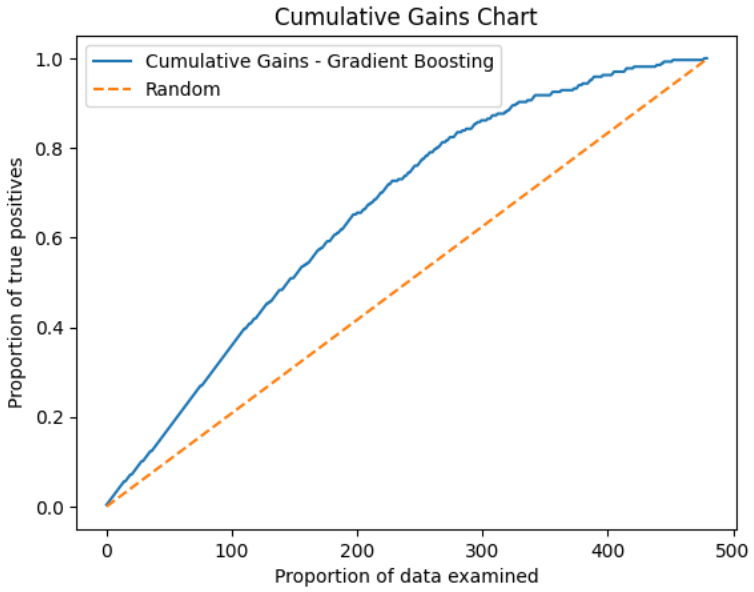
6. The cumulative gains chart is another tool used in classification models. It shows the proportion of true positives found as a function of the proportion of the dataset examined when instances are ordered by model predictions.

```
from sklearn.metrics import roc_auc_score


def plot_cumulative_gains(y_true, y_pred_proba, model_name):
    sorted_indices = np.argsort(y_pred_proba)[::-1]
    sorted_true = np.array(y_true)[sorted_indices]

    cumulative_gains = np.cumsum(sorted_true) / np.sum(sorted_true)
    random_line = np.arange(0, 1, 1 / len(y_true))
```

```
    plt.plot(cumulative_gains, label=f'Cumulative Gains -
{model_name}')
    plt.plot(random_line, '--', label='Random')
    plt.title('Cumulative Gains Chart')
    plt.xlabel('Proportion of data examined')
    plt.ylabel('Proportion of true positives')
    plt.legend()
    plt.show()

# Gain and lift chart for Random Forest
plot_cumulative_gains(y_test, rf_prob, 'Random Forest')

# Gain and lift chart for Gradient Boosting
plot_cumulative_gains(y_test, gb_prob, 'Gradient Boosting')
```

Executing the above code cell will display the following plots.





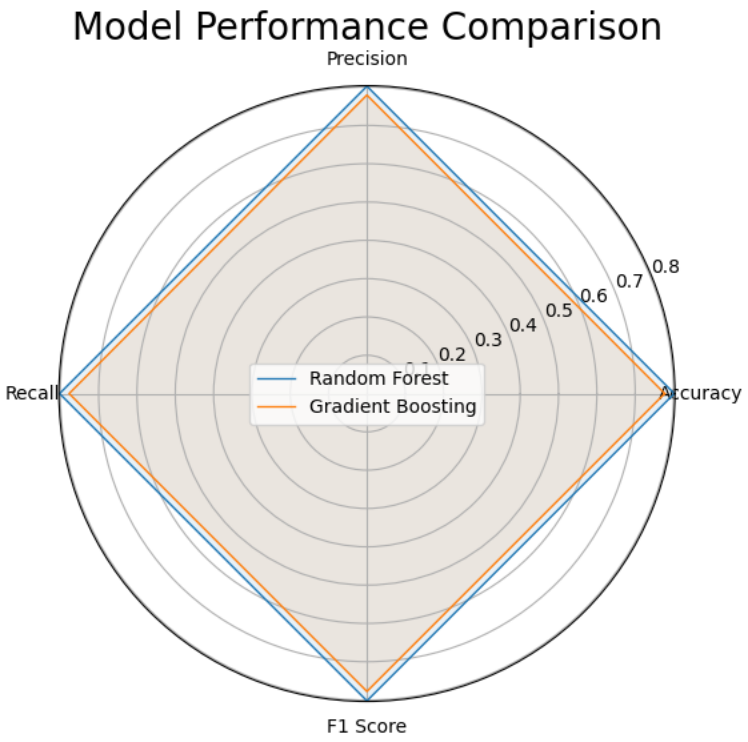
7. A spider plot (radar chart) can be used to compare models on multiple metrics in a single, intuitive plot.

```
from math import pi


metrics = ['Accuracy', 'Precision', 'Recall', 'F1 Score']

rf_metrics = [accuracy_score(y_test, rf_model.predict(X_test)),

              precision_score(y_test, rf_model.predict(X_test),
average='weighted'),

              recall_score(y_test, rf_model.predict(X_test),
average='weighted'),

              f1_score(y_test, rf_model.predict(X_test),
average='weighted')]


gb_metrics = [accuracy_score(y_test, gb_model.predict(X_test)),

              precision_score(y_test, gb_model.predict(X_test),
average='weighted'),

              recall_score(y_test, gb_model.predict(X_test),
average='weighted'),
```

```
            f1_score(y_test, gb_model.predict(X_test),
average='weighted')]


metrics_data = pd.DataFrame({
    'Metric': metrics,
    'Random Forest': rf_metrics,
    'Gradient Boosting': gb_metrics
})
def spider_plot(metrics_data):
    categories = list(metrics_data['Metric'])
    N = len(categories)
    # Create radar chart for each model
    fig, ax = plt.subplots(figsize=(6, 6),
subplot_kw=dict(polar=True))
    for model_name in ['Random Forest', 'Gradient Boosting']:
        values = metrics_data[model_name].tolist()
        values += values[:1]  # Closing the plot
        angles = [n / float(N) * 2 * pi for n in range(N)]
        angles += angles[:1]
        ax.plot(angles, values, linewidth=1, linestyle='solid',
label=model_name)
        ax.fill(angles, values, alpha=0.1)
    ax.set_xticks(angles[:-1])
    ax.set_xticklabels(categories)
    ax.set_title('Model Performance Comparison', size=20)
    ax.legend()
    plt.show()
```

```
# Create spider plot

spider_plot(metrics_data)
```

Executing the above code cell will result in displaying the following plot.



The spider plot provides a quick visual comparison across multiple metrics for different models, making it easier to understand where one model excels compared to the others.

Lab review

1. What does the ROC curve illustrate in model evaluation?
   A. The relationship between precision and recall
   B. The distribution of false positives and false negatives
   C. The trade-off between a true positive rate and a false positive rate
   D. The feature importance of the model

STOP

You have successfully completed this lab.

Lab 7: Implement Automated Hyperparameter Tuning

Lab overview

In this lab, you will:

- Set up a hyperparameter tuning job using various automated tuning methods
- Run the tuning process to identify the best hyperparameter for the model
- Analyze and interpret the tuning results to determine the best model configuration

Estimated completion time

30 minutes

Task 1: Setting up the environment and loading the dataset

This task aims to prepare the environment and data for hyperparameter tuning.

1. Create an empty folder with the name **Lab07** on the desktop of the lab environment.
2. Using Windows File Explorer, copy the dataset file **winequality-red.csv** into the Lab06 project folder, from the C:\MLOps\Data-Files folder.
3. Now copy the **Lab07.ipynb** file from the C:\MLOps\Lab-Files\Lab06 folder to the Lab06 Project folder.
4. Open the **Visual Studio Code**, click the **File** menu item, and then click the submenu **Open Folder**. In the browser window, locate the folder **Lab07** and then click **Open**. This will show the folder in the left window of the Visual Studio Code.
5. Now click on the **Terminal** menu item and then click on the sub-menu item, **New Terminal**. This will open a terminal at the bottom to enter commands.
6. Create a new virtual environment for this demo and run the following command in the terminal window to create the virtual environment.

```
virtualenv venv
```

7. Run the following command to activate the virtual environment.

```
.\venv\Scripts\activate
```

8. Now, run the following command to confirm that the virtual environment is activated.

```
python --version
```

9. A new virtual environment setup is completed for this project. To install the required libraries in the venv, run the following command in the terminal window.

```
pip install scikit-learn pandas matplotlib seaborn joblib ipykernel
optuna ipywidgets
```

10. Start by importing the necessary data manipulation, visualization, and machine learning libraries. Click the **Lab07.ipynb** file to open notebook file. Ensure that the following code is displayed in the first cell, then run the code.

```
import pandas as pd

from sklearn.model_selection import train_test_split

# Load the dataset

data = pd.read_csv('winequality-red.csv')
```

```
display(data.head())

# Split into features and target

X = data.drop(columns='quality')

y = data['quality']

# Split into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

11. Choose two models that can benefit from hyperparameter tuning (e.g., Random Forest and Gradient Boosting). This selection is based on their potential performance in the previous demos.
12. Set up parameter grids for each model. For Random Forest, include parameters like n_estimators, max_depth, and min_samples_split. For Gradient Boosting, include learning_rate, n_estimators, and max_depth.

```
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier

# Define parameter grid for Random Forest

rf_param_grid = {

    'n_estimators': [50, 100, 200],

    'max_depth': [5, 10, 20],

    'min_samples_split': [2, 5, 10]

}

# Define parameter grid for Gradient Boosting

gb_param_grid = {

    'learning_rate': [0.01, 0.1, 0.2],

    'n_estimators': [50, 100, 200],

    'max_depth': [3, 5, 7]

}
```

Setting up a well-defined hyperparameter grid is essential to explore and optimize model performance.

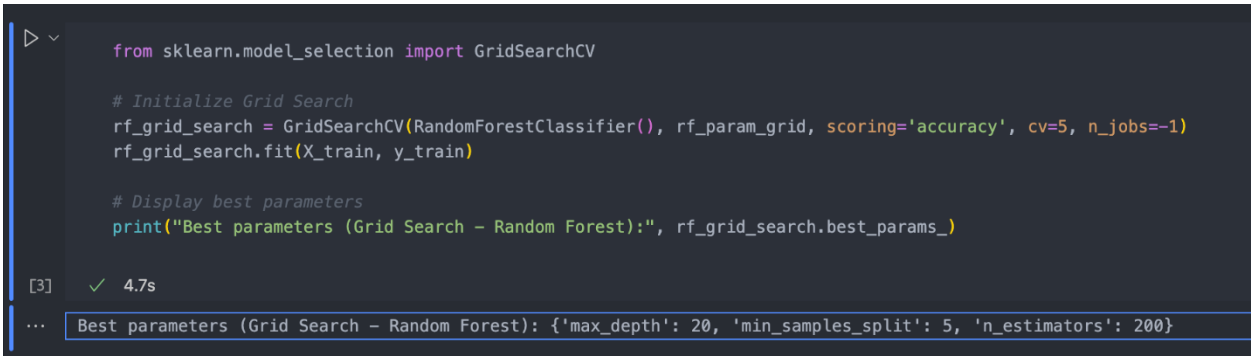
Task 2: Implementing hyperparameter tuning methods

The purpose of this task is to use **Grid Search**, **Random Search**, and **Optuna** to perform automated hyperparameter tuning.

1. Use **GridSearchCV** from **sklearn.model_selection** to search through all combinations of parameters. Train the Random Forest model using Grid Search.

```
from sklearn.model_selection import GridSearchCV

# Initialize Grid Search

rf_grid_search = GridSearchCV(RandomForestClassifier(), rf_param_grid, scoring='accuracy', cv=5, n_jobs=-1)

rf_grid_search.fit(X_train, y_train)

# Display best parameters

print("Best parameters (Grid Search - Random Forest):", rf_grid_search.best_params_)
```

   Executing the above code cell will result in the following output.

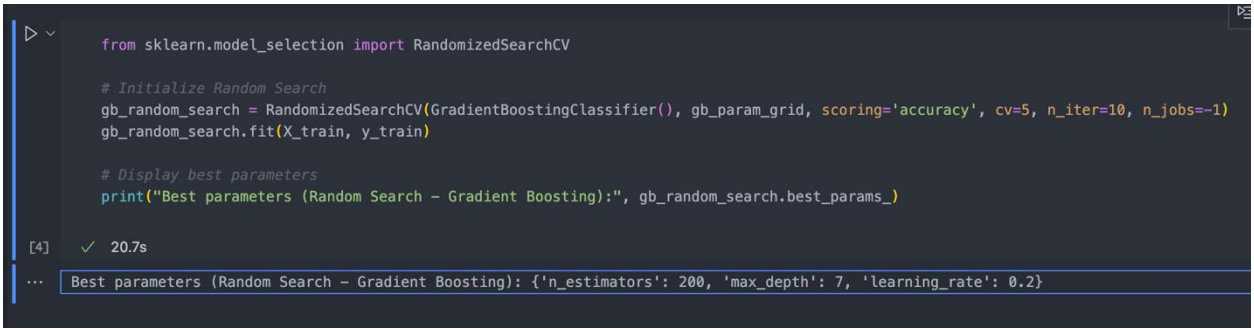


2. Use RandomizedSearchCV to randomly sample parameter combinations within the parameter grid. Train the Gradient Boosting model using Random Search.

```
from sklearn.model_selection import RandomizedSearchCV

# Initialize Random Search

gb_random_search = RandomizedSearchCV(GradientBoostingClassifier(), gb_param_grid, scoring='accuracy', cv=5, n_iter=10, n_jobs=-1)

gb_random_search.fit(X_train, y_train)
```

```
# Display best parameters

print("Best parameters (Random Search - Gradient Boosting):",
gb_random_search.best_params_)
```

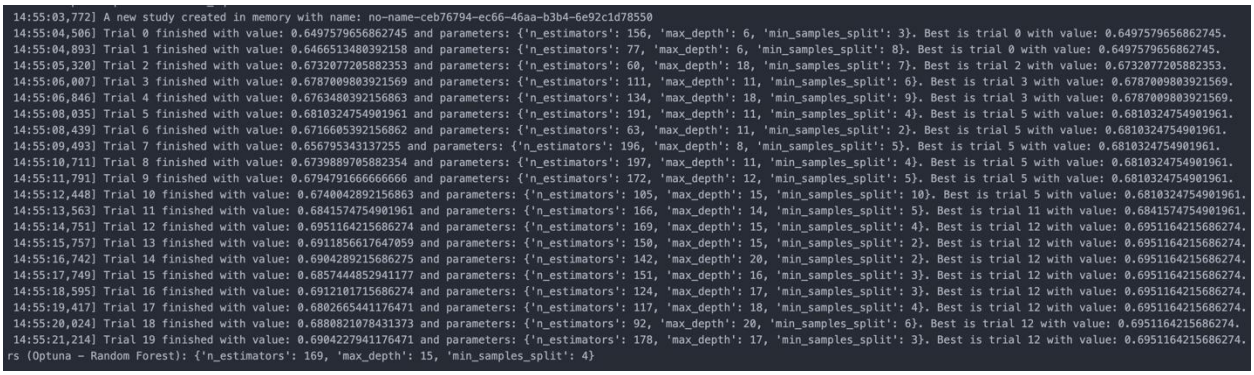
Executing the above code cell will result in the following output.



3. Use Optuna to implement Bayesian optimization for Random Forest, which can find optimal parameters more efficiently. Define an objective function and use Optuna's study.optimize to perform the search.

```
import optuna

from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import cross_val_score


def rf_objective(trial):

    n_estimators = trial.suggest_int('n_estimators', 50, 200)

    max_depth = trial.suggest_int('max_depth', 5, 20)

    min_samples_split = trial.suggest_int('min_samples_split', 2, 10)

    model = RandomForestClassifier(n_estimators=n_estimators,
max_depth=max_depth, min_samples_split=min_samples_split)

    return cross_val_score(model, X_train, y_train, cv=5).mean()

# Start Optuna Study

rf_study = optuna.create_study(direction='maximize')

rf_study.optimize(rf_objective, n_trials=20)

print("Best parameters (Optuna - Random Forest):",
rf_study.best_params)
```

Executing the above code will result in the following output.



Each method has strengths—Grid Search is exhaustive, Random Search is quicker, and Optuna is efficient with larger grids. Including all methods provides a comprehensive tuning approach.

Task 3: Analyzing and interpreting results

The objective here is to evaluate the results of each tuning method and select the best model configuration.

1. Print and compare the best hyperparameters for Random Forest and Gradient Boosting from each tuning method. Train each model using the optimal hyperparameters found and evaluate on the test set.

