**Project Title: Wine Classification with Machine Learning**

Overview:

This project aims to develop machine learning models to classify wine types based on their chemical attributes. The dataset used is the Wine dataset from the UCI Machine Learning Repository, consisting of 178 samples with 13 features each.

Week 1 Tasks:

1. **Team Formation**: Groups of three students were formed.
2. **Names**: Julius Fawaz Kamara Abdul Ernest Sesay, Silas Max-Dixon
3. **Dataset Exploration**:
   * Loaded the Wine dataset.
   * Explored dataset description, feature names, and target names.
4. **Data Preprocessing**:
   * Converted the dataset to a DataFrame.
   * Checked for missing values (No missing values found).
   * Split the dataset into features and target labels.
   * Performed feature scaling using StandardScaler.
5. **Train-Test Split**:
   * Split the dataset into training and testing sets (80-20 split).

Week 2 Tasks:

1. **Model 1: Logistic Regression**:
   * Trained a Logistic Regression model.
2. **Model 2: Decision Trees**:
   * Trained a Decision Tree classifier.
3. **Model 3: Support Vector Machines (SVM)**:
   * Trained an SVM classifier.
4. **Model Evaluation**:
   * Defined a function to evaluate models based on accuracy, precision, recall, F1 score, and confusion matrix.
   * Evaluated all three models.
   * Printed evaluation results for each model.
5. **Visualization**:
   * Defined a function to plot confusion matrices.
   * Visualized confusion matrices for each model.
6. **Results Analysis**:

**Model 1: Logistic Regression**

Analysis:

* **Code**:
  + Logistic Regression model was trained using **LogisticRegression** from scikit-learn.
  + Evaluation metrics such as accuracy, precision, recall, and F1 score were calculated using **accuracy\_score**, **precision\_score**, **recall\_score**, and **f1\_score**.
  + Confusion matrix was generated using **confusion\_matrix**.
* **Output Analysis**:
  + The accuracy of the Logistic Regression model was calculated to be around 0.9722, indicating that it correctly classified approximately 97.22% of the instances in the test set.
  + Precision score of around 0.9742 suggests that the model correctly predicted about 97.42% of positive instances out of total predicted positives.
  + Recall score of approximately 0.9722 indicates that the model correctly predicted about 97.22% of positive instances out of all actual positives.
  + F1 score, being the harmonic mean of precision and recall, provides a balance between the two metrics and is around 0.9721, indicating overall good performance.
* **Insights**:
  + The Logistic Regression model achieved high accuracy, precision, recall, and F1 score, indicating its effectiveness in classifying wine types.
  + The confusion matrix provides a detailed breakdown of correct and incorrect predictions, facilitating further analysis and insights into model performance.

**Model 2: Decision Trees**

Analysis:

* **Code**:
  + Decision Tree model was trained using **DecisionTreeClassifier** from scikit-learn.
  + Similar evaluation metrics and confusion matrix were calculated as in Logistic Regression.
* **Output Analysis**:
  + The accuracy of the Decision Tree model was observed to be around 0.9167, which is slightly lower compared to Logistic Regression.
  + Precision, recall, and F1 score were also slightly lower compared to Logistic Regression, indicating that Decision Trees may not generalize as well as Logistic Regression on this dataset.
  + The confusion matrix reveals which classes are being confused and in what proportion, providing insights into model weaknesses and strengths.
* **Insights**:
  + Decision Trees, while interpretable, may not generalize as well as Logistic Regression on this dataset, possibly due to overfitting.
  + Further analysis, such as pruning techniques, could be explored to improve the generalization ability of the Decision Tree model.

**Model 3: Support Vector Machines (SVM)**

Analysis:

* **Code**:
  + SVM model was trained using **SVC** from scikit-learn.
  + Similar evaluation metrics and confusion matrix were calculated as in previous models.
* **Output Analysis**:
  + The accuracy of the SVM model was observed to be around 0.9722, similar to Logistic Regression, indicating high performance.
  + Precision, recall, and F1 score were also comparable to Logistic Regression, suggesting that SVM performed well on this dataset.
  + The confusion matrix provides insights into how well the SVM model classified instances into different classes.
* **Insights**:
  + SVMs offer robust performance and can handle complex decision boundaries effectively.
  + The model achieved competitive results compared to Logistic Regression, indicating its effectiveness in classifying wine types.

**Conclusion:**

* Analyzing the results of all three models provides insights into their performance and suitability for the wine classification task.
* While Logistic Regression and SVM achieved high accuracy and performance, Decision Trees showed slightly lower performance, possibly due to overfitting.
* Understanding the strengths and weaknesses of each model helps in selecting the most appropriate approach for the given dataset and problem domain, facilitating informed decision-making for future improvements and optimizations.

Top of Form

* + Analyzed and interpreted model evaluation results.

1. **Documentation and Presentation**:
   * Prepared documentation in Jupyter Notebook format with code explanations, results analysis, and conclusion.
   * Presented key findings, challenges, and approach in a presentation format.
2. **Further Steps**:

Exploratory Data Analysis (EDA):

* Pairplot and correlation heatmap were generated to visualize feature distributions and correlations.
* Countplot was created to visualize target distribution.

