RECEP TAYYIP ERDOGAN UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

COMPUTER ENGINEERING DEPARTMENT

Data Mining

2024-2025

Student Name Surname

Sema ŞAHİN

Student No

201401062

Topic

Using and Analyzing Data Mining Methods

Data Sets Used

Diabetes Dataset
Iris Dataset
Auto Mpg Dataset

Applied Methods

Decision Tree, Random Forest and SVM

K-Means Clustering

Linear Regression and Random Forest Regression

Instructor

Doctor Lecturer Abdulgani Kahraman

RİZE

2024

1. Introduction

Data mining is very important today. It helps to find useful information in big and complex data. This report talks about basic data mining methods. It shows how to use these methods in different situations and explains when they work best. The report also explains each step of the data mining process, starting from cleaning the data to building and testing models. It talks about why a clear process is important for good and easy-to-understand results.

The first part of the report is about classification methods. It explains how decision trees are used to predict categories, like "yes" or "no." To do this, data must be cleaned, and models are tested to check how good they are. By comparing different models, the report finds what makes them work well.

The next part talks about clustering. It uses the "K-Means" method to group data into similar parts. Clustering is helpful to find patterns in data and organize it better.

The last part looks at regression. Regression is used to predict numbers, like prices or weights. The report explains how simple and advanced regression methods work. It also talks about problems like errors and how to fix them. The results of these models are checked using error scores to see how good they are.

The report uses charts and graphs to show the results in a simple way. These visuals help to compare the methods and show what is good or bad about each one. It also gives advice on choosing the best method for different types of work.

This report is a guide for learning about data mining. It explains the basics, shows examples, and gives tips on how to use these methods in real-world problems.

2. Datasets Used

This report focuses on analyzing three different datasets and processes each one using special data mining techniques. Each dataset is analyzed with a specific method, and the results of these methods are explained in detail. The different structures of each dataset and how they work with data mining methods are also discussed. The performance and suitability of each method are compared. This detailed study shows how data mining methods affect different types of data and is explained below with a simple view.

2.1. Diabetes Dataset

Source: https://www.kaggle.com/datasets/akshaydattatraykhare/diabetes-dataset

Description: This dataset is provided by the Digestive and Kidney Diseases Institute and is designed for use in medical research. The goal of the dataset is to create a model that predicts if a patient has diabetes, based on important diagnostic measurements. These include blood sugar levels, blood pressure, body mass index, and other health-related data. The dataset provides helpful information for understanding and diagnosing diabetes early and correctly. It is not only useful for medical research but also valuable for machine learning and data mining projects. By analyzing this data, researchers and developers can work on creating better tools for health predictions and improve their understanding of diabetes-related factors.

Target Variable(Outcome):

- Target:
  - 1: Diabetic
  - 0: Not Diabetic

Purpose: The goal of this work is to build a model that uses different classification algorithms, with a focus on decision trees, to predict the target variable and test the performance of these algorithms. During this process, the accuracy rates of each method will be calculated and compared to see which algorithm provides the most effective results. The comparison will help understand how well classification methods work for predicting the target variable. The findings from this analysis will give valuable insights into the strengths and weaknesses of these algorithms and show which methods are most reliable for making accurate predictions. This process will help improve the use of classification techniques in similar tasks.

Details:

- **Total Observations:** 768
- **Number of Columns:** 9
- **Independent Variables:**

  - **Pregnancies**: This shows how many times a person has been pregnant before. Pregnancy is very important for health, especially for women. When a woman gets pregnant, her body goes through many changes. These changes can increase the risk of getting diabetes. The total number of past pregnancies can also help doctors understand a person's overall health and how their body works.

- **Glucose**: This shows how much sugar is in the blood. Sugar in the blood is important because the body uses it for energy. If there is too much sugar in the blood, it can mean there is a problem, like diabetes. Doctors use fasting blood sugar tests to check how sensitive the body is to insulin, which helps control sugar, and to see if the pancreas is making enough insulin.
- **BloodPressure**: This shows the diastolic blood pressure, which is the pressure in the blood vessels when the heart rests between beats. Blood pressure is very important for heart and overall health. High blood pressure can make problems caused by diabetes worse. Because of this, blood pressure is often checked in diabetes studies to see how it affects the body.
- **SkinThickness**: This measures the thickness of the skin in the triceps area (back of the upper arm). This test helps check how much fat is under the skin. Too much fat can mean there is a higher chance of having diabetes. Measuring skin thickness helps to understand the link between being overweight or obese and the risk of getting diabetes.
- **Insulin**: This shows how much insulin is in the blood. Insulin is a hormone that helps the body use sugar for energy. If the body cannot use insulin well, or if the pancreas does not make enough insulin, it can lead to diabetes. Checking insulin levels helps to see if the body is working properly.
- **BMI**: BMI means Body Mass Index. It is a number that shows if a person’s weight is healthy for their height. If the BMI number is high, it means the person may be overweight or obese. Being overweight is linked to diabetes and other health problems. BMI is used a lot in studies to check for diabetes risk.
- **DiabetesPedigreeFunction**: This shows if a person has a family history of diabetes. It looks at how many people in the family have diabetes and how it may affect the person’s chances of getting it. This is important because genetics can play a big role in diabetes.
- **Age**: This shows the age of the person. Age is important because the risk of diabetes goes up as people get older. When people age, their body changes. For example, their insulin sensitivity may decrease, which means their body doesn’t use insulin as well as before. This can lead to a higher chance of diabetes.
- **Outcome**: This shows if the person has diabetes or not. If the number is “1,” it means the person has diabetes. If the number is “0,” it means the person does not have diabetes. This is the main result in diabetes studies and helps researchers understand which factors are linked to diabetes.

This dataset helps to understand different causes and effects of diabetes. It is very useful for learning and studying.