```
from sklearn.metrics import accuracy_score, classification_report

# Train best Random Forest model from Grid Search

best_rf = rf_grid_search.best_estimator_

y_pred_rf = best_rf.predict(X_test)

print("Random Forest (Grid Search) Accuracy:", accuracy_score(y_test,
y_pred_rf))

print("Classification Report:\n", classification_report(y_test,
y_pred_rf, zero_division=0))


# Train best Gradient Boosting model from Random Search

best_gb = gb_random_search.best_estimator_

y_pred_gb = best_gb.predict(X_test)
```

```
print("Gradient Boosting (Random Search) Accuracy:",
accuracy_score(y_test, y_pred_gb))

print("Classification Report:\n", classification_report(y_test,
y_pred_gb, zero_division=0))


# Train best Random Forest model from Optuna

best_rf_optuna =
RandomForestClassifier(**rf_study.best_params).fit(X_train, y_train)

y_pred_rf_optuna = best_rf_optuna.predict(X_test)

print("Random Forest (Optuna) Accuracy:", accuracy_score(y_test,
y_pred_rf_optuna))

print("Classification Report:\n", classification_report(y_test,
y_pred_rf_optuna, zero_division=0))
```

Executing the above code cell will generate the following output.

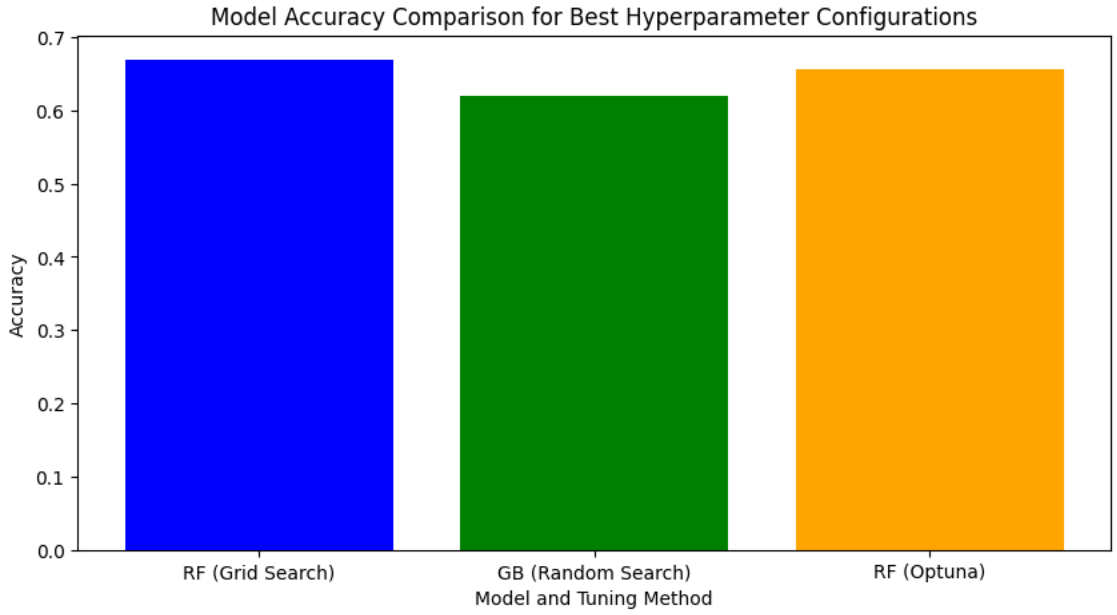
```
Random Forest (Grid Search) Accuracy: 0.66875
Classification Report:
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         1
           4       0.00      0.00      0.00        10
           5       0.72      0.75      0.73       130
           6       0.63      0.71      0.67       132
           7       0.65      0.52      0.58        42
           8       0.00      0.00      0.00         5

    accuracy                           0.67       320
   macro avg       0.33      0.33      0.33       320
weighted avg       0.64      0.67      0.65       320

Gradient Boosting (Random Search) Accuracy: 0.61875
Classification Report:
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         1
           4       0.25      0.10      0.14        10
           5       0.70      0.68      0.69       130
           6       0.58      0.67      0.62       132
           7       0.65      0.48      0.55        42
           8       0.00      0.00      0.00         5
...
    accuracy                           0.62       320
   macro avg       0.36      0.32      0.33       320
weighted avg       0.62      0.62      0.61       320
```

2. Interpret accuracy and other metrics, choosing the best model based on performance. Summarize findings and discuss how automated tuning methods helped improve model performance. This will not only confirm the best hyperparameters but also highlight the tuning impact on performance.
3. Plot a bar chart comparing the test accuracies for the best models from each tuning method.

```
import matplotlib.pyplot as plt

accuracies = [accuracy_score(y_test, y_pred_rf),
accuracy_score(y_test, y_pred_gb), accuracy_score(y_test,
y_pred_rf_optuna)]

labels = ["RF (Grid Search)", "GB (Random Search)", "RF (Optuna)"]

plt.figure(figsize=(10, 5))
```

```
plt.bar(labels, accuracies, color=['blue', 'green', 'orange'])

plt.xlabel("Model and Tuning Method")

plt.ylabel("Accuracy")

plt.title("Model Accuracy Comparison for Best Hyperparameter
Configurations")

plt.show()
```

Executing the above code cell will show the following plot.



Visualization aids in interpreting results, making it easier to explain the impact of tuning and selecting the final model.

Lab review

1. Why is hyperparameter tuning important in machine learning?
   A. To reduce the size of the dataset
   B. To increase the computational efficiency of the model
   C. To optimize the model's performance by finding the best configuration
   D. To ensure the model does not overfit

STOP

You have successfully completed this lab.

Challenge Lab 2: Experiment Tracking and Hyperparameter Optimization in MLOps

Lab overview

In this lab, you will:

- Set up MLflow for tracking model training experiments
- Log model parameters, metrics, and artifacts
- Train two ML models and track both models' performance using MLflow
- Implement hyperparameter tuning using Grid Search for Random Forest model
- Compare the performance of the tuned model to the baseline model and log the final results in MLflow

Estimated completion time

45 minutes

Task 1: Setting up the environment and MLflow for experiment tracking

This task aims to set up the working environment, import necessary libraries, and load the dataset into the workspace. This step lays the foundation for the entire lab.

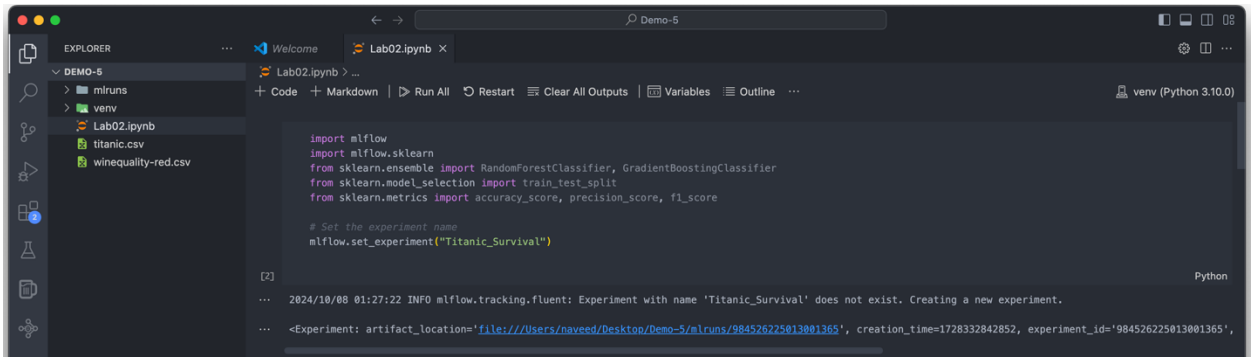
1. Create a new, empty project folder with the name SC_Lab02 in the lab environment.
2. Now copy the dataset file titanic.csv from the C:\MLOps\Data-Files folder, into the project folder. Open the folder in Visual Studio Code.
3. Open a new Terminal window, create a new virtual environment and activate it.
4. Use pip to install the required libraries in the venv, the required libraries are:

```
mlflow scikit-learn pandas matplotlib ipykernel
```

5. Create a new ipynb file in the project folder called SC_Lab02.ipynb. Then start by importing the necessary libraries and configure the MLflow experiment.

```
import mlflow

import mlflow.sklearn

from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, precision_score, f1_score


# Set the experiment name

mlflow.set_experiment("Titanic_Survival")
```

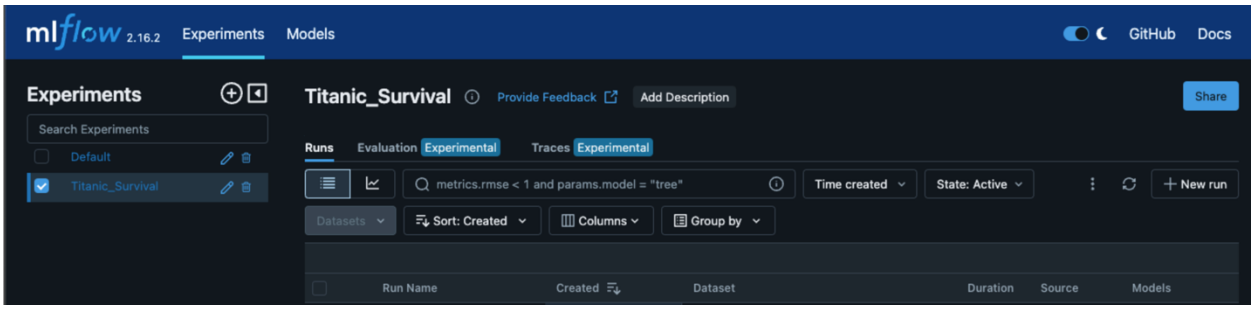
Executing the above code cell will result in creating the new experiment with name **Titanic_Survival** in the MLflow and will also create a new folder in the project with name **mlruns**.



6. Start MLflow UI (in a separate terminal) to monitor experiment runs. Run the following command in the terminal window.

```
mlflow ui
```

7. Access the MLflow UI at **http://127.0.0.1:5000** to track experiments. When you open this address in your browser it will display the following UI for MLflow.



Task 2: Training and logging baseline models (Logistic Regression and Random Forest)

The purpose of this task is to train Logistic Regression and Random Forest models and log their parameters, metrics, and artifacts.

1. Load and preprocess the Titanic survival dataset.

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

# Load Titanic dataset
```

```
data = pd.read_csv("titanic.csv")

# Drop rows with missing values for simplicity

data = data.dropna(subset=["Age", "Embarked", "Fare", "Pclass", "Sex",
"Survived"])

# Feature engineering

data["Sex"] = data["Sex"].apply(lambda x: 1 if x == "male" else 0)

data = pd.get_dummies(data, columns=["Embarked"])

X = data[["Pclass", "Sex", "Age", "Fare", "Embarked_C", "Embarked_Q",
"Embarked_S"]]

y = data["Survived"]

# Train-test split

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardize features

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)
```

2. Clean and prepare the data, converting categorical features to numerical ones and splitting them for training/testing.
3. Define a function that trains a model, logs parameters, metrcis and the model itself in MLflow.

```
from sklearn.metrics import accuracy_score, precision_score, f1_score

import mlflow.sklearn


def train_and_log_model(model, model_name):

    with mlflow.start_run(run_name=model_name):

        # Train the model

        model.fit(X_train, y_train)
```

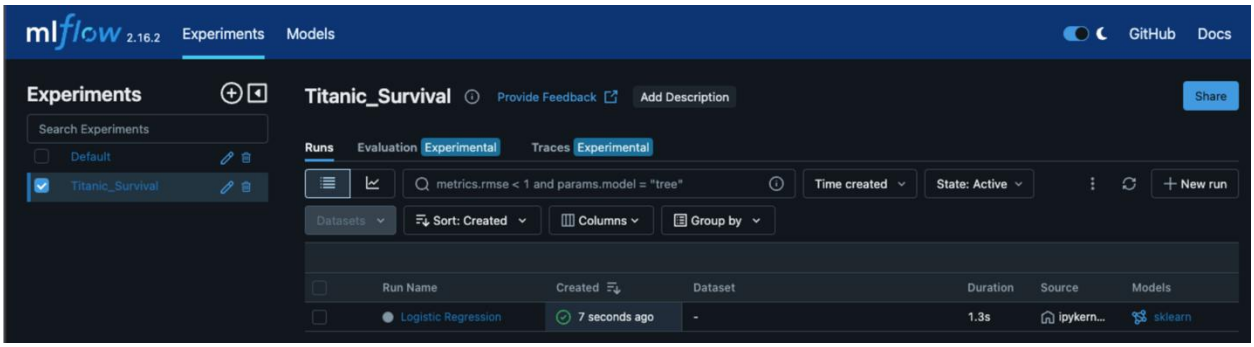
```
        y_pred = model.predict(X_test)

        # Log model parameters
        if hasattr(model, "n_estimators"):
            mlflow.log_param("n_estimators", model.n_estimators)
        if hasattr(model, "max_depth"):
            mlflow.log_param("max_depth", model.max_depth)
        if hasattr(model, "max_iter"):
            mlflow.log_param("max_iter", model.max_iter)

        # Log metrics
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred)
        f1 = f1_score(y_test, y_pred)
        mlflow.log_metric("accuracy", accuracy)
        mlflow.log_metric("precision", precision)
        mlflow.log_metric("f1_score", f1)

        # Log model
        mlflow.sklearn.log_model(model, model_name)

        print(f"Model '{model_name}' logged with accuracy:
{accuracy:.4f}, precision: {precision:.4f}, f1_score: {f1:.4f}")
```

The function starts an MLflow run, logs model parameters and metrics, and saves the model artifact in MLflow.

4. Train and log the Logistic Regression model.

```
from sklearn.linear_model import LogisticRegression


logreg = LogisticRegression(max_iter=1000)

train_and_log_model(logreg, "Logistic Regression")
```

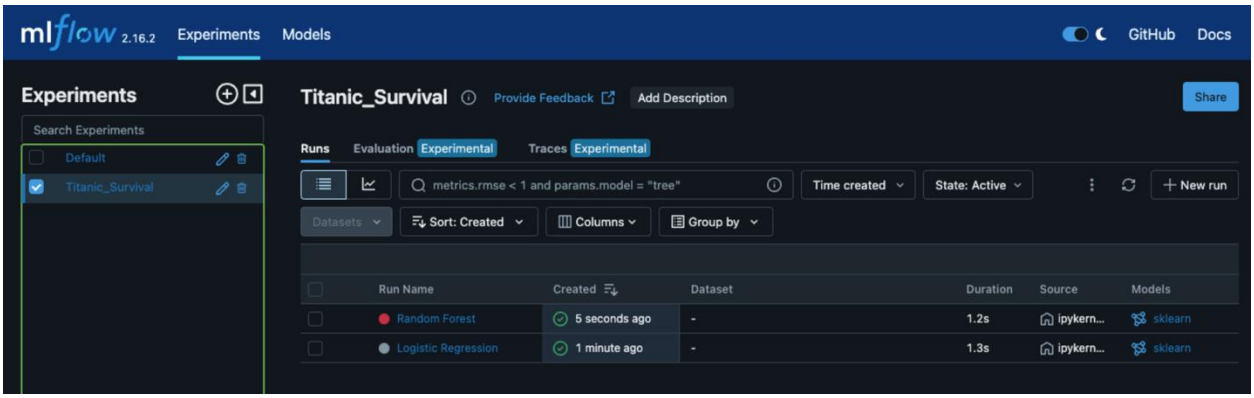
Executing the above code cell will provide the metrics for this model at the output and log the parameters on the MLflow. Now the MLflow UI should display the model.



5. Click on the Logistic Regression model to observe all the information, metrics, and artifacts.
6. Train and log Random forest model.

```
from sklearn.ensemble import RandomForestClassifier


rf = RandomForestClassifier(n_estimators=100, max_depth=5,
random_state=42)

train_and_log_model(rf, "Random Forest")
```

Executing the above code cell will provide the metrics for this model at the output and log the parameters on MLflow.



Both the models are now available on the MLflow UI. You can compare the performance and metrics on this UI.

Task 3: Implementing hyperparameter tuning for Random Forest

The objective here is to use **Grid Search** to tune the Random Forest model and log the best parameters and results.