Insights:

* Models achieved decent performance, with Logistic Regression having the highest accuracy.
* Decision Trees showed a tendency to overfit the data.
* SVM performed reasonably well but took longer to train compared to other models.
* EDA revealed some correlations among features, which could influence model performance.
* The provided code includes all necessary imports, data loading, preprocessing, model training, evaluation, and visualization.
* Code snippets are properly commented for clarity and understanding.
* **CODE:** **# Import necessary libraries**
* **import pandas as pd**
* **import matplotlib.pyplot as plt**
* **import seaborn as sns**
* **from sklearn.datasets import load\_wine**
* **from sklearn.preprocessing import StandardScaler**
* **from sklearn.model\_selection import train\_test\_split**
* **from sklearn.linear\_model import LogisticRegression**
* **from sklearn.tree import DecisionTreeClassifier**
* **from sklearn.svm import SVC**
* **from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix**
* **# Week 1 Tasks**
* **# 1. Team Formation: Form groups of three students.**
* **# 2. Dataset Exploration:**
* **# Load the dataset**
* **wine\_data = load\_wine()**
* **# Explore the dataset**
* **print(wine\_data.DESCR)**
* **print(wine\_data.feature\_names)**
* **print(wine\_data.target\_names)**
* **# 3. Data Preprocessing:**
* **# Convert dataset to DataFrame**
* **wine\_df = pd.DataFrame(data=wine\_data.data, columns=wine\_data.feature\_names)**
* **wine\_df['target'] = wine\_data.target**
* **# Check for missing values**
* **print(wine\_df.isnull().sum())**
* **# Split the dataset into features (X) and target labels (y)**
* **X = wine\_df.drop('target', axis=1)**
* **y = wine\_df['target']**
* **# Perform scaling**
* **scaler = StandardScaler()**
* **X\_scaled = scaler.fit\_transform(X)**
* **# 4. Train-Test Split:**
* **X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)**
* **# 5. Model 1: Logistic Regression**
* **log\_reg\_model = LogisticRegression(max\_iter=1000)**
* **log\_reg\_model.fit(X\_train, y\_train)**
* **# Week 2 Tasks**
* **# 6. Model 2: Decision Trees**
* **dt\_model = DecisionTreeClassifier()**
* **dt\_model.fit(X\_train, y\_train)**
* **# 7. Model 3: Support Vector Machines (SVM)**
* **svm\_model = SVC()**
* **svm\_model.fit(X\_train, y\_train)**
* **# 8. Model Evaluation:**
* **# Function to evaluate models**
* **def evaluate\_model(model, X\_test, y\_test):**
* **y\_pred = model.predict(X\_test)**
* **acc = accuracy\_score(y\_test, y\_pred)**
* **prec = precision\_score(y\_test, y\_pred, average='weighted')**
* **rec = recall\_score(y\_test, y\_pred, average='weighted')**
* **f1 = f1\_score(y\_test, y\_pred, average='weighted')**
* **cm = confusion\_matrix(y\_test, y\_pred)**
* **return acc, prec, rec, f1, cm**
* **# Evaluate all three models**
* **log\_reg\_results = evaluate\_model(log\_reg\_model, X\_test, y\_test)**
* **dt\_results = evaluate\_model(dt\_model, X\_test, y\_test)**
* **svm\_results = evaluate\_model(svm\_model, X\_test, y\_test)**
* **# Print results**
* **print("Logistic Regression Results:")**
* **print("Accuracy:", log\_reg\_results[0])**
* **print("Precision:", log\_reg\_results[1])**
* **print("Recall:", log\_reg\_results[2])**
* **print("F1 Score:", log\_reg\_results[3])**
* **print("Confusion Matrix:\n", log\_reg\_results[4])**
* **print("Decision Tree Results:")**
* **print("Accuracy:", dt\_results[0])**
* **print("Precision:", dt\_results[1])**
* **print("Recall:", dt\_results[2])**
* **print("F1 Score:", dt\_results[3])**
* **print("Confusion Matrix:\n", dt\_results[4])**
* **print("SVM Results:")**
* **print("Accuracy:", svm\_results[0])**
* **print("Precision:", svm\_results[1])**
* **print("Recall:", svm\_results[2])**
* **print("F1 Score:", svm\_results[3])**
* **print("Confusion Matrix:\n", svm\_results[4])**
* **# 9. Visualization:**
* **# Function to plot confusion matrix**
* **def plot\_confusion\_matrix(cm, target\_names):**
* **plt.figure(figsize=(8, 6))**
* **sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=target\_names, yticklabels=target\_names)**
* **plt.xlabel('Predicted Labels')**
* **plt.ylabel('True Labels')**
* **plt.title('Confusion Matrix')**
* **plt.show()**
* **# Visualize confusion matrix for Logistic Regression**
* **plot\_confusion\_matrix(log\_reg\_results[4], wine\_data.target\_names)**
* **# Visualize confusion matrix for Decision Tree**
* **plot\_confusion\_matrix(dt\_results[4], wine\_data.target\_names)**
* **# Visualize confusion matrix for SVM**
* **plot\_confusion\_matrix(svm\_results[4], wine\_data.target\_names)**
* **# 10. Results Analysis:**
* **# Results analysis is already done within the print statements after model evaluation.**
* **# 11. Documentation and Presentation:**
* **# Prepare documentation in a Jupyter Notebook or Python script format.**
* **# Include an introduction, dataset exploration, code explanations, results analysis, and conclusion.**
* **# Prepare a presentation summarizing the key findings, challenges, and the team's approach.**
* **# 12. Further Steps:**
* **# The code can be extended to include hyperparameter tuning and further analysis as mentioned in the project objectives.**
* **# Exploratory Data Analysis (EDA):**
* **# Pairplot for feature visualization**
* **sns.pairplot(wine\_df, hue='target')**
* **plt.title('Pairplot of Wine Dataset Features')**
* **plt.show()**
* **# Correlation heatmap**
* **plt.figure(figsize=(10, 8))**
* **sns.heatmap(wine\_df.corr(), annot=True, cmap='coolwarm')**
* **plt.title('Correlation Heatmap of Wine Dataset Features')**
* **plt.show()**
* **# Countplot for target distribution**
* **plt.figure(figsize=(6, 4))**
* **sns.countplot(x='target', data=wine\_df)**
* **plt.title('Target Distribution')**
* **plt.xlabel('Target')**
* **plt.ylabel('Count')**
* **plt.show()**
* **# Generate or collect new data instances**
* **# new instances**
* **new\_instances = [**
* **[13.24, 2.59, 2.87, 21.0, 118.0, 2.8, 2.69, 0.39, 1.82, 4.32, 1.04, 2.93, 735.0, 0],**
* **[12.07, 2.16, 2.17, 21.0, 85.0, 2.6, 2.65, 0.37, 1.35, 2.76, 0.86, 3.28, 378.0, 1],**
* **# Add more instances as needed**
* **]**
* **# Convert the new instances to a DataFrame**
* **new\_instances\_df = pd.DataFrame(new\_instances, columns=wine\_data.feature\_names + ['target'])**
* **# Append new instances to the existing DataFrame**
* **wine\_df\_updated = pd.concat([wine\_df, new\_instances\_df], ignore\_index=True)**
* **# Verify the new instances have been added**
* **print(wine\_df\_updated.tail())**
* **# Split the updated dataset into features (X) and target labels (y)**
* **X\_updated = wine\_df\_updated.drop('target', axis=1)**
* **y\_updated = wine\_df\_updated['target']**
* **# Perform scaling on the updated features**
* **scaler\_updated = StandardScaler()**
* **X\_scaled\_updated = scaler\_updated.fit\_transform(X\_updated)**
* **# Split the updated scaled data into train and test sets**
* **X\_train\_updated, X\_test\_updated, y\_train\_updated, y\_test\_updated = train\_test\_split(X\_scaled\_updated, y\_updated, test\_size=0.2, random\_state=42)**
* **# Now, you can proceed with retraining and evaluating the models using the updated dataset.**
* **# Visualize the data with scatter plots for each pair of features**
* **num\_features = len(wine\_df\_updated.columns) - 1 # Exclude the target column**
* **for i in range(num\_features):**
* **for j in range(i+1, num\_features): # Loop starts from i+1 to avoid duplicate pairs and plots on diagonal**
* **plt.figure(figsize=(8, 6))**
* **for target\_class in wine\_df\_updated['target'].unique():**
* **plt.scatter(wine\_df\_updated[wine\_df\_updated['target'] == target\_class].iloc[:, i],**
* **wine\_df\_updated[wine\_df\_updated['target'] == target\_class].iloc[:, j],**
* **label=f'Class {target\_class}')**
* **plt.scatter(new\_instances\_df.iloc[:, i], new\_instances\_df.iloc[:, j], color='black', marker='x', label='New Instances')**
* **plt.xlabel(wine\_df\_updated.columns[i])**
* **plt.ylabel(wine\_df\_updated.columns[j])**
* **plt.title(f'{wine\_df\_updated.columns[i]} vs {wine\_df\_updated.columns[j]}')**
* **plt.legend()**
* **plt.grid(True)**
* **plt.show()**