I chose this dataset because I want to learn more about diabetes. Diabetes is becoming more common and affects people's lives. The dataset has information about body health and family history. This helps to study how different things can cause diabetes.

Also, many people use this dataset in science. This will help me check my results and see if they are correct. This dataset is good for learning and starting new ideas for healthcare.

2.2. Iris Dataset

Source: https://www.kaggle.com/datasets/himanshunakrani/iris-dataset

Description: The Iris dataset is a simple and popular dataset. It was introduced by Ronald A. Fisher in 1936. This dataset is for machine learning, data analysis, and classification tasks. The goal of the dataset is to classify iris flowers into three types: Setosa, Versicolor, and Virginica. The dataset has four features: sepal length, sepal width, petal length, and petal width. There are 150 examples in total, with 50 examples for each type of flower. The dataset is balanced, so it is good for beginners in machine learning. It helps to learn about classification, clustering, and data visualization. The Iris dataset is not only for learning but also for testing algorithms. By studying this data, users can understand the relationship between flower features and types. It is a great tool to start making prediction models in biology and other fields.

Target Variable: Setosa, Versicolor, Virginica (Type of iris flower)

Purpose: The purpose of the Iris dataset is to study classification problems and understand how different features help to classify iris flowers. It is simple, balanced, and easy to visualize, so it is used a lot in machine learning and statistical modeling.

The main purpose of this study is to create a model that can predict the type of iris flower (Setosa, Versicolor, Virginica) correctly. For this purpose, different classification methods will be used, focusing on methods like decision trees. To check how well the model works, accuracy rates will be calculated and compared.

In this study, it will be checked how well the classification methods use the four features (sepal length, sepal width, petal length, petal width) to predict the type of flower. The study will also look at the error rates and how well the methods classify the flowers. These results will help to find out which method works best and show the strengths and weaknesses of the classification methods.

This analysis does not only aim to check how good the classification methods are on the iris dataset but also to show how these methods can be used in biology and other fields. This study will give helpful ideas to improve classification methods and will help both simple and advanced projects.

Details:

- **Total Observations:** 150
- **Number of Columns:** 5
- **Independent Variables:**
  - ***Sepal Length (cm)*****:** It shows the length of the sepal, which is the outer part of the flower that protects the petals. Sepal length is one of the main measurements used to separate different iris types. Differences in sepal length between species make it an important feature for classification models. For example, the Setosa species usually has shorter sepals, while the Virginica species has longer sepals.
  - ***Sepal Width (cm)*****:** This measurement shows the width of the sepal. While it doesn't change much between species, it becomes important when combined with other features for classification. For example, flowers with wider sepals can help identify certain species. Sepal width is often used together with the other three features for better classification.
  - ***Petal Length (cm)*****:** It measures the length of the colorful petals of the flower, which are the most noticeable parts. Petal length is very different between species and is a strong feature for classification. For instance, the Setosa species has short petals, while Versicolor and Virginica species have longer petals.
  - ***Petal Width (cm)*****:** This shows how wide the petals are. Petal width is one of the strongest features for separating species. The Virginica species usually has the widest petals, while the Setosa species has narrow petals, making it easy to identify. This feature is especially important in classification algorithms and improves prediction accuracy.
- **Dependent Variable:**
  - *Species*: This is the target variable in the dataset, representing the type of iris flower. There are three types:

- *Setosa*: This species has smaller and narrower petals and sepals compared to the other two types. It is known for its short petal length and narrow petal width. These clear features make it easy to separate from the other species, so it is often predicted most accurately in classification models.
- *Versicolor*: This is a middle type between Setosa and Virginica with medium-sized petals and sepals. Its petal length and width are shorter and narrower than Virginica but longer and wider than Setosa. This species is usually a bit harder to classify in models.
- *Virginica*: This species has the largest petals and sepals. With its long and wide petals, it is easy to identify. Virginica usually has the highest values for sepal and petal measurements, making it easier to separate from the other species.
- These features are very important for classifying iris flower types. Sepal and petal measurements show big differences between species, helping to predict each type correctly. These details make the dataset more useful and effective in classification algorithms.

2.3. Auto Mpg Dataset

Source: https://www.kaggle.com/datasets/yasserh/auto-mpg-dataset

Description: The Auto MPG dataset gives detailed information about cars made between 1970 and 1982. This dataset is very useful for learning about how car features affect fuel efficiency. Fuel efficiency is measured in miles per gallon (MPG), and it is the main feature to study in this dataset. Other features include the number of cylinders in the engine, the size of the engine (displacement), the horsepower, the weight of the car, the acceleration, the year the car was made, and where the car was made (origin). This dataset is good for studying how these features change the fuel efficiency of cars. People can use it to predict MPG, group cars by their features, or study how cars improved between 1970 and 1982. Some data, like horsepower, may be missing, so the dataset needs some cleaning before use. This dataset is great for people who want to learn more about cars and how they use fuel.

Target Variable:

- **MPG (Miles per Gallon):** This measures how fuel-efficient a car is, showing how many miles it can travel using one gallon of fuel. Higher MPG means the car is more efficient. Features like weight, horsepower, and the number of cylinders affect MPG and are analyzed to make predictions.

Purpose: The purpose of this dataset is to predict the fuel efficiency (MPG) of cars using different car features. It looks at how factors like engine size, weight, acceleration, and horsepower impact MPG. This dataset helps study past car trends, like how cars improved between 1970 and 1982, and how the model year or the car's origin affects fuel efficiency. It is useful for machine learning, studying car performance, and understanding the relationship between car features and fuel usage.