1. Set up hyperparameter tuning for Random Forest model.

```
from sklearn.model_selection import GridSearchCV


# Define parameter grid for Grid Search
param_grid = {
    "n_estimators": [50, 100, 200],
    "max_depth": [3, 5, 10],
    "min_samples_split": [2, 5, 10]
}


# Configure Grid Search
grid_search = GridSearchCV(
```

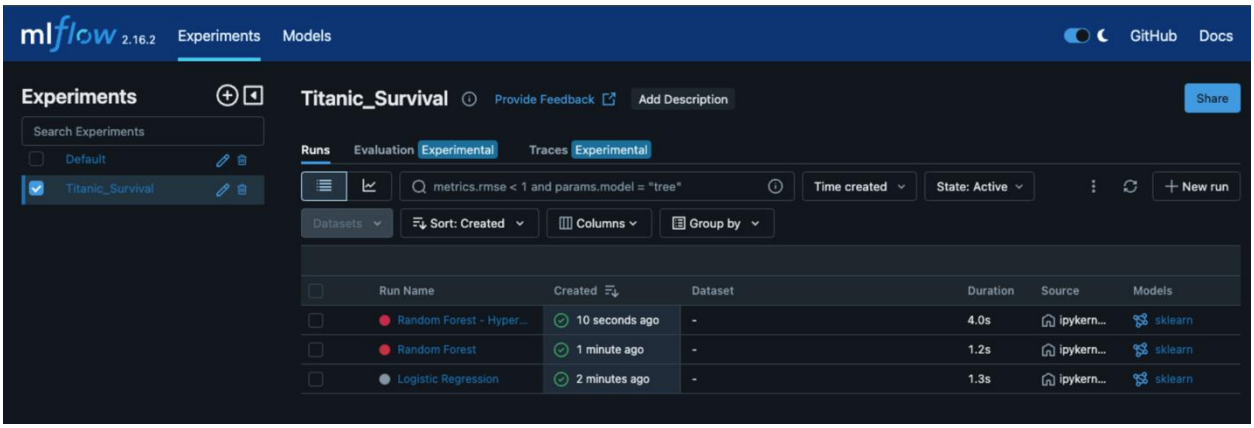
```
    estimator=RandomForestClassifier(random_state=42),
    param_grid=param_grid,
    scoring="accuracy",
    cv=5,
    n_jobs=-1,
    verbose=1
)
```

2. Run the Grid Search and log best model.

```
with mlflow.start_run(run_name="Random Forest - Hyperparameter
Tuning") as run:
    # Fit Grid Search
    grid_search.fit(X_train, y_train)
    best_rf = grid_search.best_estimator_
    y_pred = best_rf.predict(X_test)

    # Log best parameters
    mlflow.log_param("best_n_estimators",
grid_search.best_params_["n_estimators"])
    mlflow.log_param("best_max_depth",
grid_search.best_params_["max_depth"])
    mlflow.log_param("best_min_samples_split",
grid_search.best_params_["min_samples_split"])

    # Log metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
```

```
    mlflow.log_metric("accuracy", accuracy)

    mlflow.log_metric("precision", precision)

    mlflow.log_metric("f1_score", f1)

    # Log the best model

    mlflow.sklearn.log_model(best_rf, "Tuned Random Forest Model")

    print(f"Tuned Random Forest Model logged with accuracy:
{accuracy:.4f}, precision: {precision:.4f}, f1_score: {f1:.4f}")
```

This code performs hyperparameter tuning with Grid Search, logs the best parameters, and evaluates the model's performance.



Task 4: Comparing the performance of the baseline and tuned models

The objective here is to compare the performance of the baseline and tuned models to analyze improvements due to hyperparameter tuning.

1. In the MLflow UI (http://127.0.0.1:5000), navigate through the runs of Logistic Regression, Random Forest, and Tuned Random Forest Model to compare accuracy, precision, and F1 score.
2. Observe whether the tuned Random Forest model has improved metrics compared to the baseline Random Forest model. Use this analysis to make decisions on the best-performing model.

Lab review

1. What file is created by MLflow to store experiment data locally?
   A. mlflow_experiments.json
   B. ml_runs.json
   C. mlruns folder
   D. experiment_tracking.db

STOP

You have successfully completed this lab.

Lab 8: Dockerize and Deploy ML Models on Cloud Platforms

Lab overview

In this lab, you will:

- Build a Dockerfile to containerize an ML model
- Create and run the Docker container locally
- Push the Docker image to a repository (e.g., Docker Hub)
- Deploy the container on a cloud platform (e.g., Azure)

Estimated completion time

25 minutes

> **Note**
>
> This lab is launched as a separate lab deployment, due to the requirements for Azure connectivity.
>
> Save your current lab and launch the Lab 8 profile from your Skillable Lab login access.

Task 1: Training the ML model

In this task, you will prepare and train a machine-learning model on the Wine Quality dataset. This trained model will predict wine quality based on several input features.

1. Create an empty folder with the name **Lab08** on the desktop of the lab environment.
2. Using Windows File Explorer, copy the 5 files, **app.py**, **Dockerfile**, **index.html**, **requirements.txt**, and **train_model.py**, into the Lab08 project folder, from the C:\MLOps\Lab-Files\Lab08 folder.
3. Open **Visual Studio Code**, click the **File** menu item, and then click the submenu **Open Folder**. In the browser window, locate the folder Lab08 and then click **Open**. This will show the folder in the left window of Visual Studio Code.
4. Click on the **Terminal** menu item and then the sub-menu item, **New Terminal**. This will open a terminal at the bottom to enter commands.
5. Now, create a new virtual environment for this lab and run the following command in the terminal window to create the virtual environment.

```
virtualenv venv
```

6. Run the following command to activate the virtual environment.

```
.\venv\Scripts\activate
```

7. Now, run the following command to confirm that the virtual environment is activated.

```
python --version
```

8. A new virtual environment setup is completed for this project. To install the required libraries in the venv, run the following commands in the terminal window.

```
pip install pandas scikit-learn flask
```

9. Using Windows File Explorer, copy the dataset file **winequality-red.csv** into the Lab08 project folder, from the C:\MLOps\Data-Files folder.
10. Click on the Python file called **train_model.py** in the project folder and verify the following code.

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

import pickle

# Load dataset
```

```
data = pd.read_csv("winequality-red.csv")

X = data.drop("quality", axis=1)  # Features

y = data["quality"]  # Target variable

# Split data

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Train the model

model = LinearRegression()

model.fit(X_train, y_train)

# Save the model

with open("wine_quality_model.pkl", "wb") as f:

    pickle.dump(model, f)

print("Model trained and saved as wine_quality_model.pkl")
```

11. In your terminal, run the following command.

```
python train_model.py
```

After executing the above command, you should see a message saying, **"Model trained and saved as wine_quality_model.pkl."**

Task 2: Creating a Flask web application

In this task, you will create a simple web interface that allows users to input wine features and receive a prediction on the wine quality.

1. Organize the project into appropriate folders and files for the Flask application. In the project directory, create the folders with the names **templates** and **static**.
2. Click on the file named **app.py** in the project folder and verify the following code.

```
from flask import Flask, request, jsonify, render_template

import pickle

import numpy as np

# Load the model

with open("wine_quality_model.pkl", "rb") as f:
```

```
    model = pickle.load(f)

# Initialize Flask app

app = Flask(__name__)

@app.route('/')

def home():

    return render_template("index.html")

@app.route('/predict', methods=['POST'])

def predict():

    # Extract features from form

    data = [float(request.form[feature]) for feature in
request.form.keys()]

    prediction = model.predict([data])

    return render_template("index.html", prediction_text=f"Predicted
Wine Quality: {prediction[0]:.2f}")

if __name__ == "__main__":

    app.run(debug=True, host="0.0.0.0", port=5000)
```

3. Move the file named **index.html** into the **templates** folder, in the project window. Click on the file and confirm that the following content is present. This content will create an HTML form that accepts wine features and displays the predicted wine quality.

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-
scale=1.0">

    <title>Wine Quality Prediction</title>

</head>
```

```
<body style="font-family: Arial, sans-serif; background-color:
#f8f9fa; color: #333; display: flex; justify-content: center; align-
items: center; height: 100vh; margin: 0;">


    <div style="width: 100%; max-width: 600px; padding: 20px;
background-color: #fff; border: 1px solid #ddd; border-radius: 10px;
box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);">

        <h1 style="text-align: center; color: #b22222;">Wine Quality
Prediction App</h1>

        <h2 style="color: #555; text-align: center; margin-top:
10px;">Enter Wine Features:</h2>


        <form action="/predict" method="post" style="display: flex;
flex-wrap: wrap; gap: 10px; justify-content: space-between;">

            <div style="flex: 1 1 48%;">

                <label for="fixed_acidity" style="font-weight:
bold;">Fixed Acidity:</label>

                <input type="text" id="fixed_acidity"
name="fixed_acidity" required style="width: 100%; padding: 8px;
border: 1px solid #ccc; border-radius: 5px;">

            </div>

            <div style="flex: 1 1 48%;">

                <label for="volatile_acidity" style="font-weight:
bold;">Volatile Acidity:</label>

                <input type="text" id="volatile_acidity"
name="volatile_acidity" required style="width: 100%; padding: 8px;
border: 1px solid #ccc; border-radius: 5px;">

            </div>

            <div style="flex: 1 1 48%;">

                <label for="citric_acid" style="font-weight:
bold;">Citric Acid:</label>
```

```
                <input type="text" id="citric_acid" name="citric_acid"
required style="width: 100%; padding: 8px; border: 1px solid #ccc;
border-radius: 5px;">

            </div>

            <div style="flex: 1 1 48%;">

                <label for="residual_sugar" style="font-weight:
bold;">Residual Sugar:</label>

                <input type="text" id="residual_sugar"
name="residual_sugar" required style="width: 100%; padding: 8px;
border: 1px solid #ccc; border-radius: 5px;">

            </div>

            <div style="flex: 1 1 48%;">

                <label for="chlorides" style="font-weight:
bold;">Chlorides:</label>

                <input type="text" id="chlorides" name="chlorides"
required style="width: 100%; padding: 8px; border: 1px solid #ccc;
border-radius: 5px;">

            </div>

            <div style="flex: 1 1 48%;">

                <label for="free_sulfur_dioxide" style="font-weight:
bold;">Free Sulfur Dioxide:</label>

                <input type="text" id="free_sulfur_dioxide"
name="free_sulfur_dioxide" required style="width: 100%; padding: 8px;
border: 1px solid #ccc; border-radius: 5px;">

            </div>

            <div style="flex: 1 1 48%;">

                <label for="total_sulfur_dioxide" style="font-weight:
bold;">Total Sulfur Dioxide:</label>

                <input type="text" id="total_sulfur_dioxide"
name="total_sulfur_dioxide" required style="width: 100%; padding: 8px;
border: 1px solid #ccc; border-radius: 5px;">

            </div>
```

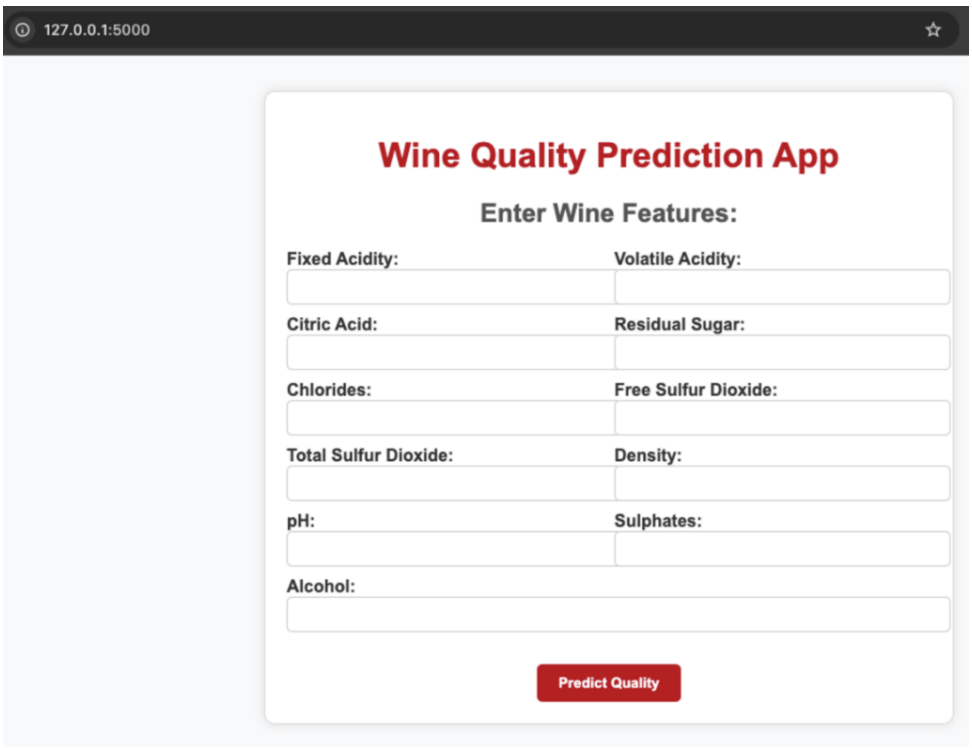
```
            <div style="flex: 1 1 48%;">

                <label for="density" style="font-weight:
bold;">Density:</label>

                <input type="text" id="density" name="density"
required style="width: 100%; padding: 8px; border: 1px solid #ccc;
border-radius: 5px;">

            </div>

            <div style="flex: 1 1 48%;">

                <label for="pH" style="font-weight: bold;">pH:</label>

                <input type="text" id="pH" name="pH" required
style="width: 100%; padding: 8px; border: 1px solid #ccc; border-
radius: 5px;">

            </div>

            <div style="flex: 1 1 48%;">

                <label for="sulphates" style="font-weight:
bold;">Sulphates:</label>

                <input type="text" id="sulphates" name="sulphates"
required style="width: 100%; padding: 8px; border: 1px solid #ccc;
border-radius: 5px;">

            </div>

            <div style="flex: 1 1 48%;">

                <label for="alcohol" style="font-weight:
bold;">Alcohol:</label>

                <input type="text" id="alcohol" name="alcohol"
required style="width: 100%; padding: 8px; border: 1px solid #ccc;
border-radius: 5px;">

            </div>

            <div style="width: 100%; text-align: center;">

                <input type="submit" value="Predict Quality"
style="margin-top: 20px; background-color: #b22222; color: white;
font-weight: bold; padding: 10px 20px; border: none; border-radius:
5px; cursor: pointer;">
```

```
            </div>
        </form>


        {% if prediction_text %}
            <h3 style="text-align: center; color: #555; margin-top:
20px;">{{ prediction_text }}</h3>
        {% endif %}
    </div>
</body>
</html>
```

4. Enter the following command in the terminal to run the Flask app locally.

```
python app.py
```

5. Open a browser and go to http://127.0.0.1:5000/ (Wine Quality Predictor bookmark). You will see the following form.



6. Enter the following values in the app to see if it provides the prediction.
   - fixed acidity = 7.4
   - volatile acidity = 0.70
   - citric acid = 0.00
   - residual sugar = 1.9
   - chlorides = 0.07
   - free sulfur dioxide = 11.0
   - total sulfur dioxide = 34.0
   - density = 0.9978
   - pH = 3.51
   - sulphates = 0.56
   - alcohol = 9.4
7. Click on the red button **Predict Quality,** and you will see the following result.

8. After reviewing, use **CTRL+C** in the terminal window to stop executing the Flask app.

Task 3: Dockerizing the Flask app

In this task, you will containerize the Flask web application using **Docker**, making it easier to deploy across different environments.

1. Write a Dockerfile that defines how the Flask application should be built and run within a container. In the project folder, verify the contents of a file named **Dockerfile** to be as follows.

```
FROM python:3.9-slim

COPY . /app

WORKDIR /app

RUN pip install -r requirements.txt

EXPOSE 5000

CMD ["python","app.py"]
```

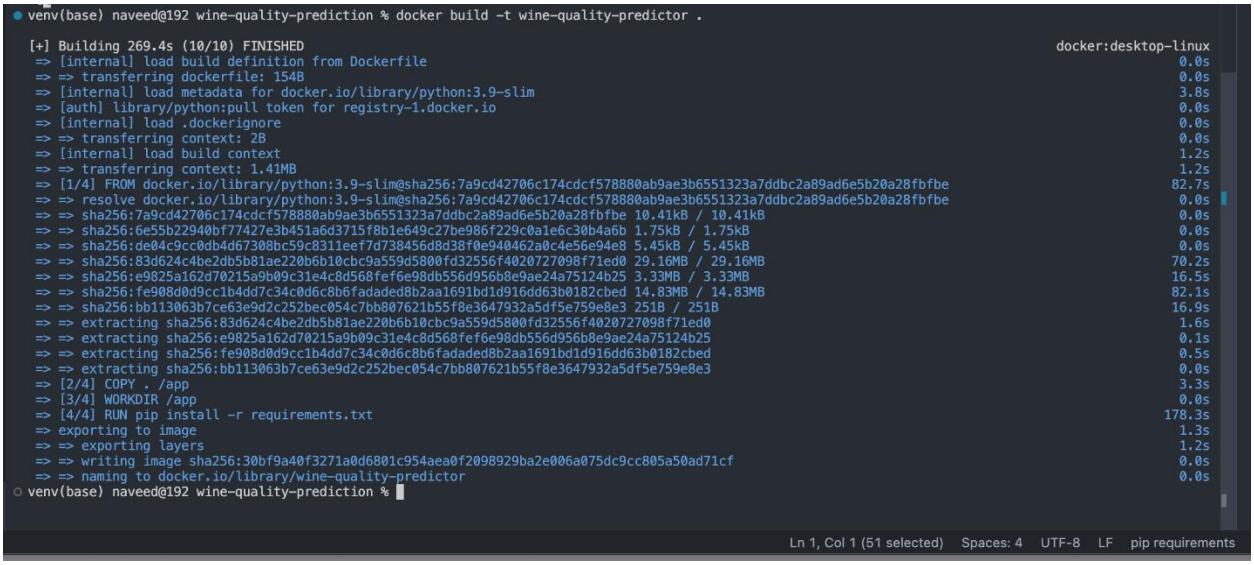
2. Next, click on the requirements.txt file and check it contains the following dependencies.

```
Flask

scikit-learn

pandas

numpy

matplotlib

gunicorn
```

3. Enter the following command in the terminal to build the docker image.

```
docker build -t wine-quality-predictor .
```

Observe the following on the terminal window after executing the above command.