**important information on understanding the code**

**Import necessary libraries**

import pandas as pd (1): This line imports the Pandas library, which is used for data manipulation and analysis. import matplotlib.pyplot as plt

(2): This line imports the Pyplot module from Matplotlib library, which is used for data visualization. import seaborn as sns

(3): This line imports the Seaborn library, which is used for statistical data visualization. from sklearn.datasets import load\_wine

(4): This line imports the load\_wine function from the sklearn.datasets module, which is used to load the wine dataset. from sklearn.preprocessing import StandardScaler

(5): This line imports the StandardScaler class from the sklearn.preprocessing module, which is used for feature scaling. from sklearn.model\_selection import train\_test\_split

(6): This line imports the train\_test\_split function from the sklearn.model\_selection module, which is used to split the dataset into training and testing sets. from sklearn.linear\_model import LogisticRegression

(7): This line imports the LogisticRegression class from the sklearn.linear\_model module, which is used to create a logistic regression model. from sklearn.tree import DecisionTreeClassifier

(8): This line imports the DecisionTreeClassifier class from the sklearn.tree module, which is used to create a decision tree model. from sklearn.svm import SVC

(9): This line imports the SVC class from the sklearn.svm module, which is used to create a support vector machine model. from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix (10): These lines import the evaluation metrics used to assess the performance of the models.

The code snippet imports the load\_wine function from the sklearn.datasets module. This function is part of the scikit-learn library in Python and is used to load a sample dataset called the Wine dataset. The Wine dataset contains the results of a chemical analysis of wines from three different cultivars grown in the same region in Italy. It consists of 13 different measurements for each wine, such as alcohol content, malic acid content, and ash content. The goal of using this dataset is typically to classify wines into one of three classes corresponding to the three different cultivars.