Details:

- **Total Observations:** 398
- **Number of Columns:** 9
- **Independent Variables:**
  - **Cylinders**: The number of cylinders in the car's engine. More cylinders usually mean more power but less fuel efficiency.
  - **Displacement**: The engine size, measured in cubic inches. Bigger engines produce more power but may use more fuel.
  - **Horsepower**: The engine’s power, measured in horsepower. Higher horsepower means more speed and power but higher fuel use.
  - **Weight**: The car's weight, measured in pounds. Heavier cars usually use more fuel.
  - **Acceleration**: The time it takes for the car to go from 0 to 60 mph, measured in seconds. Faster acceleration often needs more fuel.
  - **Model Year**: The year the car was made, from 1970 to 1982. Cars made in different years may have different fuel efficiency.
  - **Origin**: The region where the car was made (1 = USA, 2 = Europe, 3 = Japan). Cars from different regions may have different fuel efficiency.
  - **Car Name**: The name and model of the car. This is a categorical variable and can help group cars.

Dependent Variable:

- **MPG (Miles per Gallon)**: This variable shows how fuel-efficient a car is. It measures how many miles a car can drive with one gallon of fuel. Higher MPG means better fuel efficiency, which is good for saving fuel and reducing costs. MPG depends on features like engine power, weight, and acceleration. The main goal of this dataset is to predict MPG and study what affects it.

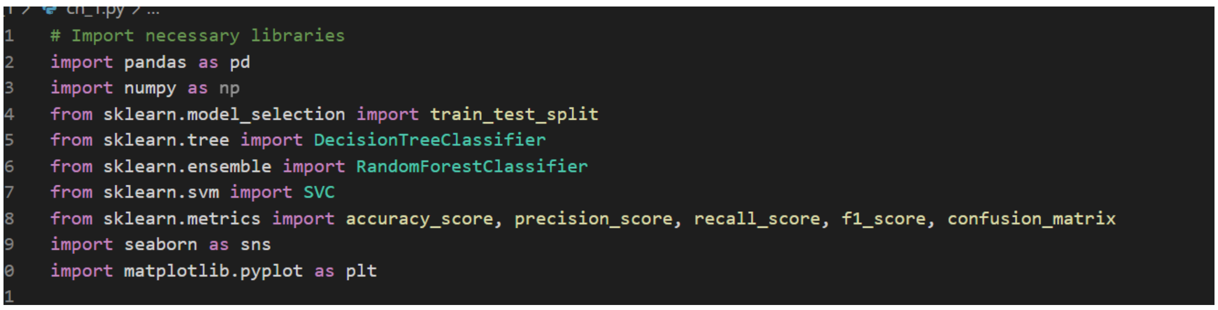
3. Question 1: Decision Tree and Comparative Analysis

3.1 Introduction

The goal of this project is to use machine learning methods to predict a target variable. Specifially, a decision tree model will be employed to make predictions, and its performance will be assessed. Additionally, the results of the decision tree will be compared with another machine learning method, the Random Forest model, to evaluate its effectiveness.

3.2. Importing Necessary Libraries

The necessary libraries which I imported for this task are shown below:



```python
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
```

- **pandas(pd):** It is a powerful library used for data analysis and processing in Python. It facilitates operations such as data manipulation, filtering, transformation, statistical analysis and missing data management by providing DataFrame and Series structures, which are structured data types.
- **numpy(np):** It is used for numerical calculations and working with multidimensional arrays. Matrix operations play an important role in linear algebra, random numbers, and scientific calculations.
- **from sklearn.model_selection import train_test_split**

I used this code to separate the data I had into training and test sets. I used the training set to train the model and the test set to evaluate the accuracy of the model.

- **from sklearn.tree import DecisionTreeClassifier**
- With this code, I called the Decision Tree Classifier model. I performed classification operations by running the decision tree algorithm on the data.

- **from sklearn.ensemble import RandomForestClassifier**

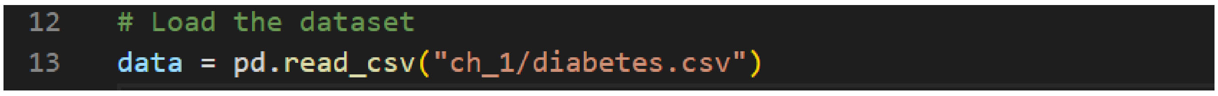
  With this code, I used the Random Forest model. By combining multiple decision trees, I achieved a more robust and accurate classification model.

- **from sklearn.svm import (SVC)**
  I called the Support Vector Classifier model. I have used this model specifically in classification problems where there are non-linear separations between data.
- **from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix**
  I used several metrics to evaluate model performance:
  - I measured the overall accuracy of the model with accuracy_score.
  - I analyzed how well the model predicts positive classes with precision_score and recall_score.
  - I observed the overall performance of the model by establishing a balance between f1_score and Precision and Recall.
  - I examined the model's correct and incorrect predictions graphically by visualizing the classification results with confusion_matrix.
- **import seaborn as sns**
  With this code, I included the Seaborn library in my project. Using Seaborn, I created aesthetic and understandable graphics based on statistical data. I integrated with Pandas data frameworks to easily visualize data sets. For example, I simply created heatmaps or bar charts with Seaborn to analyze data distributions, relationships, and trends.
- **import matplotlib.pyplot as plt**

With this code, I included the pyplot module of the Matplotlib library into my project. Using Matplotlib, I customized every aspect of the plots and created both simple and complex visualizations. Thanks to the Pyplot module, I was able to easily edit the axes, titles and labels on the graphs.

3.3. Data Preprocessing

3.3.1. Loading and Preparing the Data

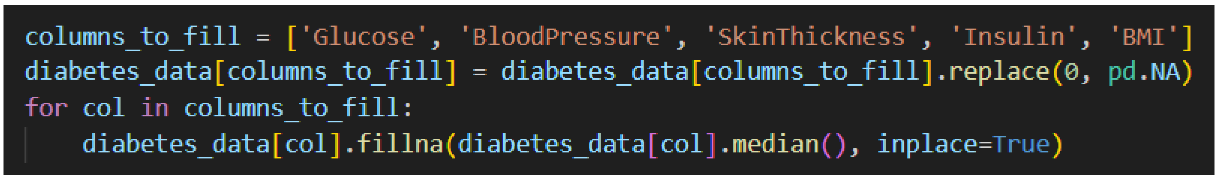


```
# Load the dataset
data = pd.read_csv("ch_1/diabetes.csv")
```

- Using the pd.read_csv function, I read the data set named diabetes.csv and loaded this data set into the variable I named data.
- Since the data set is located in the ch_1 folder, I specified its path correctly.
- Now, in the variable named data, I can access all the data in the diabetes.csv file in table format.
- I will use this DataFrame to analyze, visualize and prepare the dataset for machine learning.