4. Enter the following command in the terminal to run the docker container.

```
docker run -p 5000:5000 wine-quality-predictor
```

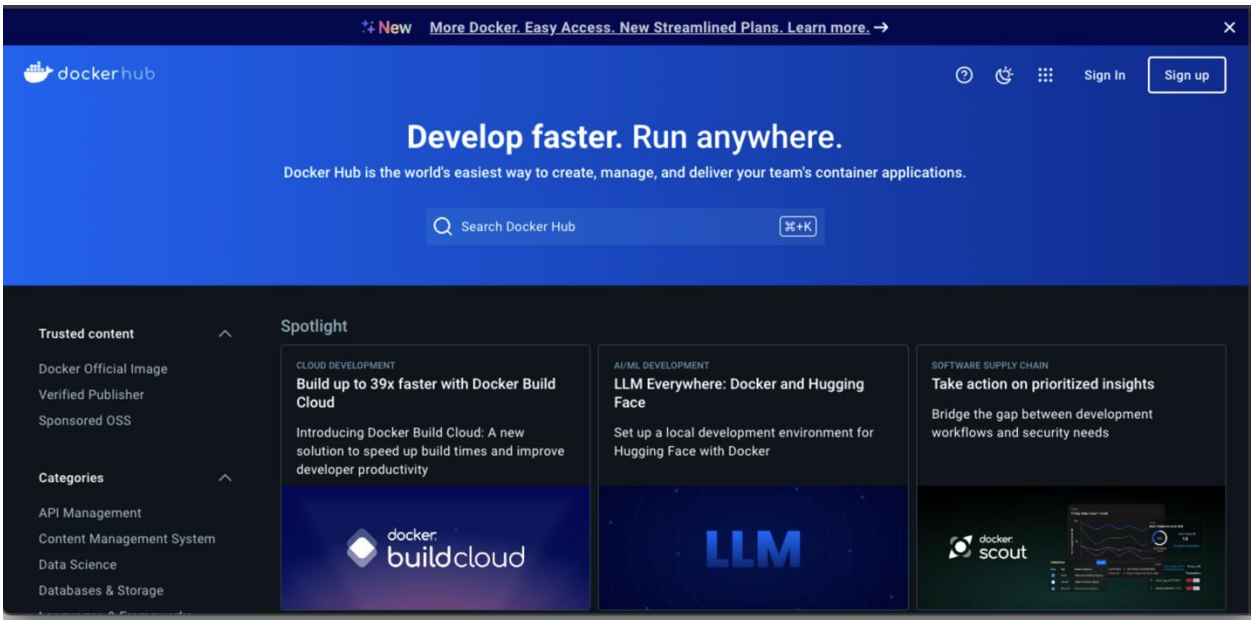
Observe the following in the terminal window after executing the above command.

5. Open the browser and go to http://127.0.0.1:5000 to test the application. This time the application is running from the docker container. Use CTRL+C to quit.

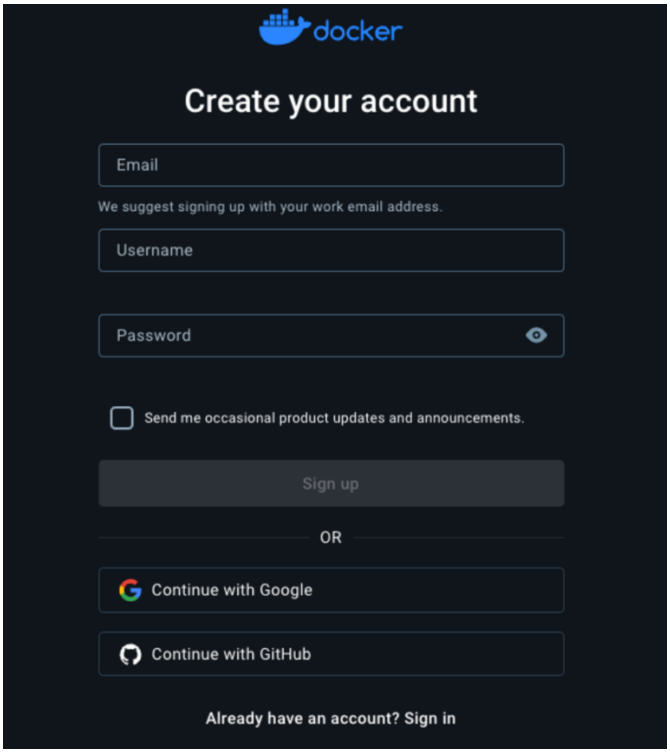
Task 4: Pushing Docker image to Docker hub

In this task, you will push the Docker image to the Docker hub for deployment in the cloud.

1. Open the browser and enter the URL **https://hub.docker.com/**.



2. Click on the **Sign up** button and create an account, or if you already have an account then sign in.



3. Enter the following command in the VS Code Terminal to log in to Docker hub.

```
docker login -u yourusername
```

   Replace the yourusername with your actual username on the Docker hub. Then type in your Docker password, when prompted.

> **Note**
>
> If using SSO and/or 2FA for your Docker account, you will need to log in with a Personal Access Token, as follows:
>
> **docker login -u yourusername**
>
> Then paste in your PAT at the password prompt.

4. Tag the Docker image that you have created by executing the following command.

```
docker tag wine-quality-predictor yourusername/wine-quality-predictor
```

   Replace the yourusername with your actual username on the Docker hub.

5. Execute the following command to push the image to the Docker hub.

```
docker push yourusername/wine-quality-predictor
```

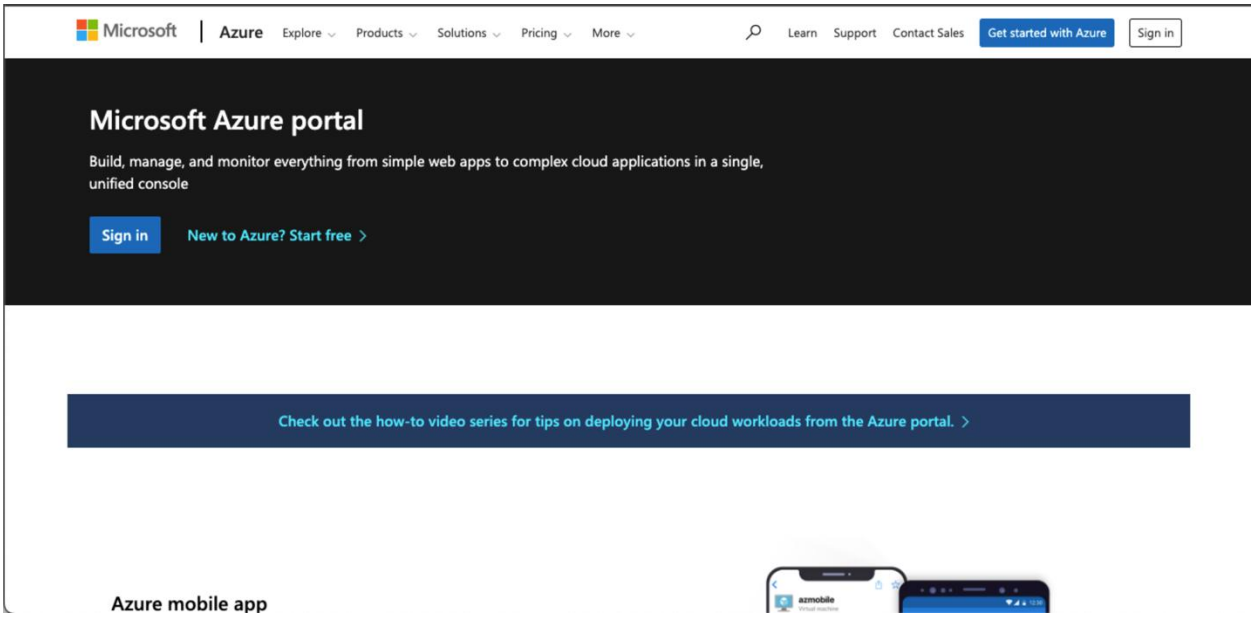
   Replace the yourusername with your actual username on the Docker hub.

After executing this command the container image will be pushed to the Docker hub and can be verified.

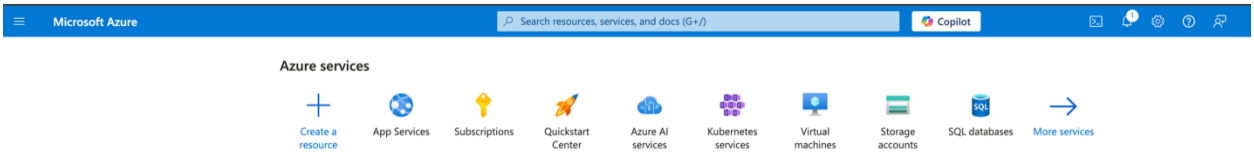
Task 5: Deploying the app on Azure

In this task, you will deploy the Dockerized Flask application on **Azure** for global access.

1. Open the browser and enter the URL **https://azure.microsoft.com/en-us/get-started/azure-portal**.



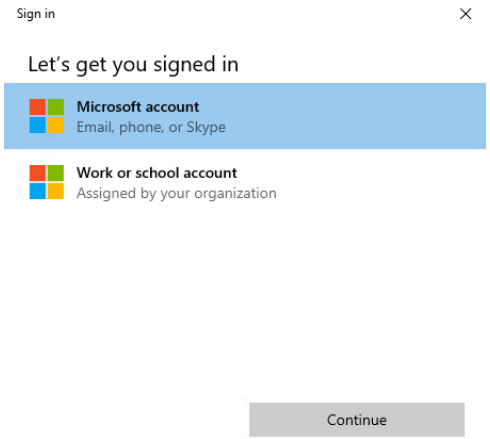
2. Sign in to the Azure portal using the credentials provided on the **Resources** tab of the Lab Access screen.
3. Click on the **Username** box to transfer the login user to the web page.
4. Likewise, click on the **TAP** box to transfer the Temporary Access Pass code to the web page.
5. After signing in to the portal, press **Cancel** on the Getting Started screen and then you will see the following dashboard.



6. There are two ways to set up the Web app on the Azure portal: one is to click on **App Services**, which is straightforward, and the other is to use command line access.
7. The first step is to log in to Azure by executing the following command in the VS Code Terminal screen.

```
az login
```

This will bring up a Sign in screen (**Note**: it may be hidden behind any other open windows).



8. At the Sign in screen, select **Work or School account** then sign in using the provided Azure credentials (Username and TAP code). Click **No, sign in to this app only**.
9. In the VS Code terminal window press **Enter** for no changes to the subscription and tenant.
10. After successful login to Azure, the next step would be to create a **Resource Group**. However, owing to the required security profile for the lab, this has already been created. Verify the group exists with the following command.

```
az group show --name myResourceGroup
```

The following is an example of the expected response.

```
(venv) PS C:\Users\student\Desktop\Lab08> az group show --name myResourceGroup
{
  "id": "/subscriptions/ff7d29de-f30f-41fb-83fd-abcc3ebb0aec/resourceGroups/myResourceGroup",
  "location": "eastus2",
  "managedBy": null,
  "name": "myResourceGroup",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": {
    "LODManaged": "lod",
    "LabInstance": "46685325",
    "LabProfile": "177700",
    "PoolOrgId": "2279",
    "ProfileOrgId": "3510",
    "SeriesId": "33672",
    "TS": "133785693500989020"
  },
  "type": "Microsoft.Resources/resourceGroups"
}
(venv) PS C:\Users\student\Desktop\Lab08>
```

11. Execute the following command in the terminal to create the web app on the Azure portal.

```
az webapp create --resource-group myResourceGroup --plan
myAppServicePlan --name wine-quality-predictor-app-<LabInstanceId> --
container-image-name <yourusername>/wine-quality-predictor
```

    Replace the <yourusername> with your docker hub username, in the above command, according to your actual information.

    Replace the <LabInstanceId> with the ID presented on the Instructions tab of the lab interface. The Web App name must be unique on the lab Azure account.

12. Set the Docker credentials on Azure by executing the following command.

```
az webapp config container set --name wine-quality-predictor-app-
<LabInstanceId> --resource-group myResourceGroup --container-registry-
url https://index.docker.io --container-registry-user <yourusername> -
-container-registry-password <yourpassword>
```

    Replace the fields yourusername and yourpassword for accessing the correct container from the Docker hub.

    Replace the <LabInstanceId> with the ID presented on the Instructions tab of the lab interface.

13. Go to the Azure portal and click on the App Services icon to view the wine-quality-predictor-app.
14. Click on the app name and from here you can see the overview and activity log, and see how the application is running.
15. From the Overview screen, click on the Default domain link to open the app is a browser tab. Note that it may take a while for the app to start up fully.

> **Note**
>
> Be sure to log out from the Azure portal and your Docker hub account before closing down this lab.

16. When complete, close the lab deployment using the Exit Lab link at the top right of the Lab Access window.

Lab review

1. Which command is used to build the Docker image for the Flask application?
   A. docker run -p 5000:5000
   B. docker build -t wine-quality-predictor
   C. docker push wine-quality-predictor
   D. docker tag wine-quality-predictor yourusername/wine-quality-predictor

STOP

You have successfully completed this lab.

Lab 9: Implement CI/CD Pipelines for ML with GitHub Actions

Lab overview

In this lab, you will:

- Set up a CI/CD pipeline for the Iris dataset with GitHub actions
- Train a machine learning model using scikit-learn
- Version and deploy the model using Continuous Machine Learning
- Deploy to GitHub pages from the automated workflow

Estimated completion time

25 minutes

Task 1: Setting up the project repository and loading the dataset

In this task, you will create a GitHub repository and set up the basic project folder structure for this lab.

1. Open a Web browser, log in to GitHub (if necessary), and create a new public repository: e.g., **iris-html-ci-cd** add a description, and initialize with a **README**.
2. Open the **Visual Studio Code**, open a new terminal window and clone the repository locally, using this command:

```
git clone https://github.com/your-username/iris-html-ci-cd.git
```

3. Using Windows Explorer, copy all of the following files: **ci.yml**, **generate_html.py**, **requirements.txt** and **train_model.py** from the C:\MLOps\Lab-Files folder, into the new **iris-html-ci-cd** project folder: **C:\Users\student\iris-html-ci-cd**.
4. In VS Code, click **File > Open Folder** and search for the new project folder location (C:\Users\student\iris-html-ci-cd). The folder will show in the left window of the Visual Studio Code. Now click on the **Terminal** menu item and then the sub-menu item, **New Terminal**. This will open a terminal.
5. Create a new virtual environment for this task. Run the following command in the terminal window to create the virtual environment.

```
virtualenv venv
```

6. Run the following command to activate the virtual environment.

```
.\venv\Scripts\activate
```

7. Now, run the following command to confirm that the virtual environment is activated.

```
python --version
```

8. A new virtual environment setup is completed for this project. To install the required libraries in the virtual environment, run the following command in the terminal window.

```
pip install scikit-learn pandas matplotlib
```

9. Create/verify the necessary folder structure, as below (create the directories and move the files, as required, in either Windows Explorer or VS Code).

```
iris-html-ci-cd/
├── Data/
    └── iris.data
```

```
├── train_model.py
├── generate_html.py
├── requirements.txt
├── .github/
      └── workflows/
          └── ci.yml
├── index.html  # This file will be auto-generated.
├── README.md
```

10. Copy the **iris.data** file from C:\MLOps\Data-Files into the project directory **Data/** folder.

    Alternatively, execute the below command in the terminal window to automatically download the dataset in the data folder.

```
wget https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data -P Data/
```

Task 2: Training a model and generating an HTML app

The goal of this task is to process the dataset, train a model, and write a script to automatically generate the HTML file in the project folder to be used later to show the results.