After loading the dataset using load\_wine(), you can access the features (attributes), target variable (class labels), and feature names using wine\_data.data, wine\_data.target, and wine\_data.feature\_names, respectively. This allows for further exploration and analysis of the dataset.

**Load the dataset**

wine\_data = load\_wine()

The provided code snippet loads the Wine dataset using the load\_wine() function from the sklearn.datasets module in Python. Here's a breakdown of what this code does:

load\_wine(): This function is part of scikit-learn (sklearn), a popular machine learning library in Python. It loads the Wine dataset, which is one of the sample datasets provided by scikit-learn for practicing and learning machine learning techniques.

wine\_data = load\_wine(): The result of calling load\_wine() is assigned to a variable named wine\_data. This variable now holds a dictionary-like object containing the Wine dataset.

After executing this code, the wine\_data variable contains the following attributes:

data: A NumPy array containing the features (attributes) of the Wine dataset. Each row corresponds to a different wine sample, and each column corresponds to a different feature (e.g., alcohol content, malic acid content, etc.).

target: A NumPy array containing the target variable (class labels) of the Wine dataset. Each element in this array represents the class label of the corresponding wine sample.

feature\_names: A list containing the names of the features (attributes) in the Wine dataset. These names correspond to the columns in the data array.

target\_names: A list containing the names of the target classes in the Wine dataset. These names correspond to the unique values present in the target array.

By loading the Wine dataset using load\_wine(), you can now access and analyze its features, target variable, and other relevant information to perform tasks such as data preprocessing, exploratory data analysis, and building machine learning models for classification or regression tasks.

**Explore the dataset**

print(wine\_data.DESCR) print(wine\_data.feature\_names) print(wine\_data.target\_names)

The provided code snippet explores the Wine dataset by printing out various information about its contents.

print(wine\_data.DESCR): This line prints a detailed description of the Wine dataset. The description typically includes information about the dataset's source, number of samples, number of features, a summary of each feature's statistics, and any relevant background information.

print(wine\_data.feature\_names): This line prints the names of the features (attributes) present in the Wine dataset. Each feature name corresponds to a specific characteristic or measurement associated with each wine sample.

print(wine\_data.target\_names): This line prints the names of the target classes in the Wine dataset. In classification tasks, the target variable represents the labels or categories that we want to predict. The target\_names attribute provides the names corresponding to each unique label in the target variable.

In summary, the provided code snippet helps explore the Wine dataset by providing:

A detailed description of the dataset's contents and characteristics.

The names of the features, which represent different measurements associated with each wine sample.

The names of the target classes, which represent the categories we want to predict in classification tasks. These names provide insight into the types of wines present in the dataset.

**Data Preprocessing:**

import pandas as pd

The provided code snippet imports the pandas library in Python using the alias pd. Here's a breakdown of what this code does:

import pandas as pd: This line imports the pandas library, which is a powerful tool for data manipulation and analysis in Python. By using the as pd syntax, the library is aliased as pd, making it easier to refer to pandas functions and objects throughout the code.

Data preprocessing is a crucial step in the machine learning pipeline, involving tasks such as handling missing values, encoding categorical variables, scaling numerical features, and splitting the dataset into training and testing sets. While the provided code snippet doesn't perform any specific data preprocessing tasks, importing the pandas library suggests that subsequent code may involve data manipulation using pandas DataFrame objects, which are widely used for preprocessing and analyzing structured data in Python.

**Convert dataset to DataFrame**

wine\_df = pd.DataFrame(data=wine\_data.data, columns=wine\_data.feature\_names) wine\_df['target'] = wine\_data.

The provided code snippet converts the Wine dataset into a pandas DataFrame and adds a new column named 'target' to the DataFrame. Here's a breakdown of what this code does:

wine\_df = pd.DataFrame(data=wine\_data.data, columns=wine\_data.feature\_names): This line creates a pandas DataFrame named wine\_df using the pd.DataFrame() constructor. The data parameter is set to wine\_data.data, which contains the features (attributes) of the Wine dataset. The columns parameter is set to wine\_data.feature\_names, which provides the names of the features. This constructs a DataFrame where each row represents a wine sample, and each column represents a specific feature associated with the wine.

wine\_df['target'] = wine\_data.target: This line adds a new column named 'target' to the DataFrame wine\_df. The values in this column are taken from the wine\_data.target array, which contains the target variable (class labels) of the Wine dataset. Each value in the 'target' column corresponds to the class label of the corresponding wine sample in the DataFrame.

In summary, the provided code snippet converts the Wine dataset into a pandas DataFrame, making it easier to manipulate and analyze the data using pandas functions and tools. Additionally, it adds a new column to the DataFrame to include the target variable, which is essential for supervised learning tasks such as classification.

**Check for missing values**

print(wine\_df.isnull().sum

The provided code snippet checks for missing values in the DataFrame wine\_df and prints out the sum of missing values for each column. Here's a breakdown of what this code does:

wine\_df.isnull(): This part of the code generates a boolean DataFrame where each element is True if the corresponding value in wine\_df is missing (i.e., NaN), and False otherwise.

.sum(): This part of the code calculates the sum of missing values along each column of the boolean DataFrame generated in the previous step.

print(): This function is used to display the result of the operation inside the parentheses.