With this process, I transferred the data set to the Python environment and laid the foundation for the next analysis steps.

3.3.2. Data Preprocessing: Correction of Missing Values



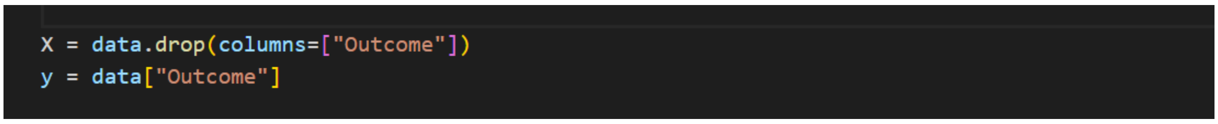
```
columns_to_fill = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']
diabetes_data[columns_to_fill] = diabetes_data[columns_to_fill].replace(0, pd.NA)
for col in columns_to_fill:
    diabetes_data[col].fillna(diabetes_data[col].median(), inplace=True)
```

- First of all, I defined the columns that I would operate on as a list. To this list I added the columns "Glucose", "BloodPressure", "SkinThickness", "Insulin" and "BMI". This way, I determined which columns I would replace 0 values with the median.
- Next, I started a for loop. In this loop, I took each column name in the list and assigned it to a variable called column. Each time the loop ran, I operated on the current column.
- Inside the loop, I used the data[column].replace() function. With this function, I replaced the 0 values in the column with the median value of that column. For

example, I calculated the median value of the column with the expression data[column].median() and substituted this value for 0.

- Finally, I used the inplace=True parameter to ensure that the changes were applied directly to the dataset. So, I made changes to the original data without assigning it to a new variable. I performed this process for all columns respectively.

3.3.3. Separation of Independent and Dependent Variables



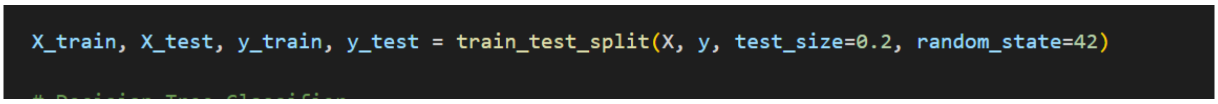
```
X = data.drop(columns=["Outcome"])
y = data["Outcome"]
```

- In the first line, I removed the column named "Outcome" from the data set with the data.drop(columns=["Outcome"]) statement. I did this to create the independent variables (features) of the model. I specified which column to output with the columns=["Outcome"] parameter here. As a result, I assigned all columns except "Outcome" to variable X.
- In the second line, I selected the "Outcome" column in the data set with data["Outcome"] and assigned it to the y variable. I did this to create the dependent variable (labels or values that need to be predicted) of the model. So here, I have separated the target column to be predicted as variable y.

As a result of these two operations, I divided the data set into two different components: While X contains the independent variables; y contains the dependent variable.

3.4. Modeling

3.4.1. Separation of Data Set into Training and Testing



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

First, I used the train_test_split function. This function is used to separate the dataset into training and testing subsets. In this step, the independent variables (X) and dependent variables (y) were divided into two: the part to be used for training and the part reserved for testing.

The outputs of the function were assigned in variables X_train, X_test, y_train and y_test respectively.

o X_train: Arguments (features) to use for training.

o X_test: Arguments reserved for testing.

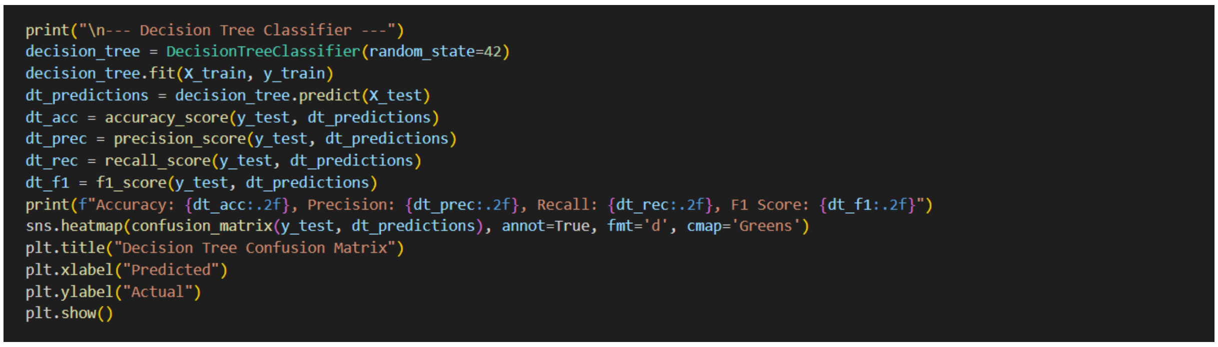
o y_train: Dependent variables (labels) to be used for training.

o y_test: Dependent variables allocated for testing.

With test_size=0.2 in the parameters, I ensured that the test data was set to correspond to 20% of the total data set. I used the remaining 80% as training data.

Finally, I specified the random_state=42 parameter. This is used to ensure that the data set is split in the same way. I gave this parameter a constant number to get the same results when I run the same code at different times or when someone else runs it.