1. Click on the file with the name **train_model.py** in the project folder and check/verify that the following code is present.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import json


# Load the Iris dataset
df = pd.read_csv('Data/iris.data', header=None)
```

```
df.columns = ['sepal_length', 'sepal_width', 'petal_length',
'petal_width', 'species']


# Split the data

X = df.iloc[:, :-1]

y = df.iloc[:, -1]

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)


# Train a Random Forest model

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)


# Evaluate the model

y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)

print(f"Model Accuracy: {accuracy:.2f}")


# Save results to a JSON file

results = {

    "accuracy": accuracy,

    "feature_importances": list(model.feature_importances_)

}

with open("results.json", "w") as f:

    json.dump(results, f)


print("Model training completed. Results saved to results.json.")
```

This script will train the RandomForest model and save the results (accuracy, feature importances) to a JSON file for later use.

2. Run the script by executing the following command in the terminal.

```
python train_model.py
```

3. Verify that results.json is generated and contains model accuracy and feature importances after executing the above model training script.
4. Now click on the Python file generate_html.py in the project folder and verify the contents, as below. This script will read the results.json file and generate an HTML file (index.html) for the app.

```
import json

# Load results from JSON

with open("results.json", "r") as f:

    results = json.load(f)

# Generate HTML

html_content = f"""

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-
scale=1.0">

    <title>Iris Model Results</title>

    <style>

        body {{ font-family: Arial, sans-serif; margin: 20px; }}

        h1 {{ color: #4CAF50; }}

        .results {{ margin-top: 20px; }}

    </style>

</head>

<body>
```

```
    <h1>Iris Model Results</h1>
    <div class="results">
        <p><strong>Accuracy:</strong> {results['accuracy']:.2f}</p>
        <p><strong>Feature Importances:</strong></p>
        <ul>
            <li>Sepal Length:
{results['feature_importances'][0]:.2f}</li>
            <li>Sepal Width:
{results['feature_importances'][1]:.2f}</li>
            <li>Petal Length:
{results['feature_importances'][2]:.2f}</li>
            <li>Petal Width:
{results['feature_importances'][3]:.2f}</li>
        </ul>
    </div>
</body>
</html>
"""
# Save HTML to a file
with open("index.html", "w") as f:
    f.write(html_content)

print("HTML file generated: index.html")
```

5. Run the above Python script by executing the following command in the terminal window.

```
python generate_html.py
```

6. Verify that index.html is generated with the model results (Open the file from Windows Explorer, which will open a browser window).

Task 3: Automating CI/CD with GitHub actions

This task aims to set up a CI/CD pipeline that pushes the files to the GitHub and deploy on the GitHub pages.

1. Click on the ci.yml file, which should be now located in the .github/workflows folder, and confirm that the content is as below.

```
name: CI/CD for Hosting Iris HTML App


on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]


jobs:
  build:
    runs-on: ubuntu-latest


    steps:
    - name: Checkout code
      uses: actions/checkout@v3


    - name: Set up Python
      uses: actions/setup-python@v3
      with:
        python-version: 3.9


    - name: Install dependencies
```

```
      run: |
        python -m venv venv
        source venv/bin/activate
        pip install -r requirements.txt

    - name: Train the model
      run: |
        source venv/bin/activate
        python train_model.py

    - name: Generate HTML app
      run: |
        source venv/bin/activate
        python generate_html.py

    - name: Deploy to GitHub Pages
      uses: peaceiris/actions-gh-pages@v4
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./
```

2. Click on the **requirements.txt** file and confirm that the following dependencies are listed in it.

```
pandas
scikit-learn
matplotlib
```

3. To commit and push changes to the GitHub repository, execute the following commands in the terminal window.

```
git add .
```

```
git commit -m "Set up CI/CD pipeline for GitHub Pages"
```

```
git push origin main
```

4. To enable GitHub Pages, go to the GitHub repository settings, select **Pages**, set the source to the **Deploy from a Branch**, the Branch should be set to **Main** and **/** (root). Click **Save**.
5. You should now see a message in the GitHub pages section that the site is live. Click on the **Visit site** button to view your site in a browser window or access the URL e.g., https://your-username.github.io/iris-html-ci-cd/.

Lab review

1. Which command is used to push the data to GitHub?
   A. git add
   B. git push origin main
   C. github push repo main
   D. git commit -m "Push to GitHub"

STOP

You have successfully completed this lab.

Lab 10: Monitor ML Models with Prometheus and Grafana

Lab overview

In this lab, you will:

- Install monitoring tools Prometheus and Grafana
- Train a machine learning model using scikit-learn
- Set up Prometheus to scrape metrics from an ML model
- Visualize those metrics in Grafana

Estimated completion time

25 minutes

Task 1: Setting up virtual environment and installing libraries

This task aims to create a virtual environment and install the required libraries and monitoring tools.

1. Create an empty folder with the name **Lab10** on the desktop of the lab environment.
2. Copy the **train_model.py** and **iris_test.py** files from the C:\MLOps\Lab-Files\Lab10 folder to the Lab10 project folder. All the files should then be visible in the VS Code Explorer window.
3. Open the Visual Studio Code, click the **File** menu item, and then click the submenu **Open Folder**. In the browser window, locate the folder **Lab10** and then click **Open**. This will show the folder in the left window of the Visual Studio Code.
4. Click on the **Terminal** menu item and then click on the sub-menu item, **New Terminal**. This will open a terminal at the bottom to enter commands.
5. Create a new virtual environment for this lab and run the following command in the terminal window to create the virtual environment.

```
virtualenv venv
```

6. Run the following command to activate the virtual environment.

```
.\venv\Scripts\activate
```

7. Now, run the following command to confirm that the virtual environment is activated.

```
python --version
```

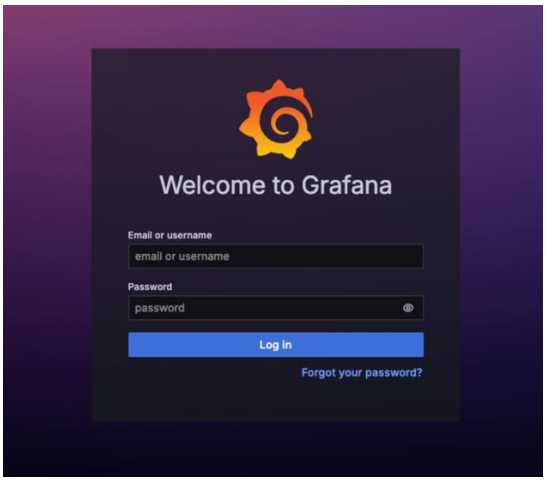
8. A new virtual environment setup is completed for this project. To install the required libraries in the venv, run the following command in the terminal window.

```
pip install scikit-learn prometheus-client uvicorn
```

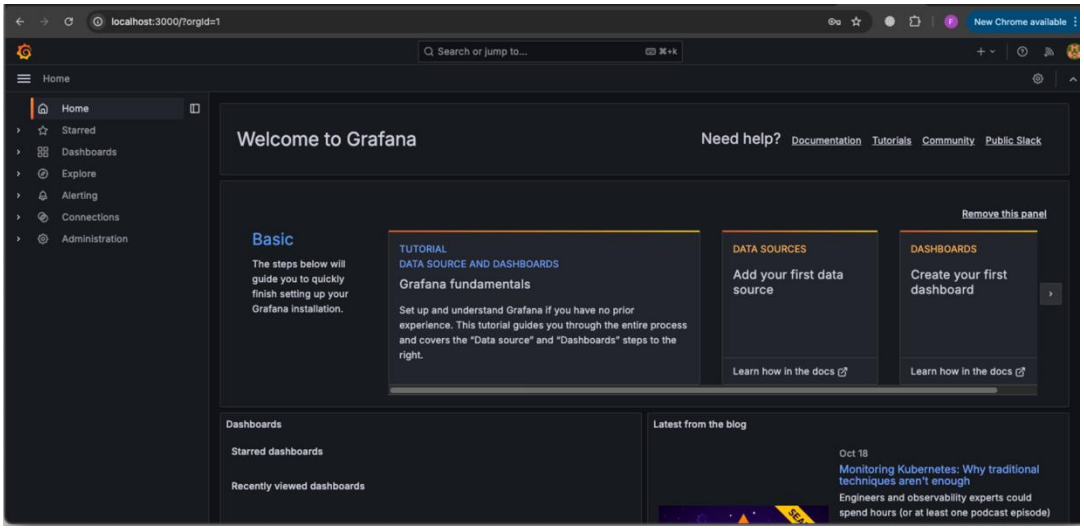
```
pip install matplotlib pandas
```

```
pip install gunicorn
```

9. Open the browser and enter the URL **http://localhost:3000/:** to view Grafana.



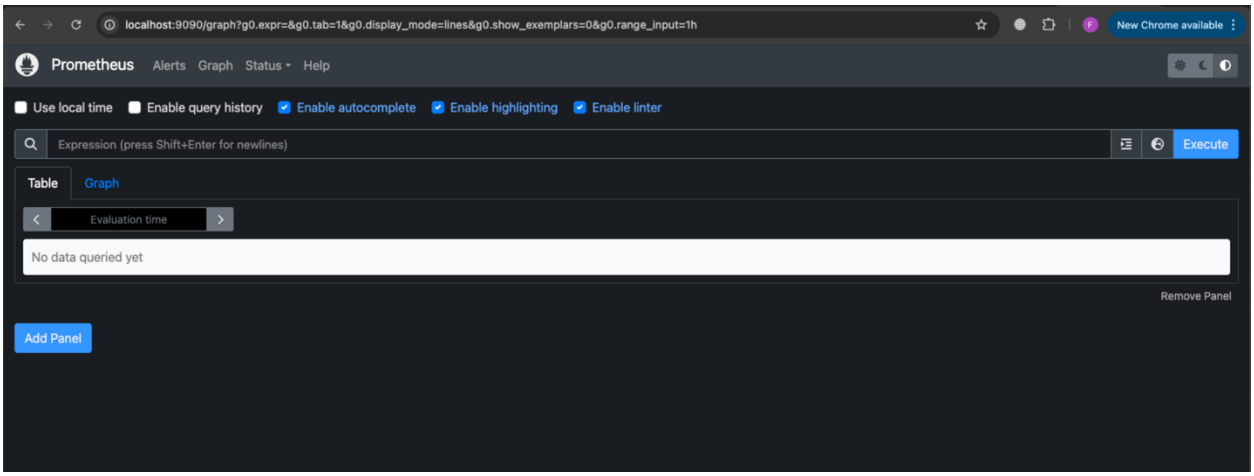
10. Log in to the dashboard with username: **admin** and password: **Pa$$w0rd!**.



11. Open a Windows command prompt, navigate to the folder **C:\prometheus** where Prometheus has been extracted, and run the following command to start the Prometheus app.

```
prometheus.exe --config.file=prometheus.yml
```

12. Now open a web browser tab and verify it is running at **http://localhost:9090**:



Task 2: Training the iris model and creating the test model script

The goal is to train the iris model, generate the test model script to serve the iris prediction model, and expose metrics to Prometheus.

1. In VS Code, click on the **train_model.py** file, verify it contains the following code.

```
from sklearn.datasets import load_iris

from sklearn.ensemble import RandomForestClassifier

import pickle

# Load Iris dataset

iris = load_iris()

X, y = iris.data, iris.target

# Train a Random Forest model

model = RandomForestClassifier()

model.fit(X, y)

# Save the model

with open("iris_model.pkl", "wb") as f:

    pickle.dump(model, f)

print("Model trained and saved as iris_model.pkl")
```

2. In the terminal window, run the training script to create the model file.

```
python train_model.py
```

The trained model will be saved as **iris_model.pkl** in the project directory.

3. Click on the **iris_test.py** file and verify its contents.

```
import time

import random

import numpy as np

from prometheus_client import Counter, Summary, Gauge,
start_http_server

from sklearn.datasets import load_iris

from sklearn.ensemble import RandomForestClassifier

import pickle


# Define Prometheus metrics

REQUEST_COUNT = Counter('request_count', 'Total number of requests to
the model')

INFERENCE_TIME = Summary('inference_time', 'Time spent processing
predictions')

PREDICTION_ACCURACY = Gauge('prediction_accuracy', 'Model prediction
accuracy (if ground truth is available)')


# Load the trained model

with open("iris_model.pkl", "rb") as f:

    model = pickle.load(f)


# Load the Iris dataset for synthetic data generation

iris = load_iris()

X, y = iris.data, iris.target
```

```
# Start the Prometheus metrics server
start_http_server(8001)

print("Prometheus metrics server running at
http://localhost:8001/metrics")

# Function to periodically call the model and record metrics
def simulate_model_requests():

    while True:
        # Increment request count
        REQUEST_COUNT.inc()

        # Simulate input data
        input_data = random.choice(X)
        ground_truth = y[np.where(X == input_data)[0][0]]

        # Start timing the inference
        start_time = time.time()
        prediction = model.predict([input_data])
        inference_time = time.time() - start_time

        # Record inference time
        INFERENCE_TIME.observe(inference_time)
        print(f"Inference Time: {inference_time:.4f} seconds")
        # Evaluate prediction accuracy (if ground truth is available)
```

```
        is_correct = int(prediction[0] == ground_truth)
        PREDICTION_ACCURACY.set(is_correct)
        print(f"Prediction Accuracy: {is_correct}")

        # Wait before the next simulation (e.g., 5 seconds)
        time.sleep(5)
# Start simulating model requests
simulate_model_requests()
```

Task 3: Setting up Prometheus to scrape metrics

This task aims to configure the Prometheus yml file to capture the exposed metrics from the test script and create the dashbaord in Grafana for visualization.

1. From the Windows command window that is running Prometheus, stop the app with CTRL+C. Edit the prometheus.yml file to include the target for your model test script. Verify the full script of the prometheus.yml as below.

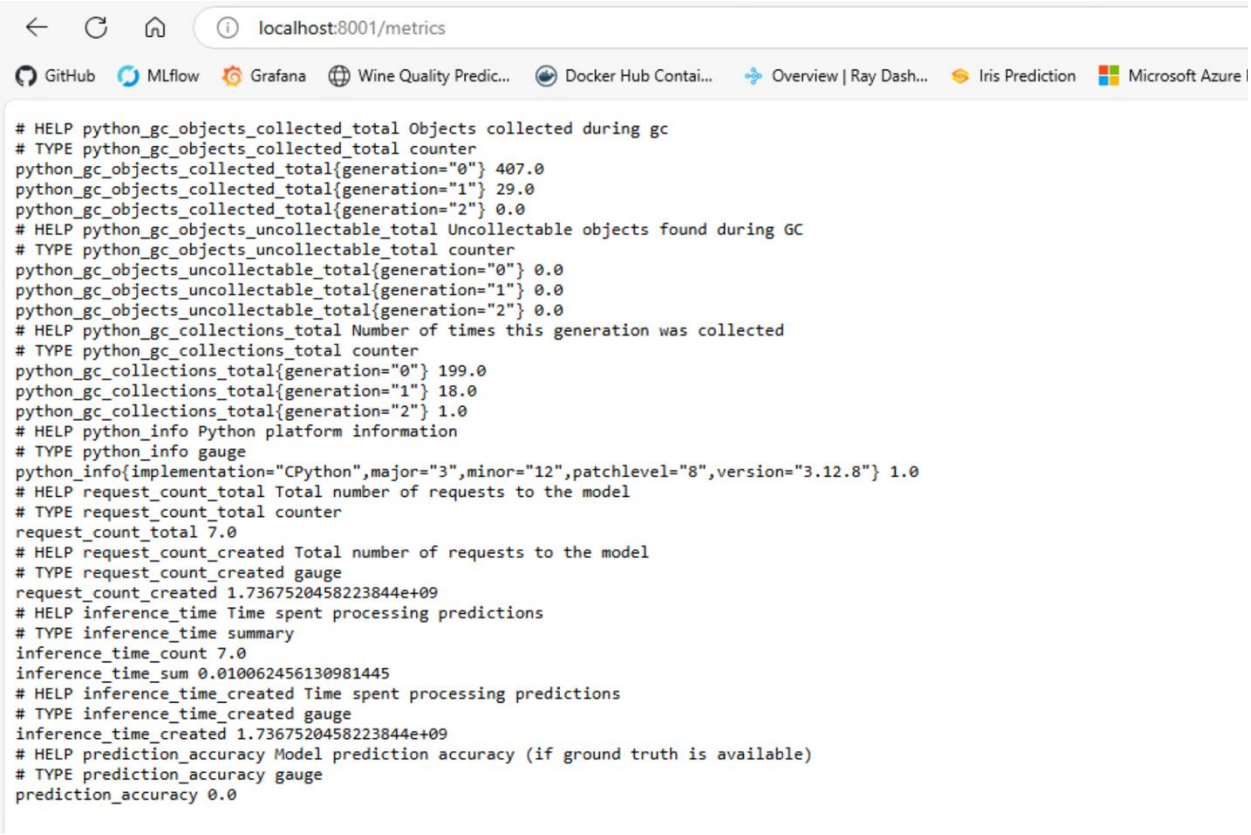
```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds.
Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The
default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
```

```
        - targets:
          # - alertmanager:9093


# Load rules once and periodically evaluate them according to the
global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"


# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any
timeseries scraped from this config.
  - job_name: "ml_model_metrics"


    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.


    static_configs:
      - targets: ["localhost:8001"]
```

2. Run the following command in a terminal window of VScode to test the iris model and expose the metrics.