In summary, the provided code snippet computes the total number of missing values for each column in the DataFrame wine\_df and prints out the result. This information is helpful for identifying and addressing any missing data in the dataset during the data preprocessing phase.

**Split the dataset into features (X) and target labels (y)**

X = wine\_df.drop('target', axis=1) y = wine\_df['target'

The provided code snippet splits the Wine dataset stored in the DataFrame wine\_df into features (X) and target labels (y). Here's a breakdown of what this code does:

X = wine\_df.drop('target', axis=1): This line creates a new DataFrame X by dropping the 'target' column from the original DataFrame wine\_df. The drop() function is used with the axis=1 parameter, which specifies that we want to drop columns (not rows). As a result, X contains all the columns/features of the Wine dataset except for the 'target' column.

y = wine\_df['target']: This line creates a new Series y containing the values of the 'target' column from the original DataFrame wine\_df. This column represents the target labels or class labels associated with each wine sample.

In summary, the provided code snippet separates the features (X) from the target labels (y) in the Wine dataset, allowing for further analysis and model building. The features (X) represent the input variables used to predict the target labels (y), which are the output variables we want our model to predict.

**Perform scaling**

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler() X\_scaled = scaler.fit\_transform(X

The provided code snippet performs feature scaling on the features (X) of the Wine dataset using the StandardScaler from scikit-learn. Here's a breakdown of what this code does:

from sklearn.preprocessing import StandardScaler: This line imports the StandardScaler class from the sklearn.preprocessing module. The StandardScaler is a preprocessing technique used to standardize features by removing the mean and scaling them to unit variance.

scaler = StandardScaler(): This line creates an instance of the StandardScaler class, which will be used to scale the features.

X\_scaled = scaler.fit\_transform(X): This line applies the scaling transformation to the features (X) using the fit\_transform() method of the StandardScaler object. The fit\_transform() method computes the mean and standard deviation of each feature in X and then standardizes the features based on these statistics. The standardized features are stored in the variable X\_scaled.

In summary, the provided code snippet standardizes the features of the Wine dataset using the StandardScaler, ensuring that each feature has a mean of 0 and a standard deviation of 1. This preprocessing step is commonly performed in machine learning to ensure that features are on a similar scale, which can improve the performance of certain algorithms and make the optimization process more efficient.

**TRAIN-TEST SPLIT**:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

The code snippet splits the standardized features (X\_scaled) and target labels (y) into training and testing sets using the train\_test\_split function from scikit-learn. Here's a breakdown of what this code does:

from sklearn.model\_selection import train\_test\_split: This line imports the train\_test\_split function from the sklearn.model\_selection module. This function is used to split datasets into random train and test subsets.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42): This line splits the standardized features (X\_scaled) and target labels (y) into training and testing sets. The parameters passed to the train\_test\_split function are as follows:

X\_scaled: The standardized features.

y: The target labels.

test\_size=0.2: The proportion of the dataset to include in the test split. Here, 20% of the data is reserved for testing, while the remaining 80% is used for training.

random\_state=42: This parameter sets the random seed for reproducibility. By setting a specific random\_state, the split will always be the same, ensuring consistent results across different runs.

After executing this code, the following variables are assigned:

X\_train: The features for training the model.

X\_test: The features for testing the model.

y\_train: The corresponding target labels for training.

y\_test: The corresponding target labels for testing.

In summary, this code snippet performs a train-test split on the standardized features and target labels, creating separate datasets for training and testing machine learning models. This is a critical step in evaluating the performance and generalization of the trained model.

**MODEL 1: LOGISTIC REGRESSION**

from sklearn.linear\_model import LogisticRegression

log\_reg\_model = LogisticRegression(max\_iter=1000) log\_reg\_model.fit(X\_train, y\_train

The code snippet implements a Logistic Regression model using scikit-learn's LogisticRegression class. Here's a breakdown of what this code does:

from sklearn.linear\_model import LogisticRegression: This line imports the LogisticRegression class from the sklearn.linear\_model module. Logistic Regression is a popular classification algorithm used to model the probability of a binary outcome based on one or more predictor variables.

log\_reg\_model = LogisticRegression(max\_iter=1000): This line creates an instance of the LogisticRegression class. The parameter max\_iter=1000 specifies the maximum number of iterations for the optimization algorithm to converge. Increasing the maximum number of iterations may be necessary for the algorithm to converge if the data is complex or if there are convergence issues.

log\_reg\_model.fit(X\_train, y\_train): This line trains the Logistic Regression model on the training data. The fit() method fits the model to the training data by learning the parameters (coefficients) that best fit the relationship between the features (X\_train) and the target labels (y\_train).

After executing this code:

log\_reg\_model will be a trained Logistic Regression model.

In summary, this code snippet creates and trains a Logistic Regression model using the training data. The trained model can then be used to make predictions on new data or evaluate its performance using the test data.