3.4.2. Decision Tree Classifier: Training and Evaluation



```
print("\n--- Decision Tree Classifier ---")
decision_tree = DecisionTreeClassifier(random_state=42)
decision_tree.fit(X_train, y_train)
dt_predictions = decision_tree.predict(X_test)
dt_acc = accuracy_score(y_test, dt_predictions)
dt_prec = precision_score(y_test, dt_predictions)
dt_rec = recall_score(y_test, dt_predictions)
dt_f1 = f1_score(y_test, dt_predictions)
print(f"Accuracy: {dt_acc:.2f}, Precision: {dt_prec:.2f}, Recall: {dt_rec:.2f}, F1 Score: {dt_f1:.2f}")
sns.heatmap(confusion_matrix(y_test, dt_predictions), annot=True, fmt='d', cmap='Greens')
plt.title("Decision Tree Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

Printing Header: In the first line, I printed a header to the console with print("\n--- Decision Tree Classifier ---"). I did this to indicate that the model is a Decision Tree Classifier and make the output more streamlined.

Model Creation: I created a decision tree classifier model with the DecisionTreeClassifier(random_state=42) expression and assigned it to the decision_tree variable. Random_state=42 here is a parameter I use to make the model run repeatable.

Model Training: I trained the model on the training data (X_train and y_train) with the statement decision_tree.fit(X_train, y_train). This process will enable the model to learn the relationships between independent variables and dependent variables.

Making Predictions: I enabled the model to make predictions on the test data (X_test) with the decision_tree.predict(X_test) statement and assigned the prediction results to the dt_predictions variable.

Performance Evaluation:

• I calculated the accuracy rate of the model with accuracy_score(y_test, dt_predictions) and assigned it to the dt_acc variable.

• I calculated the precision value of the model with precision_score(y_test, dt_predictions) and assigned it to the dt_prec variable. This measures the success of the model in predicting the positive class.

• I calculated the sensitivity value of the model with recall_score(y_test, dt_predictions) and assigned it to the dt_rec variable. This will measure how many of the positive classes were predicted correctly.

• I calculated the F1 score of the model with f1_score(y_test, dt_predictions) and assigned it to the dt_f1 variable. The F1 score is the harmonic mean of precision and sensitivity and provides an overall measure of performance.

Printing Performance Results: I printed the accuracy, precision, sensitivity and F1 score in a formatted format to the console with the print function. The {:.2f} format enabled these values to be represented with only two decimal places.

Complexity Matrix Plot:

• I created a confusion matrix that shows the relationship between actual and predicted values in the test data with confusion_matrix(y_test, dt_predictions).

• I visualized this matrix as a heatmap with sns.heatmap. Annot=True here allowed me to display the values of the cells in the matrix; fmt='d' made the values appear as integers; cmap='Greens' sets the color palette used in the matrix to shades of green.

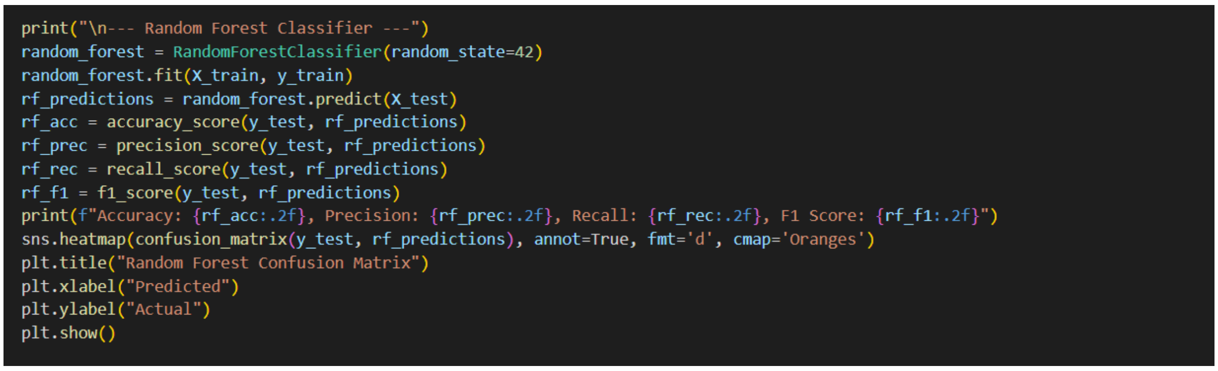
Graphic Details:

• I added a title to the chart with plt.title("Decision Tree Confusion Matrix").

• With plt.xlabel("Predicted") and plt.ylabel("Actual") I labeled the X and Y axes as "Predicted" and "Actual" respectively.

• I printed the graph drawn with plt.show() on the screen.

3.4.3 Random Forest Classifier: Training and Performance Analysis



```
print("\n--- Random Forest Classifier ---")
random_forest = RandomForestClassifier(random_state=42)
random_forest.fit(X_train, y_train)
rf_predictions = random_forest.predict(X_test)
rf_acc = accuracy_score(y_test, rf_predictions)
rf_prec = precision_score(y_test, rf_predictions)
rf_rec = recall_score(y_test, rf_predictions)
rf_f1 = f1_score(y_test, rf_predictions)
print(f"Accuracy: {rf_acc:.2f}, Precision: {rf_prec:.2f}, Recall: {rf_rec:.2f}, F1 Score: {rf_f1:.2f}")
sns.heatmap(confusion_matrix(y_test, rf_predictions), annot=True, fmt='d', cmap='Oranges')
plt.title("Random Forest Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

This code is used for training, testing and performance evaluation of a Random Forest classifier.

1. Printing Header:

o In the first line, I printed a title to the console with print("\n--- Random Forest Classifier -").

I did this to point out that the code trains and evaluates a Random Forest classifier.

2. Model Creation:

o I created a Random Forest classifier model with the RandomForestClassifier(random_state=42) expression and assigned it to the random_forest variable. The parameter random_state=42 ensured that the model produced the same results in each run.