```
python iris_test.py
```

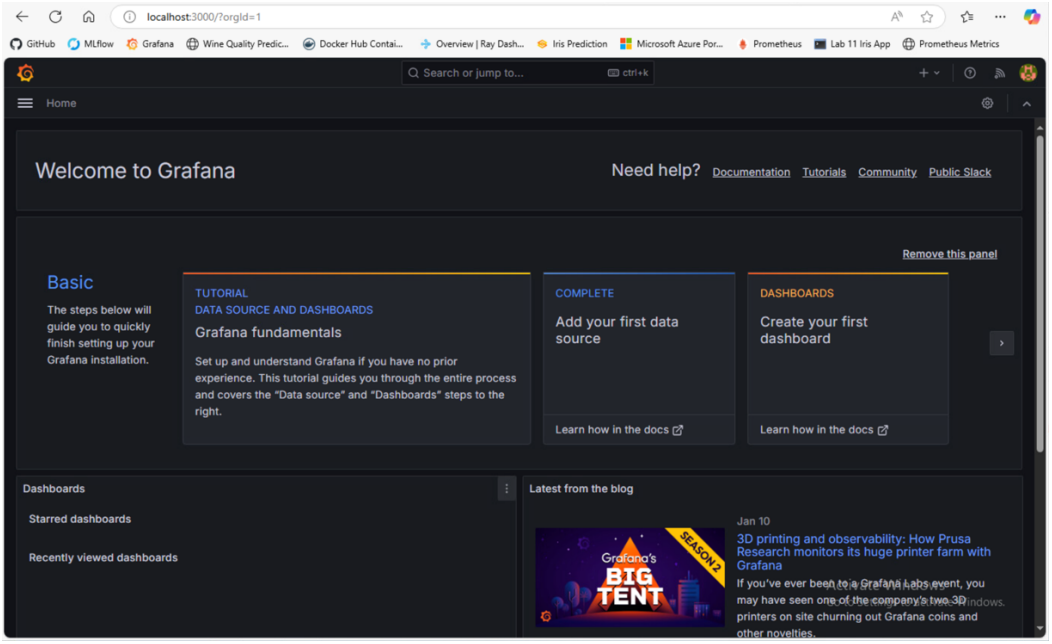
3. Open http://localhost:8001/metrics to view the exposed metrics. (Wait for the script server running message).



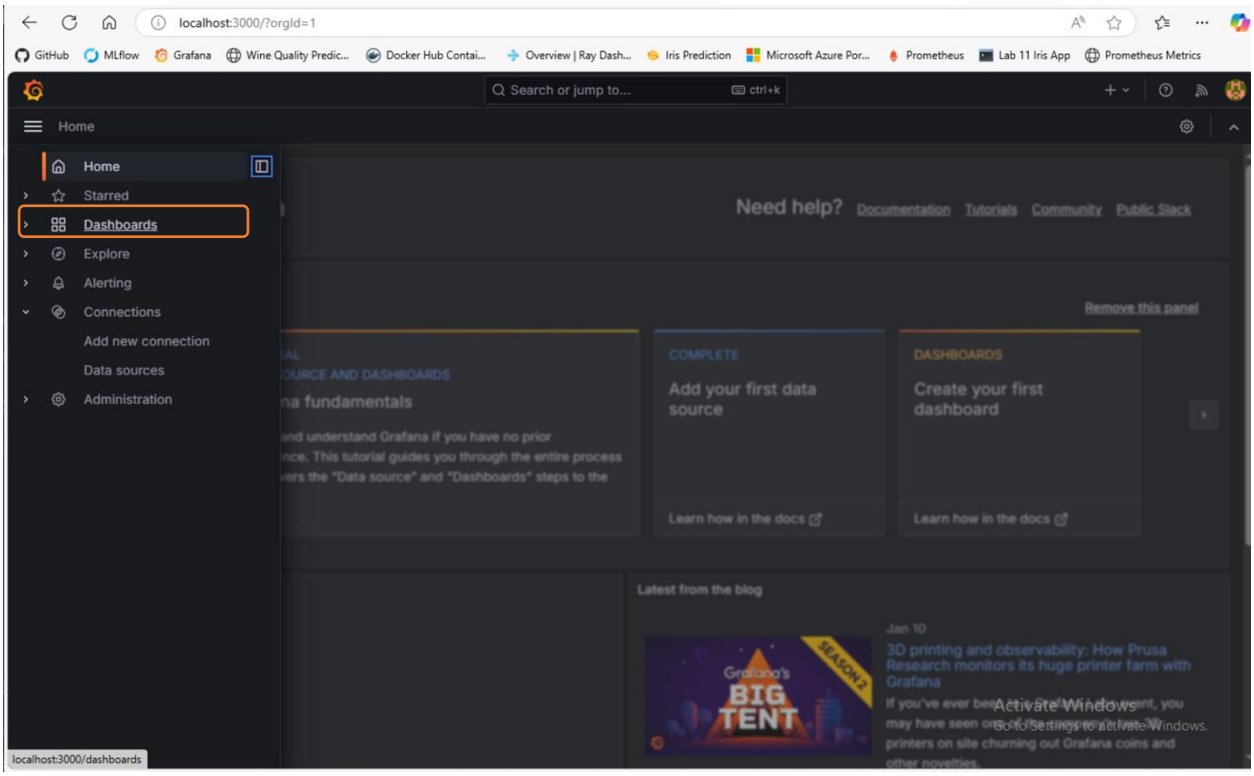
4. Restart Prometheus with this amended configuration.