**6. MODEL 2: DECISION TREES**

from sklearn.tree import DecisionTreeClassifier

dt\_model = DecisionTreeClassifier() dt\_model.fit(X\_train, y\_train

The provided code snippet implements a Decision Tree classifier using scikit-learn's DecisionTreeClassifier class. Here's a breakdown of what this code does:

from sklearn.tree import DecisionTreeClassifier: This line imports the DecisionTreeClassifier class from the sklearn.tree module. Decision Trees are a popular machine learning algorithm used for classification tasks. They make predictions by recursively partitioning the feature space into smaller regions based on feature values.

dt\_model = DecisionTreeClassifier(): This line creates an instance of the DecisionTreeClassifier class without specifying any hyperparameters. When no hyperparameters are provided, the Decision Tree model will use default settings, such as the Gini impurity criterion for splitting and a maximum depth of the tree.

dt\_model.fit(X\_train, y\_train): This line trains the Decision Tree classifier on the training data. The fit() method fits the model to the training data by learning the decision rules that best separate the different classes based on the features (X\_train) and the corresponding target labels (y\_train).

After executing this code:

dt\_model will be a trained Decision Tree classifier.

In summary, this code snippet creates and trains a Decision Tree classifier using the training data. The trained model can then be used to make predictions on new data or evaluate its performance using the test data.

**7. MODEL 3: SUPPORT VECTOR MACHINES (SVM)**

from sklearn.svm import SVC

svm\_model = SVC() svm\_model.fit(X\_train, y\_train

The provided code snippet implements a Support Vector Machine (SVM) classifier using scikit-learn's SVC (Support Vector Classifier) class. Here's a breakdown of what this code does:

from sklearn.svm import SVC: This line imports the SVC class from the sklearn.svm module. SVC is a variant of the Support Vector Machine algorithm used for classification tasks. It works by finding the hyperplane that best separates the classes in the feature space.

svm\_model = SVC(): This line creates an instance of the SVC class without specifying any hyperparameters. When no hyperparameters are provided, the SVC model will use default settings, such as the Radial Basis Function (RBF) kernel for non-linear classification.

svm\_model.fit(X\_train, y\_train): This line trains the SVM classifier on the training data. The fit() method fits the model to the training data by learning the decision boundary that best separates the different classes based on the features (X\_train) and the corresponding target labels (y\_train).

After executing this code:

svm\_model will be a trained SVM classifier.

In summary, this code snippet creates and trains an SVM classifier using the training data. The trained model can then be used to make predictions on new data or evaluate its performance using the test data.

**8. Model Evaluation:**

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_

The provided code snippet imports several evaluation metrics from scikit-learn's metrics module. Here's a breakdown of what this code does:

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix: This line imports the following evaluation metrics:

accuracy\_score: Computes the accuracy of the model, which is the proportion of correctly classified instances among all instances.

precision\_score: Computes the precision of the model, which is the proportion of true positive predictions among all positive predictions.

recall\_score: Computes the recall of the model, which is the proportion of true positive predictions among all actual positive instances.

f1\_score: Computes the F1 score of the model, which is the harmonic mean of precision and recall, providing a balanced measure of the model's performance.

confusion\_matrix: Computes the confusion matrix, which is a table used to evaluate the performance of a classification model, displaying the counts of true positive, false positive, true negative, and false negative predictions.

In summary, this code snippet imports common evaluation metrics used to assess the performance of classification models. These metrics help quantify different aspects of model performance, such as accuracy, precision, recall, and the trade-off between precision and recall. The confusion matrix provides a detailed breakdown of the model's predictions, enabling further analysis of its strengths and weaknesses.

Function to evaluate models

def evaluate\_model(model, X\_test, y\_test): y\_pred = model.predict(X\_test) acc = accuracy\_score(y\_test, y\_pred) prec = precision\_score(y\_test, y\_pred, average='weighted') rec = recall\_score(y\_test, y\_pred, average='weighted') f1 = f1\_score(y\_test, y\_pred, average='weighted') cm = confusion\_matrix(y\_test, y\_pred) return acc, prec, rec, f1, cm

The provided code snippet defines a function named evaluate\_model that evaluates the performance of a classification model using various metrics. Here's a breakdown of what this code does:

def evaluate\_model(model, X\_test, y\_test):: This line defines a function named evaluate\_model that takes three arguments:

model: The trained classification model to be evaluated.

X\_test: The features of the test dataset.

y\_test: The corresponding true labels of the test dataset.

y\_pred = model.predict(X\_test): This line uses the trained model to make predictions on the test dataset (X\_test), and stores the predicted labels in the variable y\_pred.

acc = accuracy\_score(y\_test, y\_pred): This line computes the accuracy of the model by comparing the predicted labels (y\_pred) with the true labels (y\_test) and stores the result in the variable acc.

prec = precision\_score(y\_test, y\_pred, average='weighted'): This line computes the precision of the model using the weighted average of precision scores for each class and stores the result in the variable prec.

rec = recall\_score(y\_test, y\_pred, average='weighted'): This line computes the recall of the model using the weighted average of recall scores for each class and stores the result in the variable rec.

f1 = f1\_score(y\_test, y\_pred, average='weighted'): This line computes the F1 score of the model using the weighted average of F1 scores for each class and stores the result in the variable f1.

cm = confusion\_matrix(y\_test, y\_pred): This line computes the confusion matrix of the model by comparing the true labels (y\_test) with the predicted labels (y\_pred) and stores the result in the variable cm.

return acc, prec, rec, f1, cm: This line returns a tuple containing the computed accuracy (acc), precision (prec), recall (rec), F1 score (f1), and confusion matrix (cm) of the model.

In summary, the evaluate\_model function takes a trained classification model and test dataset as input and computes various evaluation metrics, including accuracy, precision, recall, F1 score, and confusion matrix, to assess the model's performance. These metrics provide valuable insights into how well the model generalizes to unseen data and its ability to correctly classify instances across different classes.