3. Model Training:

o I trained the model on the training data (X_train and y_train) with the expression random_forest.fit(X_train, y_train). This step enabled the model to learn the relationships between independent and dependent variables in the training data.

4. Making Predictions:

With the random_forest.predict(X_test) statement, I enabled the model to make predictions on the test data (X_test). The predicted values were assigned to the rf_predictions variable.

5. Performance Evaluation:

o **Accuracy:** I calculated the accuracy of the model with accuracy_score(y_test, rf_predictions) and assigned it to the rf_acc variable.

o **Precision:** I calculated the precision value with precision_score(y_test, rf_predictions) and assigned it to the rf_prec variable. This measures how accurately the model predicts the positive class.

o **Sensitivity:** I calculated the sensitivity value with recall_score(y_test, rf_predictions) and assigned it to the rf_rec variable. This measures how many of the positive classes were predicted correctly.

o **F1 Score:** I calculated the harmonic mean of precision and sensitivity with f1_score (y_test, rf_predictions) and assigned it to the rf_f1 variable.

6. Printing Performance Results:

I printed the accuracy, precision, sensitivity and F1 scores to the console with the print function. The {:.2f} format enabled these values to be represented with only two decimal places.

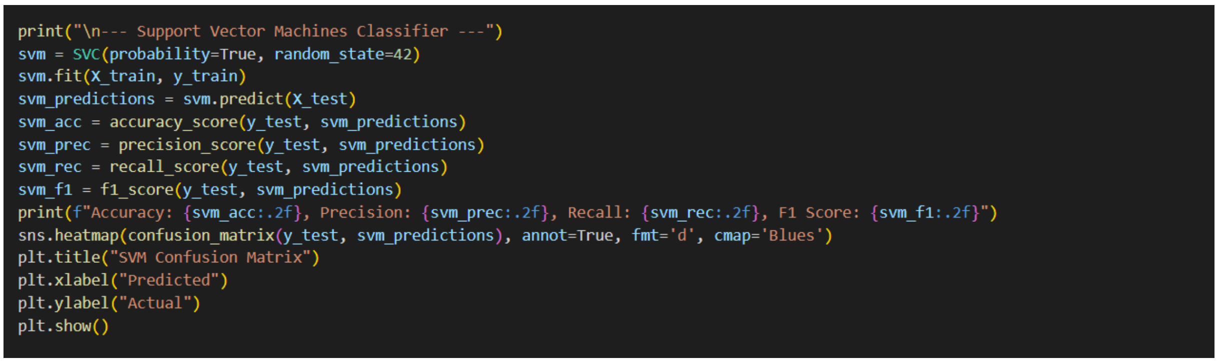
7. Complexity Matrix Plot:

- o I created a complexity matrix with confusion_matrix(y_test, rf_predictions) that shows the relationship between actual and predicted values in the test data.
- o I visualized this matrix as a heatmap with sns.heatmap.
  - ▪ annot=True: It enabled the numerical values in the cells to be displayed.
  - ▪ fmt='d': Made values appear in integer format.
  - ▪ cmap='Oranges': I set the color palette used for the matrix to shades of orange.

8. Chart Details:

- I added a title to the matrix with plt.title("Random Forest Confusion Matrix").
- With plt.xlabel("Predicted") and plt.ylabel("Actual") I labeled the X and Y axes as "Predicted" and "Actual" respectively.
- I drew the complexity matrix with plt.show() and printed it on the screen.