```
Prometheus.exe --config.file=prometheus.yml
```

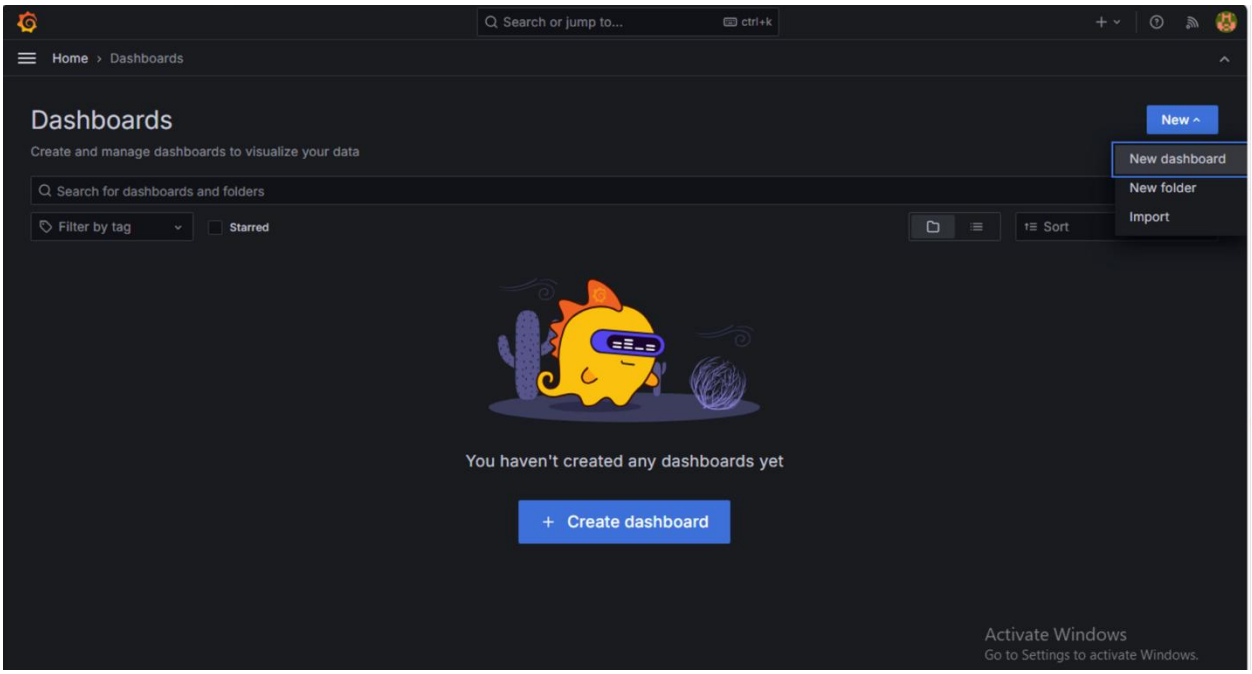
5. Add Prometheus as a data source in Grafana. Open Grafana at **http://localhost:3000**.



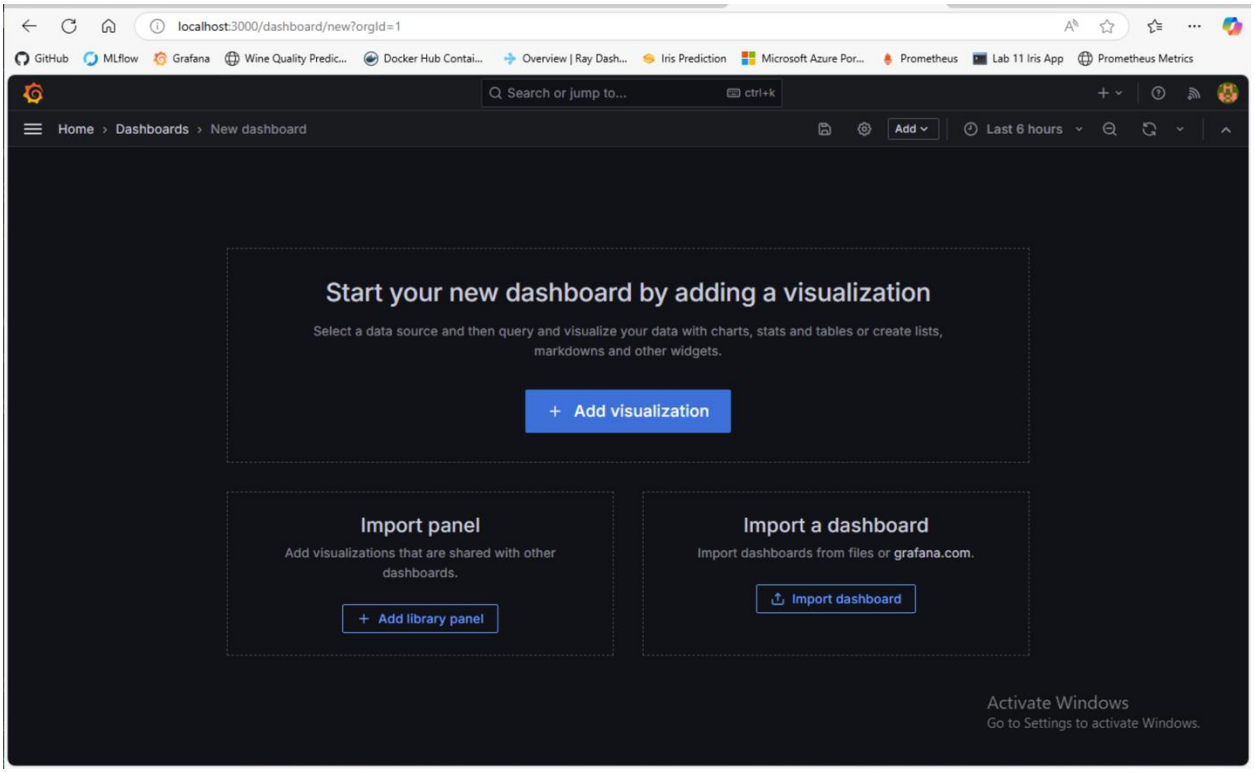
6. Click on top left menu **Home** and then click on the **Dashboards**.



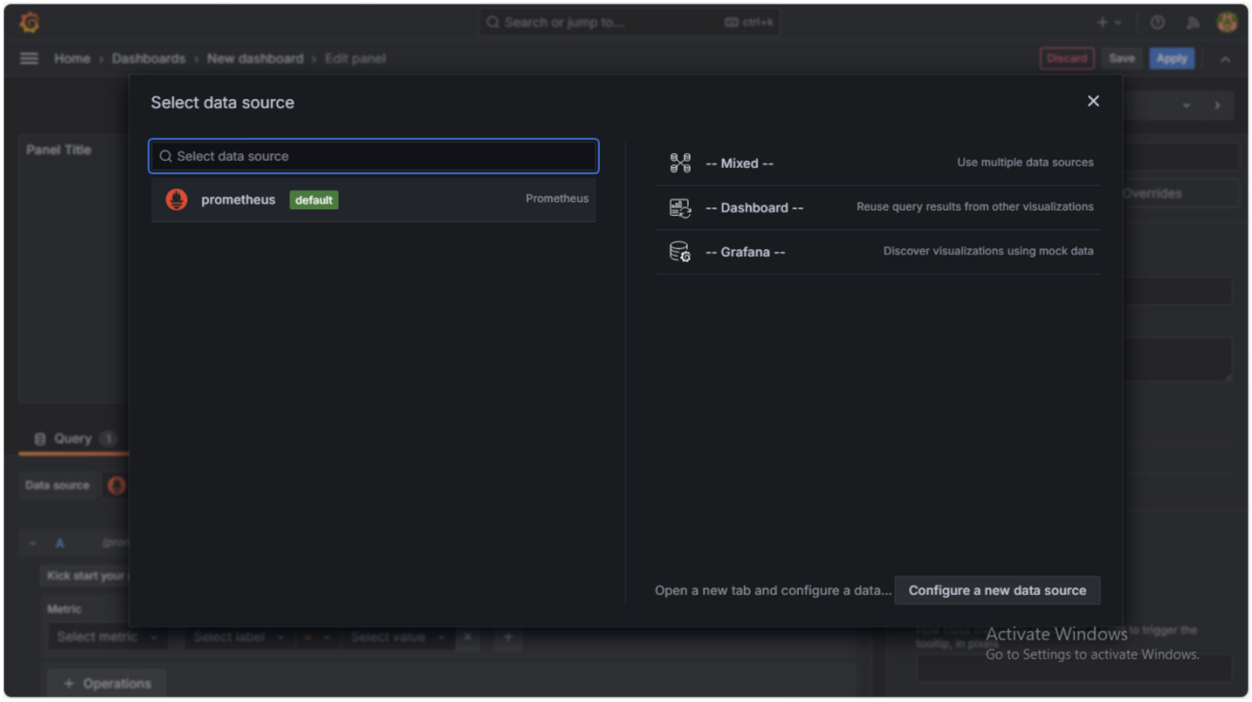
7. In the Dashboards menu, click on the top right blue button **New** and then click on **New Dashboard**.



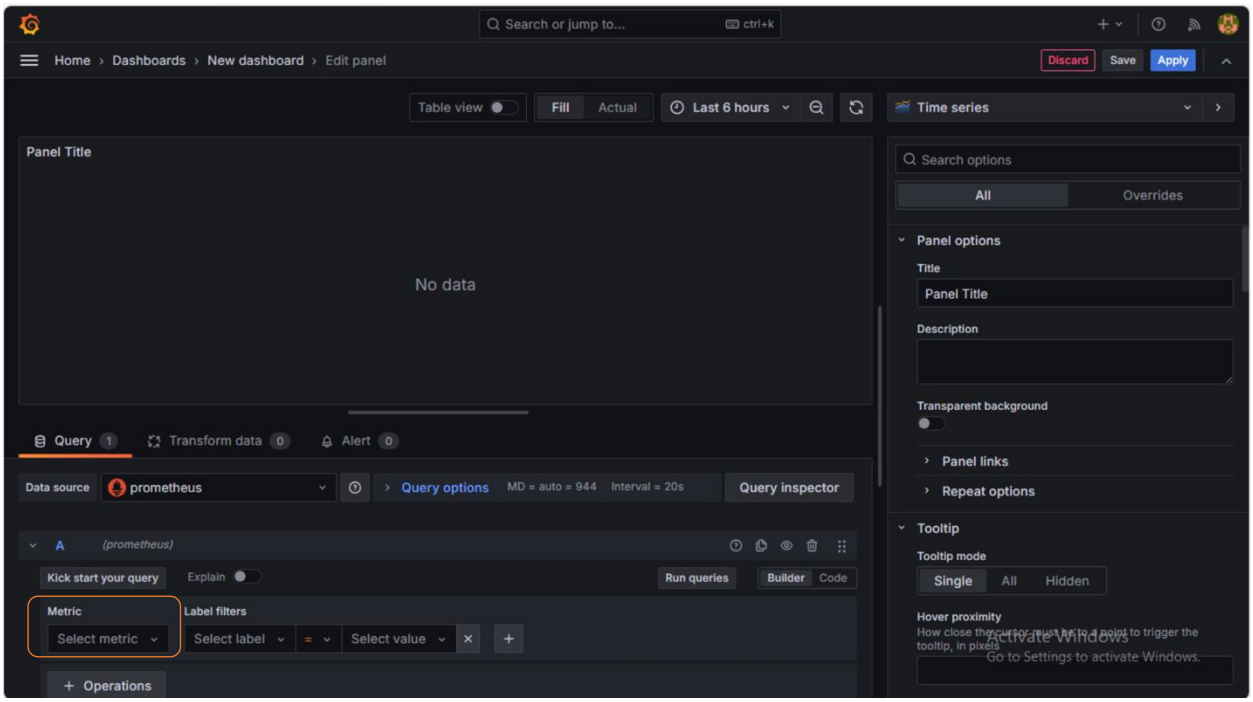
8. To visualize the exposed metrics from the model, click on the **Add Visualization** button.



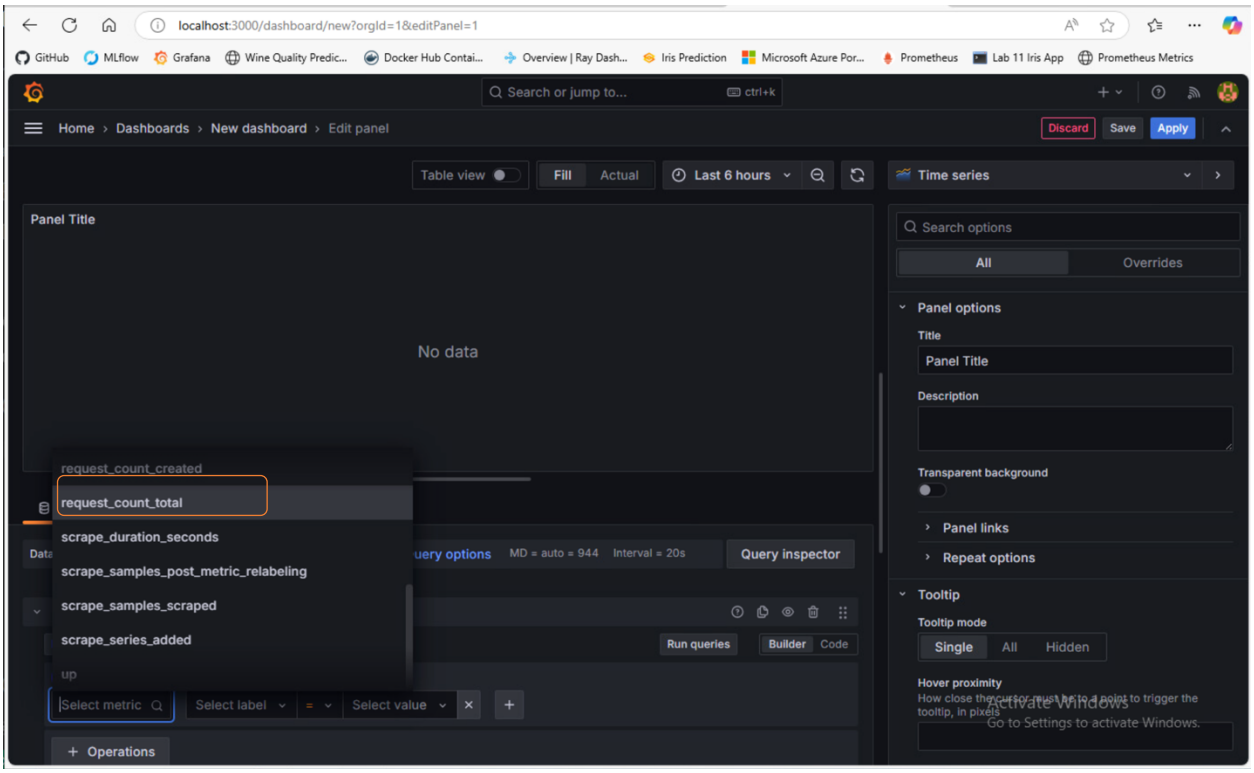
9. Select prometheus as a data source.



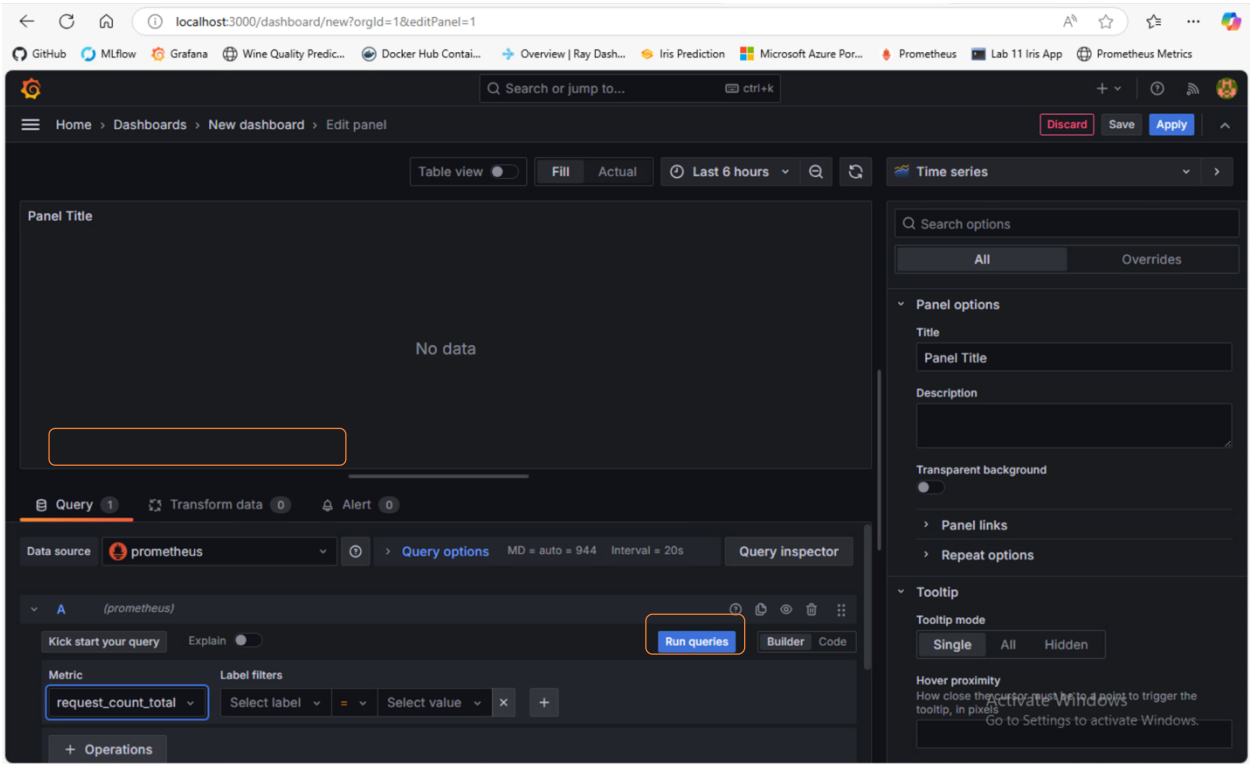
10. Click on the **Metric** menu item to start the query, as shown below.



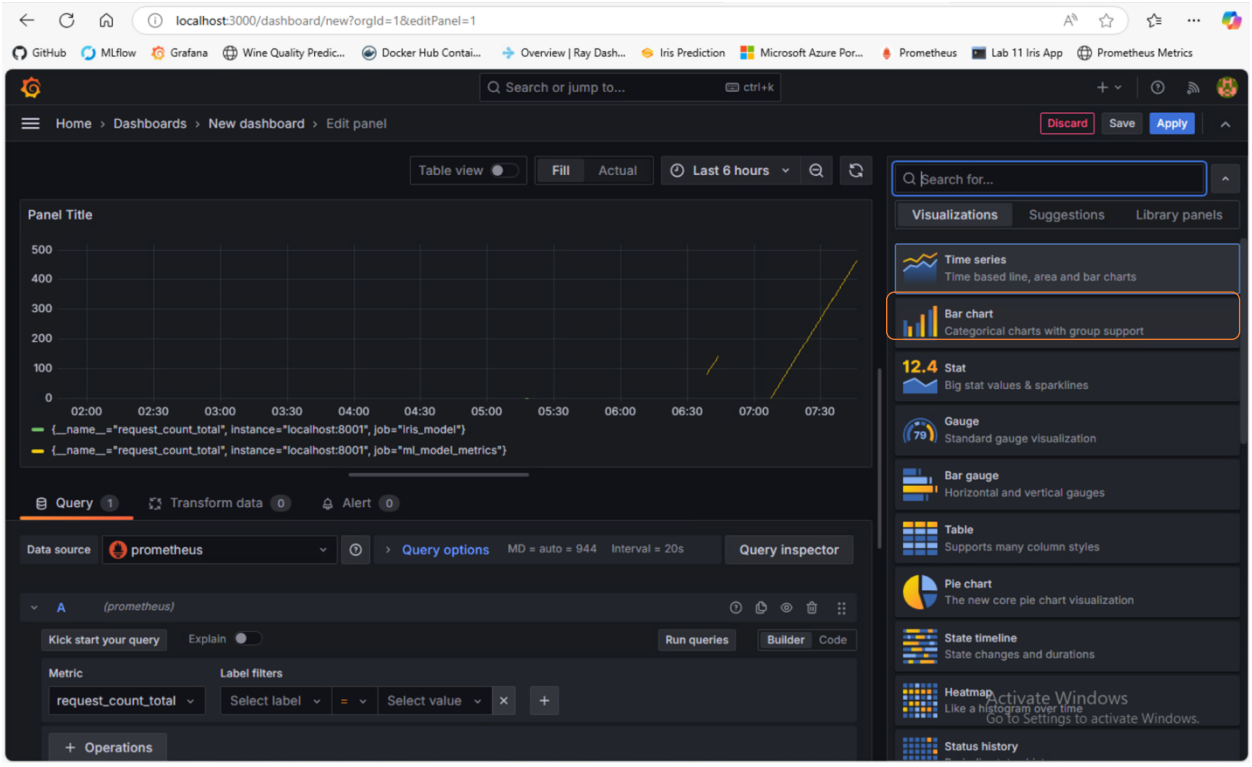
11. Select the **request_count_total** from the Metric menu.



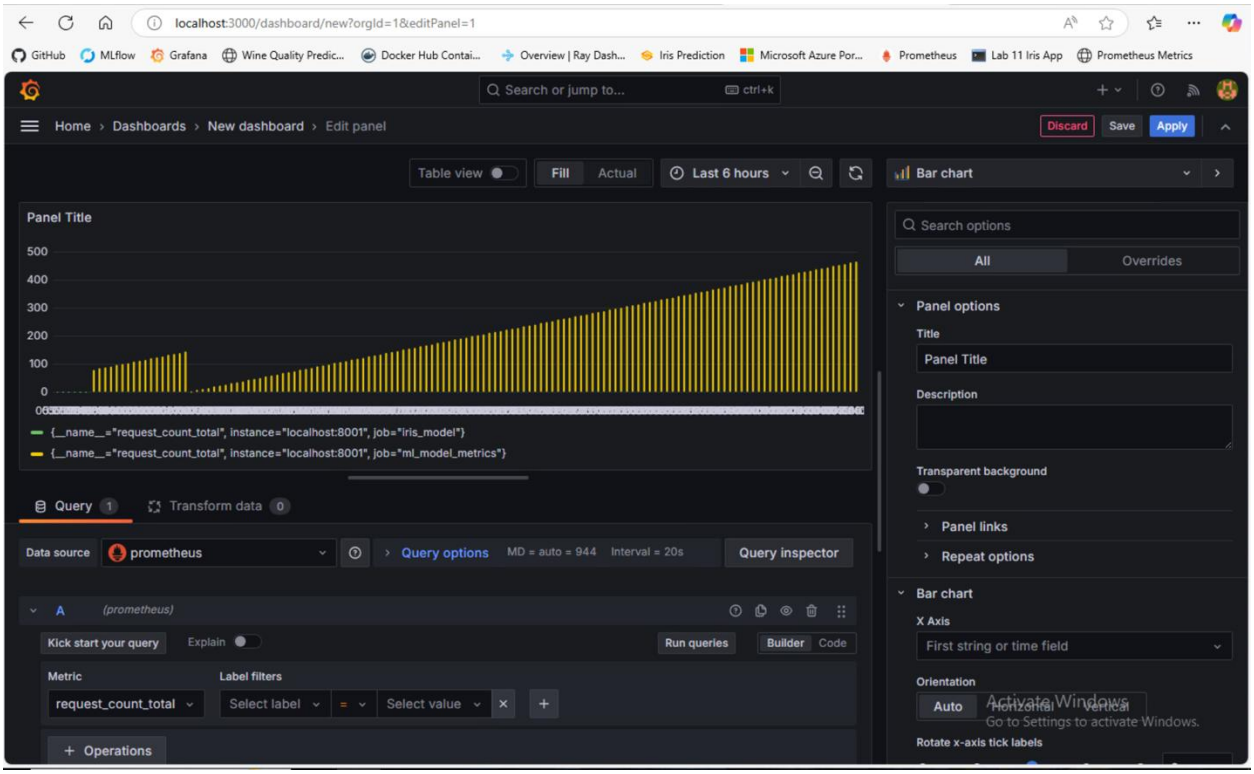
12. Click on the button **Run queries** for visualization.



13. On the right panel select the visualizations type to **Bar chart**.

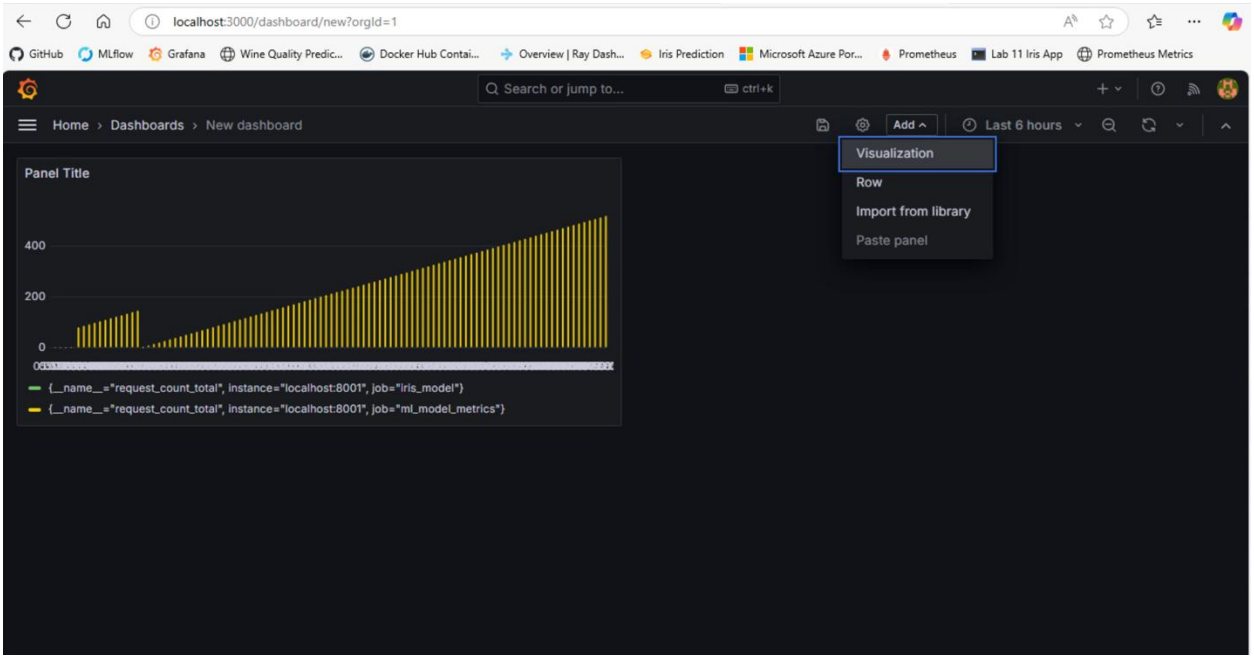


14. On the top right corner click on the button **Apply**.

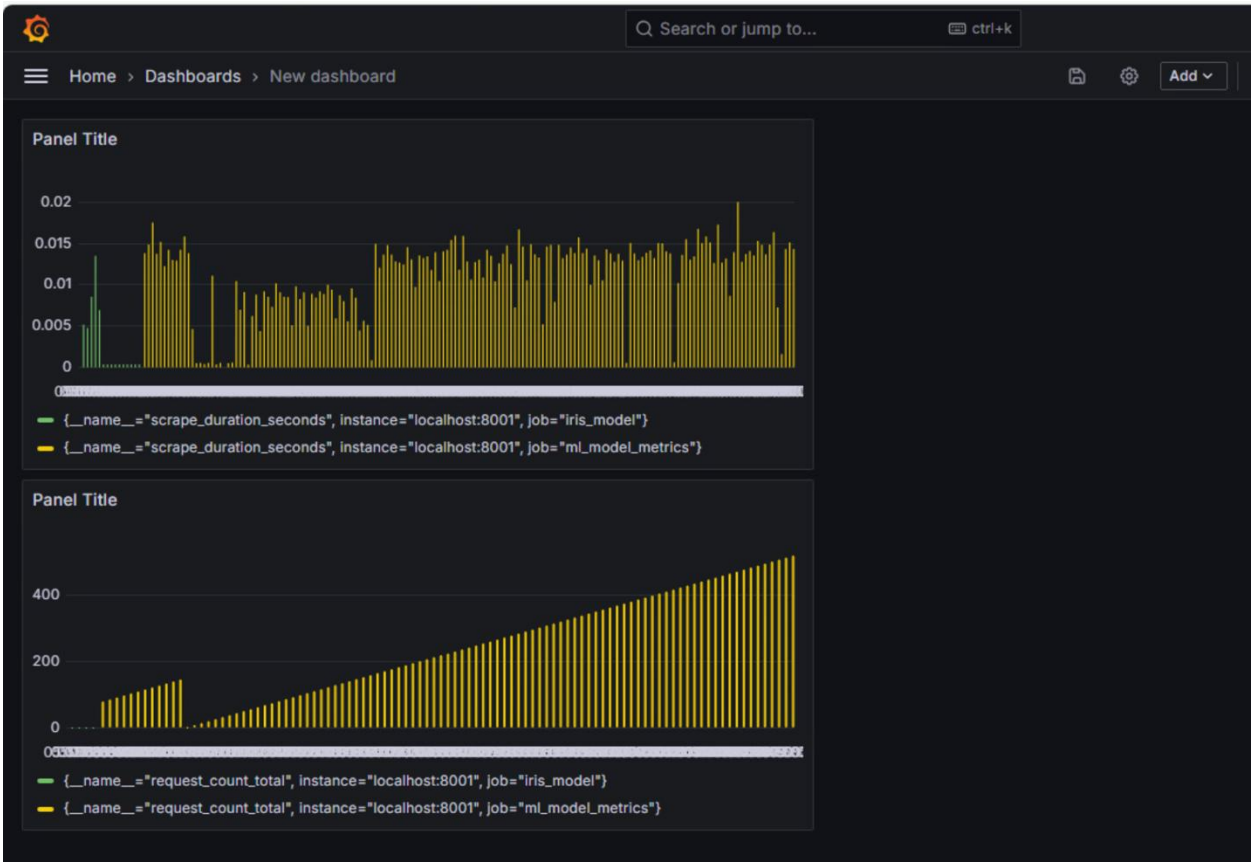


15. This is the final view of the dashboard for the metric values exposed from the test model script in Grafana.

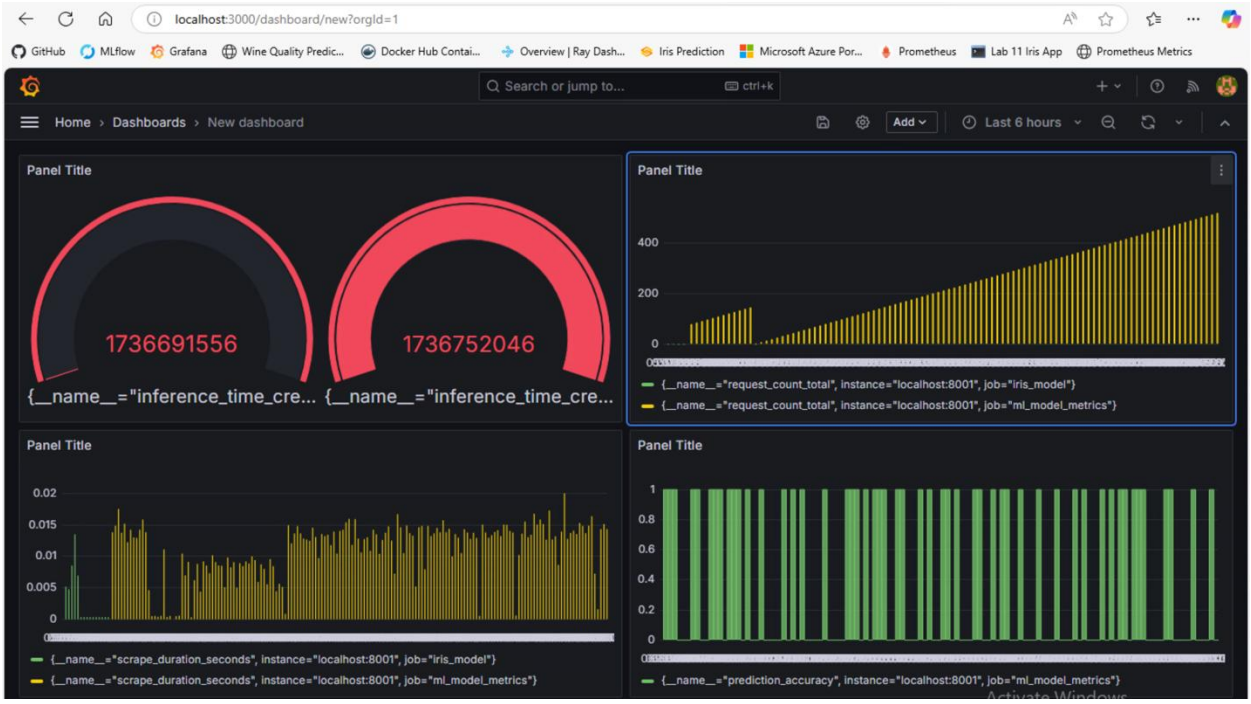
16. To run more queries and add more visualizations click on the **Add** button on the top menu bar and select **Visualization**.



17. Run the next query **scrape_duration_seconds** and add the panel to your dashboard.



18. This way, you can run multiple queries in separate panels, add panels on the dashboard, adjust their positions (drag and drop) to create a beautiful dashboard and learn how your model is performing in the production environment.



19. Configure the Time Interval and Refresh timings to suit (top right of the Grafana window).

Lab review

1. How do you start the Prometheus server after editing the **prometheus.yml** file?
   A. python prometheus.py
   B. prometheus.exe --config.file=prometheus.yml
   C. prometheus start
   D. run_prometheus --start

STOP

You have successfully completed this lab.

Challenge Lab 3: Automate ML Workflows: CI/CD for Wheat Seeds Model with GitHub Actions

Lab overview

In this lab, you will:

- Set up a CI/CD pipeline for the wheat seeds dataset using GitHub Actions
- Train a machine learning model using scikit-learn
- Generate and deploy the model results as an HTML page to GitHub pages

Estimated completion time

45 minutes

Task 1: Setting up the project repository and loading the dataset

This task aims to create a GitHub repository and set up this lab's basic project folder structure.

1. Open your Web browser, log in to GitHub (if necessary), and create a new repository, e.g., wheat-seeds-html-ci-cd add a description, and initialize with a README.
2. Open the Visual Studio Code, open a new terminal window and clone the repository locally, then switch to the folder.
3. Open the project folder in VS Code and launch a new terminal.
4. Now, create a new virtual environment for this task and activate it. Run the following command in the terminal window to create the virtual environment.
5. Once the new virtual environment is set up, install the required libraries in the venv as follows.