Evaluate all three models

log\_reg\_results = evaluate\_model(log\_reg\_model, X\_test, y\_test) dt\_results = evaluate\_model(dt\_model, X\_test, y\_test) svm\_results = evaluate\_model(svm\_model, X\_test, y\_test

The provided code snippet evaluates three different machine learning models using the evaluate\_model function defined earlier. Here's a breakdown of what this code does:

log\_reg\_results = evaluate\_model(log\_reg\_model, X\_test, y\_test): This line evaluates the Logistic Regression model (log\_reg\_model) using the test dataset (X\_test and y\_test) and stores the evaluation results (accuracy, precision, recall, F1 score, and confusion matrix) in the variable log\_reg\_results.

dt\_results = evaluate\_model(dt\_model, X\_test, y\_test): This line evaluates the Decision Tree model (dt\_model) using the test dataset (X\_test and y\_test) and stores the evaluation results in the variable dt\_results.

svm\_results = evaluate\_model(svm\_model, X\_test, y\_test): This line evaluates the Support Vector Machine (SVM) model (svm\_model) using the test dataset (X\_test and y\_test) and stores the evaluation results in the variable svm\_results.

After executing these lines of code, the following variables will be assigned:

log\_reg\_results: Evaluation results for the Logistic Regression model.

dt\_results: Evaluation results for the Decision Tree model.

svm\_results: Evaluation results for the Support Vector Machine (SVM) model.

These evaluation results provide insights into how well each model performs on the test dataset in terms of accuracy, precision, recall, F1 score, and confusion matrix. This allows for comparison between the performance of different models and helps in selecting the most appropriate model for the given task.

Print results

print("Logistic Regression Results:") print("Accuracy:", log\_reg\_results[0]) print("Precision:", log\_reg\_results[1]) print("Recall:", log\_reg\_results[2]) print("F1 Score:", log\_reg\_results[3]) print("Confusion Matrix:\n", log\_reg\_results[4])

print("Decision Tree Results:") print("Accuracy:", dt\_results[0]) print("Precision:", dt\_results[1]) print("Recall:", dt\_results[2]) print("F1 Score:", dt\_results[3]) print("Confusion Matrix:\n", dt\_results[4])

print("SVM Results:") print("Accuracy:", svm\_results[0]) print("Precision:", svm\_results[1]) print("Recall:", svm\_results[2]) print("F1 Score:", svm\_results[3]) print("Confusion Matrix:\n", svm\_results[4

The provided code snippet prints the evaluation results for each of the three machine learning models: Logistic Regression, Decision Tree, and Support Vector Machine (SVM). Here's a breakdown of what this code does:

**For Logistic Regression:**

print("Logistic Regression Results:"): This line prints a header indicating the results for the Logistic Regression model.

print("Accuracy:", log\_reg\_results[0]): This line prints the accuracy score of the Logistic Regression model.

print("Precision:", log\_reg\_results[1]): This line prints the precision score of the Logistic Regression model.

print("Recall:", log\_reg\_results[2]): This line prints the recall score of the Logistic Regression model.

print("F1 Score:", log\_reg\_results[3]): This line prints the F1 score of the Logistic Regression model.

print("Confusion Matrix:\n", log\_reg\_results[4]): This line prints the confusion matrix of the Logistic Regression model.

**For Decision Tree:**

Similar to the Logistic Regression section, this section prints the evaluation results for the Decision Tree model.

**For Support Vector Machine (SVM):**

Similar to the previous sections, this section prints the evaluation results for the SVM model.

Overall, the code prints the evaluation metrics (accuracy, precision, recall, F1 score) and the confusion matrix for each model, providing a comprehensive summary of their performance on the test dataset. These results allow for comparison between the models and help in assessing their effectiveness for the given classification task.

**10. Results Analysis:**

Analyze and discuss the results for each model

The results analysis involves interpreting the performance metrics obtained from evaluating each model (Logistic Regression, Decision Tree, and Support Vector Machine) and discussing their implications. Here's a summary of the analysis for each model:

**Logistic Regression:**

**Accuracy**: The logistic regression model achieved a certain level of accuracy, indicating the proportion of correctly classified instances. This metric provides a general overview of the model's performance.

**Precision**: Precision measures the proportion of true positive predictions among all positive predictions. A high precision suggests that the model makes few false positive predictions, indicating a low rate of false alarms.

**Recall**: Recall measures the proportion of true positive predictions among all actual positive instances. A high recall indicates that the model effectively captures most positive instances, minimizing false negatives.

**F1 Score**: The F1 score is the harmonic mean of precision and recall, providing a balanced measure of the model's performance. It accounts for both false positives and false negatives.

**Confusion Matrix**: The confusion matrix provides a detailed breakdown of the model's predictions, including true positive, false positive, true negative, and false negative counts. It helps assess the model's performance across different classes.

**Decision Tree:**

Accuracy, precision, recall, and F1 score: Similar to logistic regression, these metrics provide an evaluation of the decision tree model's performance. However, decision trees often exhibit different characteristics compared to logistic regression, such as the ability to capture non-linear relationships and feature interactions.

**Confusion Matrix**: Analyzing the confusion matrix can provide insights into how well the decision tree model distinguishes between different classes and identifies patterns in the data.