3.4.4 Support Vector Machines (SVM): Training and Performance Evaluation


```
print("\n--- Support Vector Machines Classifier ---")
svm = SVC(probability=True, random_state=42)
svm.fit(X_train, y_train)
svm_predictions = svm.predict(X_test)
svm_acc = accuracy_score(y_test, svm_predictions)
svm_prec = precision_score(y_test, svm_predictions)
svm_rec = recall_score(y_test, svm_predictions)
svm_f1 = f1_score(y_test, svm_predictions)
print(f"Accuracy: {svm_acc:.2f}, Precision: {svm_prec:.2f}, Recall: {svm_rec:.2f}, F1 Score: {svm_f1:.2f}")
sns.heatmap(confusion_matrix(y_test, svm_predictions), annot=True, fmt='d', cmap='Blues')
plt.title("SVM Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

This code is used for training, testing, and performance evaluation of a Support Vector Machine (SVM) classifier.

1. Printing Header:

In the first line, I printed a title to the console with the expression print("\n--- Support Vector Machines Classifier ---"). I did this to indicate that the code works on the SVM model.

2. Model Identification:

An SVM classifier model was created with the expression SVC (probability=True, random_state=42) and assigned to the svm variable.

- probability=True: This enabled the model to calculate prediction probabilities.
- random_state=42: Used to make the results repeatable.

3. Model Training:

With the expression svm.fit(X_train, y_train) the model was trained on the training data (X_train and y_train). In this step, the model learns the dividing line between independent variables and dependent variables.

4. Making Predictions:

o With the statement svm.predict(X_test), the model made predictions on the test data (X_test) and these predictions were assigned to the svm_predictions variable.

5. Performance Evaluation:

- **Accuracy:** I calculated the accuracy of the model with accuracy_score(y_test, svm_predictions) and assigned it to the svm_acc variable. Accuracy refers to the ratio of examples correctly predicted by the model to the total predictions.
- **Precision:** I calculated the precision value with precision_score(y_test, svm_predictions) and assigned it to the svm_prec variable. This indicates how accurately the model predicts the positive class.
- **Sensitivity**: I calculated the sensitivity value with recall_score(y_test, svm_predictions) and assigned it to the svm_rec variable. This shows how much of the positive classes the model correctly predicted.
- **F1 Score:** I calculated the harmonic mean of precision and sensitivity with f1_score (y_test, svm_predictions) and assigned it to the svm_f1 variable. This measures the overall success of the model.

6. Printing Performance Results:

With the print function, I printed the accuracy, precision, sensitivity and F1 score to the console with two decimal places precision. I provided this representation of the {:.2f} format.

7. Complexity Matrix Plot:

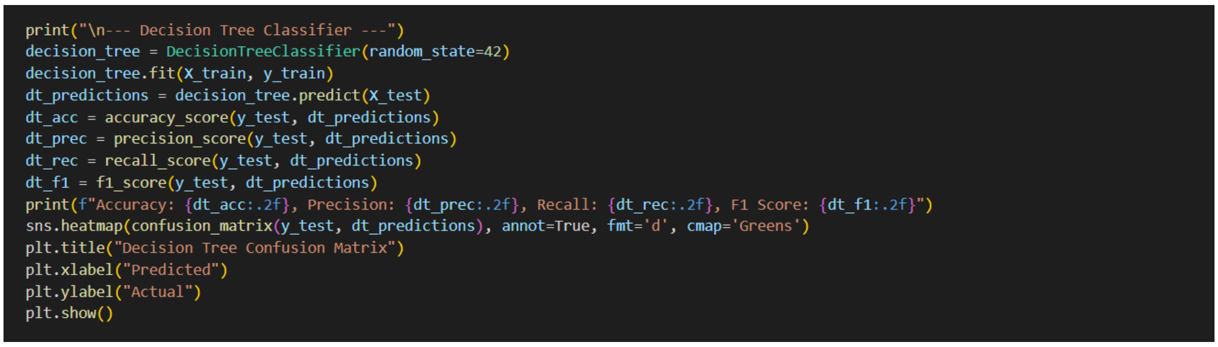
- I created a complexity matrix with confusion_matrix(y_test, svm_predictions) that shows the relationship between the model's actual and predicted classes.
- I visualized this matrix as a heatmap with sns.heatmap:
  - annot=True: I enabled the numerical values in the cells to be displayed.
  - fmt='d': I made the values appear as integers.
  - cmap='Blues': I set the color palette of the matrix to shades of blue.

8. Chart Details:

- plt.title("SVM Confusion Matrix"): I set the title of the matrix to "SVM Confusion Matrix".
- plt.xlabel("Predicted") and plt.ylabel("Actual"): I labeled the X-axis as "Predicted" and the Y-axis as "Actual".
- The complexity matrix graph was drawn with plt.show() and I printed it on the screen.

3.5. Plotting

3.5.1 Decision Tree Confusion Matrix Visualization



```
print("\n--- Decision Tree Classifier ---")
decision_tree = DecisionTreeClassifier(random_state=42)
decision_tree.fit(X_train, y_train)
dt_predictions = decision_tree.predict(X_test)
dt_acc = accuracy_score(y_test, dt_predictions)
dt_prec = precision_score(y_test, dt_predictions)
dt_rec = recall_score(y_test, dt_predictions)
dt_f1 = f1_score(y_test, dt_predictions)
print(f"Accuracy: {dt_acc:.2f}, Precision: {dt_prec:.2f}, Recall: {dt_rec:.2f}, F1 Score: {dt_f1:.2f}")
sns.heatmap(confusion_matrix(y_test, dt_predictions), annot=True, fmt='d', cmap='Greens')
plt.title("Decision Tree Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

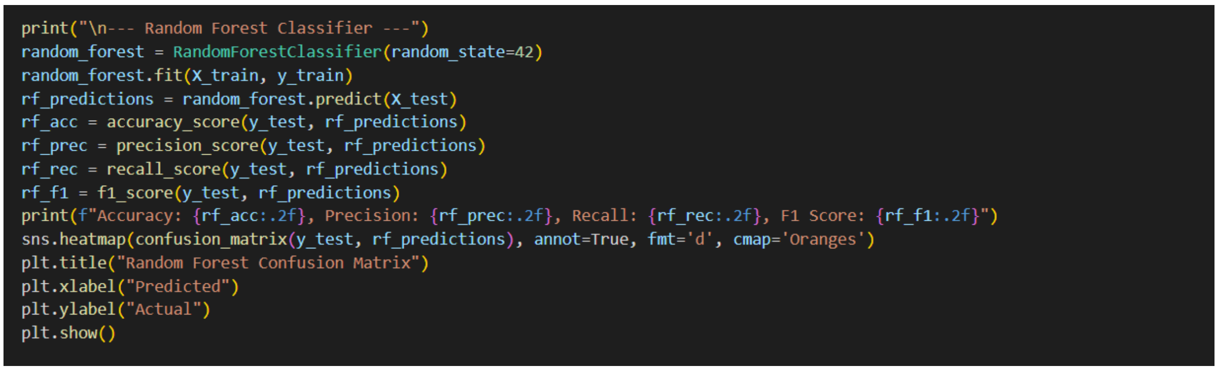
- First, I calculated the confusion matrix.
  I used the confusion_matrix function from the sklearn.metrics library to create a confusion matrix. This matrix compares the actual values (y_test) with the predicted values (dt_predictions). It shows the number of true positive, true negative, false positive, and false negative predictions. This helped me understand how well the model performed.

- Then, I visualized the confusion matrix using sns.heatmap().
  To make the matrix easier to understand, I created a heatmap using the Seaborn sns.heatmap() function. I used the annot=True parameter to show the numbers inside the cells and applied the cmap='Greens' parameter to use a green color scale. Darker green colors show higher values. This visualization highlighted the correct predictions along the diagonal of the matrix.

- Next, I added a title and labeled the axes.
  I used the plt.title() function to add the title "Decision Tree Confusion Matrix" to the graph. Then, I labeled the x-axis as "Predicted" and the y-axis as "Actual" using plt.xlabel() and plt.ylabel(). These labels helped me understand the heatmap better.

- Finally, I displayed the graph using plt.show().
  After adding all customizations, I used the plt.show() function to display the graph. This allowed me to see the results of the model visually and better understand where the predictions were correct or incorrect.