```
scikit-learn pandas matplotlib
```

6. Create a necessary folder structure as below.

```
Wheat-seeds-html-ci-cd/
├── Data/
   └── seeds_dataset.txt
├── train_model.py
├── generate_html.py
├── requirements.txt
├── .github/
       └── workflows/
           └── ci.yml
├── index.html  # Will be auto-generated
├── README.md
```

7. Execute the command below in the terminal window to download the dataset in the data folder. A copy has been provided in C:\MLOps\Data-Files, if download is an issue.

```
wget https://archive.ics.uci.edu/ml/machine-learning-
databases/00236/seeds_dataset.txt -P Data/
```

Task 2: Training a model and generating an HTML app

This task aims to process the dataset, train a model, and write a script to automatically generate the HTML file in the project folder which will be used later to show the results.

1. Create a new file with the name **train_model.py** in the project folder and type in the following code.

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score

import json


# Load the Wheat Seeds dataset

column_names = ['Area', 'Perimeter', 'Compactness', 'Length', 'Width',
'Asymmetry', 'Groove', 'Class']

df = pd.read_csv("Data/seeds_dataset.txt", delim_whitespace=True,
header=None, names=column_names)


# Check for any malformed rows and drop them

df = df.dropna()  # Drop rows with missing or malformed data


# Split the data into features and target

X = df.drop('Class', axis=1)

y = df['Class']


# Split into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

```
# Train a Random Forest model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Evaluate the model
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")

# Save results to a JSON file
results = {
    "accuracy": accuracy,
    "feature_importances": list(model.feature_importances_)
}
with open("results.json", "w") as f:
    json.dump(results, f)

print("Model training completed. Results saved to results.json.")
```

This script will train the RandomForest model and save the results (accuracy, feature importances) to a JSON file for later use.

2. Run the **train_model.py** python script in the VS Code terminal.
3. Verify that **results.json** is generated and contains model accuracy and feature importances after executing the above model training script.

4. Create a new Python file in the project folder with the name **generate_html.py**. This script will read the **results.json** file and generate an HTML file (**index.html**) for the app.

```
import json
# Load results from JSON
with open("results.json", "r") as f:
    results = json.load(f)
# Generate HTML
html_content = f"""
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Wheat Seeds Model Results</title>
</head>
<body>
    <h1>Wheat Seeds Model Results</h1>
    <p><strong>Accuracy:</strong> {results['accuracy']:.2f}</p>
    <h2>Feature Importances:</h2>
    <ul>
        <li>Area: {results['feature_importances'][0]:.2f}</li>
        <li>Perimeter: {results['feature_importances'][1]:.2f}</li>
        <li>Compactness: {results['feature_importances'][2]:.2f}</li>
        <li>Length: {results['feature_importances'][3]:.2f}</li>
        <li>Width: {results['feature_importances'][4]:.2f}</li>
        <li>Asymmetry: {results['feature_importances'][5]:.2f}</li>
        <li>Groove: {results['feature_importances'][6]:.2f}</li>
```

```
    </ul>
</body>
</html>
"""
# Save HTML to a file
with open("index.html", "w") as f:
    f.write(html_content)
print("HTML file generated: index.html")
```

5. Run the above Python script, **generate_html.py,** in the VS Code terminal window.
6. Verify that **index.html** is generated with the model results.

Task 3: Automating CI/CD with GitHub actions

This task aims to setup a CI/CD pipeline that pushes the files to GitHub and deploys them on the GitHub pages.

1. In the **.github/workflows** folder, create a file called **ci.yml**, containing the following.

```
name: CI/CD for Hosting Wheat Seeds HTML App

on:
  push:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
```

```
  - name: Checkout code
    uses: actions/checkout@v3

  - name: Set up Python
    uses: actions/setup-python@v3
    with:
      python-version: 3.9

  - name: Install dependencies
    run: |
      python -m venv venv
      source venv/bin/activate
      pip install -r requirements.txt

  - name: Train the model
    run: |
      source venv/bin/activate
      python train_model.py

  - name: Generate HTML
    run: |
      source venv/bin/activate
      python generate_html.py

  - name: Deploy to GitHub Pages
    uses: peaceiris/actions-gh-pages@v4
```

```
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./
```

2. Create a **requirements.txt** file in the project folder with the following dependencies in it.

```
pandas
scikit-learn
matplotlib
```

3. Commit and push changes to the GitHub repository.

```
git add .
git commit -m "Set up CI/CD pipeline for GitHub Pages"
git push origin main
```

4. To enable GitHub pages, go to the GitHub repository, select **Settings** and under pages, set the source to the **Deploy from a branch** and the branch to main /root, then click **Save**.
5. To view the deployed app, click on **Visit site** or navigate to your GitHub page URL (e.g., http://your-username.github.io/wheat-seeds-html-ci-cd/).

Lab review

1. What is the purpose of the **Deploy to GitHub Pages** step in the CI/CD workflow?
   A. To train the machine learning model
   B. To push the model's accuracy and feature importance to GitHub
   C. To publish the index.html file to GitHub Pages
   D. To configure GitHub repository secrets

STOP

You have completed this lab.

Appendix: Lab Review Answer Key

Lab 1: Initialize an ML project with GitHub, virtual environments, and pre-commit hooks

1. What does the command **git push origin main** do?
   A. Stages the changes
   B. Commits the changes locally
   C. Pushes changes to the remote repository
   D. Displays changes in the README.md

> **Correct answer**
>
> C. Pushes changes to the remote repository

Lab 2: EDA, feature engineering, and data cleaning

1. What command is used to save the cleaned dataset as a new CSV file?
   A. titanic_data.save('titanic_cleaned.csv')
   B. titanic_data.to_csv('titanic_cleaned.csv')
   C. titanic_data.write('titanic_cleaned.csv')
   D. titanic_data.export('titanic_cleaned.csv')

> **Correct answer**
>
> B. titanic_data.to_csv('titanic_cleaned.csv')

Lab 3: Building and automating a basic ML pipeline

1. What is the final output of the trained model in this lab?
   A. A .csv file containing predictions
   B. A .pkl file containing the trained pipeline
   C. A .txt file containing model metrics
   D. A .json file containing feature importance

Correct answer

B. A **.pkl** file containing the trained pipeline

Challenge lab 1: Create an end-to-end MLOps pipeline – from preprocessing to GitHub automation with the Iris dataset

1. What is the purpose of the requirements.txt file?
   A. To store project data
   B. To list and manage project dependencies
   C. To configure GitHub Actions workflows
   D. To automate the ML pipeline

Correct answer

B. To list and manage project dependencies

Lab 4: Selecting, implementing, and evaluating algorithms

1. What does the confusion matrix visualize in the evaluation step?
   A. Correlation between features
   B. True positives, false positives, true negatives, and false negatives
   C. Accuracy of the model predictions
   D. Distribution of numerical features

Correct answer

B. True positives, false positives, true negatives, and false negatives

Lab 5: Experiment tracking and model management with MLflow

1. Which command is used to start the MLflow UI?
   A. mlflow start
   B. mlflow run
   C. mlflow ui
   D. mlflow track

Correct answer

C. mlflow ui

Lab 6: Model performance evaluation and comparison

1. What does the ROC curve illustrate in model evaluation?
   A. The relationship between precision and recall
   B. The distribution of false positives and false negatives
   C. The trade-off between a true positive rate and a false positive rate
   D. The feature importance of the model

Correct answer

C. The trade-off between a true positive rate and a false positive rate

Lab 7: Implementing automated hyperparameter tuning

1. Why is hyperparameter tuning important in machine learning?
   A. To reduce the size of the dataset
   B. To increase the computational efficiency of the model
   C. To optimize the model's performance by finding the best configuration
   D. To ensure the model does not overfit

Correct answer

C. To optimize the model's performance by finding the best configuration

Challenge lab 2: Experiment tracking and hyperparameter optimization in MLOps

1. What file is created by MLflow to store experiment data locally?
   A. mlflow_experiments.json
   B. ml_runs.json
   C. mlruns folder
   D. experiment_tracking.db

Correct answer

C. mlruns folder

Lab 8: Dockerize and deploy ML models on cloud platforms

1. Which command is used to build the Docker image for the Flask application?
   A. docker run -p 5000:5000
   B. docker build -t wine-quality-predictor
   C. docker push wine-quality-predictor
   D. docker tag wine-quality-predictor yourusername/wine-quality-predictor

Correct answer

B. docker build -t wine-quality-predictor

Lab 9: Implement CI/CD pipelines for ML with GitHub actions

1. Which command is used to push the data to GitHub?
   A. git add
   B. git push origin main
   C. github push repo main
   D. git commit -m "Push to GitHub"

Correct answer

B. git push origin main

Lab 10: Monitor ML models with Prometheus and Grafana

1. How do you start the Prometheus server after editing the **prometheus.yml** file?
   A. python prometheus.py
   B. prometheus.exe --config.file=prometheus.yml
   C. prometheus start
   D. run_prometheus --start

Correct answer

B. prometheus.exe --config.file=prometheus.yml

Challenge Lab 3: Automate ML workflows: CI/CD for wheat seeds model with GitHub actions

1. What is the purpose of the **Deploy to GitHub Pages** step in the CI/CD workflow?
   A. To train the machine learning model
   B. To push the model's accuracy and feature importance to GitHub
   C. **To publish the index.html file to GitHub Pages**
   D. To configure GitHub repository secrets

> **Correct answer**
>
> C. To publish the index.html file to GitHub Pages.