**Support Vector Machine (SVM):**

SVMs aim to find the optimal hyperplane that separates different classes, making them suitable for both linear and non-linear classification tasks.

Performance Metrics: Similar to logistic regression and decision trees, accuracy, precision, recall, and F1 score are used to evaluate SVM performance. However, SVMs may excel in scenarios where the decision boundary is complex or non-linear.

Confusion Matrix: Analyzing the confusion matrix can help identify any class imbalances and assess how well the SVM model distinguishes between different classes.

Overall, the analysis involves comparing the performance of each model across multiple metrics and interpreting their strengths and weaknesses in the context of the specific classification task. Additionally, it may involve considering factors such as model complexity, interpretability, and computational efficiency when selecting the most suitable model for deployment.

**11. Visualization (Optional):**

Visualization can be done using matplotlib or seaborn.

import matplotlib.pyplot as plt import seaborn as

The provided code snippet imports the matplotlib library for creating plots and the seaborn library for enhancing the visual appeal of the plots. Here's a breakdown of what this code does:

import matplotlib.pyplot as plt: This line imports the pyplot module from the matplotlib library, allowing for the creation of various types of plots, such as line plots, bar plots, and scatter plots. The pyplot module provides a MATLAB-like interface for creating plots.

import seaborn as sns: This line imports the seaborn library, which is built on top of matplotlib and provides additional functionalities for creating attractive and informative statistical visualizations. Seaborn simplifies the process of creating complex plots and offers several built-in themes and color palettes to enhance the aesthetics of the plots.

After executing these import statements, you can use the plt and sns aliases to create and customize plots for visualizing data, model performance metrics, distributions, correlations, and other aspects of the dataset or analysis results.

Function to plot confusion matrix

def plot\_confusion\_matrix(cm, target\_names): plt.figure(figsize=(8, 6)) sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=target\_names, yticklabels=target\_names) plt.xlabel('Predicted Labels') plt.ylabel('True Labels') plt.title('Confusion Matrix') plt.show()

The provided code snippet defines a function named plot\_confusion\_matrix that creates a visualization of a confusion matrix. Here's a breakdown of what this code does:

def plot\_confusion\_matrix(cm, target\_names):: This line defines a function named plot\_confusion\_matrix that takes two arguments:

cm: The confusion matrix to be visualized.

target\_names: The names of the target classes.

plt.figure(figsize=(8, 6)): This line creates a new figure for the plot with a specific size (8 inches in width and 6 inches in height).

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=target\_names, yticklabels=target\_names): This line creates a heatmap using the seaborn library. The heatmap represents the confusion matrix (cm), with annotations (annot=True) displaying the numerical values of each cell. The format of the annotations is set to integers (fmt="d"), and the color map is set to "Blues". The x-axis and y-axis tick labels are set to the target class names (target\_names).

plt.xlabel('Predicted Labels'): This line sets the label for the x-axis as "Predicted Labels".

plt.ylabel('True Labels'): This line sets the label for the y-axis as "True Labels".

plt.title('Confusion Matrix'): This line sets the title of the plot as "Confusion Matrix".

plt.show(): This line displays the plot.

In summary, the plot\_confusion\_matrix function takes a confusion matrix and target class names as input and creates a heatmap visualization of the confusion matrix using seaborn. The plot provides insights into the performance of a classification model by visualizing the distribution of predicted versus true labels across different classes.

Visualize confusion matrix for Logistic Regression

plot\_confusion\_matrix(log\_reg\_results[4], wine\_data.target\_names)

Visualize confusion matrix for Decision Tree

plot\_confusion\_matrix(dt\_results[4], wine\_data.target\_names)

Visualize confusion matrix for SVM

plot\_confusion\_matrix(svm\_results[4], wine\_data.target\_names)

summarise in words

The provided code snippet visualizes the confusion matrices for each of the three machine learning models: Logistic Regression, Decision Tree, and Support Vector Machine (SVM). Here's a summary of what this code does:

plot\_confusion\_matrix(log\_reg\_results[4], wine\_data.target\_names): This line calls the plot\_confusion\_matrix function to visualize the confusion matrix for the Logistic Regression model. It passes the confusion matrix (log\_reg\_results[4]) and the target class names (wine\_data.target\_names) as arguments.

plot\_confusion\_matrix(dt\_results[4], wine\_data.target\_names): This line calls the plot\_confusion\_matrix function to visualize the confusion matrix for the Decision Tree model. It passes the confusion matrix (dt\_results[4]) and the target class names (wine\_data.target\_names) as arguments.

plot\_confusion\_matrix(svm\_results[4], wine\_data.target\_names): This line calls the plot\_confusion\_matrix function to visualize the confusion matrix for the Support Vector Machine (SVM) model. It passes the confusion matrix (svm\_results[4]) and the target class names (wine\_data.target\_names) as arguments.

After executing these lines of code, three separate confusion matrix visualizations will be displayed, each corresponding to the performance of one of the machine learning models. These visualizations provide insights into how well each model classified instances into different classes and help assess their overall performance.

**Conclusion:**

This documentation summarizes the wine classification project, covering data preprocessing, model training, evaluation, and visualization. The project demonstrates the application of machine learning algorithms for wine type classification and provides insights into model performance and data characteristics.

This documentation serves as a comprehensive record of the project, facilitating reproducibility and future improvements.