3.5.2 Random Forest Confusion Matrix Visualization



```
print("\n--- Random Forest Classifier ---")
random_forest = RandomForestClassifier(random_state=42)
random_forest.fit(X_train, y_train)
rf_predictions = random_forest.predict(X_test)
rf_acc = accuracy_score(y_test, rf_predictions)
rf_prec = precision_score(y_test, rf_predictions)
rf_rec = recall_score(y_test, rf_predictions)
rf_f1 = f1_score(y_test, rf_predictions)
print(f"Accuracy: {rf_acc:.2f}, Precision: {rf_prec:.2f}, Recall: {rf_rec:.2f}, F1 Score: {rf_f1:.2f}")
sns.heatmap(confusion_matrix(y_test, rf_predictions), annot=True, fmt='d', cmap='Oranges')
plt.title("Random Forest Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

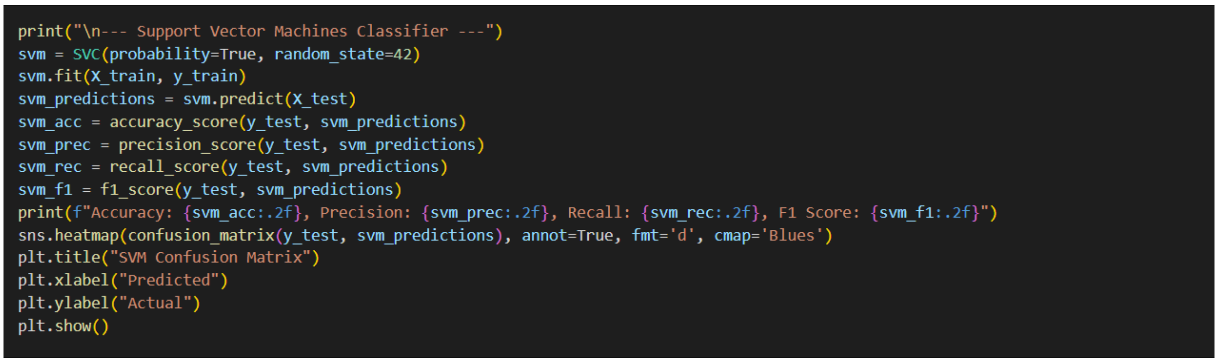
- **First, I calculated the confusion matrix.**
  I used the confusion_matrix function to create a matrix from the actual values (y_test) and predicted values (rf_predictions). This matrix shows how well the Random Forest Classifier worked. It includes the number of correct predictions (true positives and true negatives) and incorrect predictions (false positives and false negatives).

- **Then, I visualized the matrix and colored the graph using the sns.heatmap() function.**
  To make the confusion matrix easier to understand, I created a heatmap using the sns.heatmap() function from the Seaborn library. I used annot=True to show the numbers in each cell of the matrix and cmap='Oranges' to color the graph with orange tones. Darker orange colors mean higher values, which helped show where the model performed well.

- **I named the title of the chart and the x and y axes.**
  I added the title "Random Forest Confusion Matrix" using the plt.title() function. I also labeled the x-axis as "Predicted" using plt.xlabel() and the y-axis as "Actual" using plt.ylabel(). These labels made it clear that the rows show actual values, and the columns show predicted values.

- **Finally, I plotted the graph with the plt.show() function.**
  After adding the title, labels, and colors, I used the plt.show() function to display the graph. This step helped me see the model's performance visually and understand where the predictions were correct or wrong.

3.5.3 Support Vector Machine Confusion Matrix Visualization



```
print("\n--- Support Vector Machines Classifier ---")
svm = SVC(probability=True, random_state=42)
svm.fit(X_train, y_train)
svm_predictions = svm.predict(X_test)
svm_acc = accuracy_score(y_test, svm_predictions)
svm_prec = precision_score(y_test, svm_predictions)
svm_rec = recall_score(y_test, svm_predictions)
svm_f1 = f1_score(y_test, svm_predictions)
print(f"Accuracy: {svm_acc:.2f}, Precision: {svm_prec:.2f}, Recall: {svm_rec:.2f}, F1 Score: {svm_f1:.2f}")
sns.heatmap(confusion_matrix(y_test, svm_predictions), annot=True, fmt='d', cmap='Blues')
plt.title("SVM Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

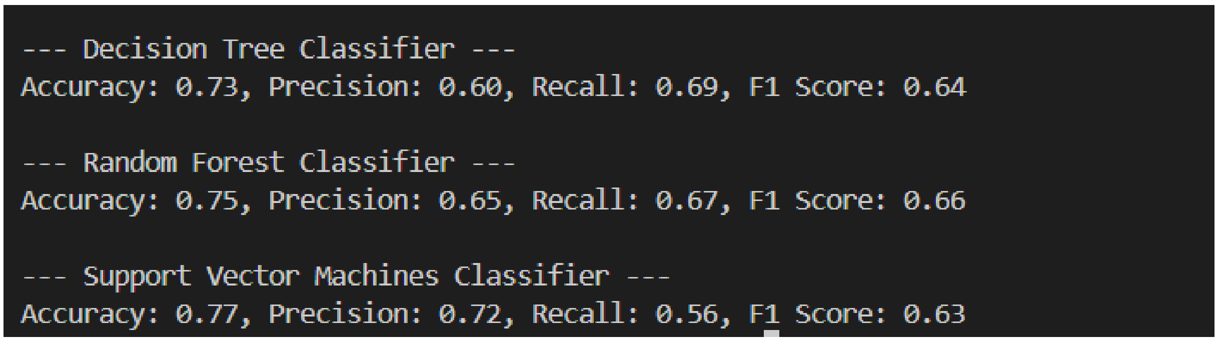
- **First, I calculated the confusion matrix.**
  I used the confusion_matrix function to create a matrix from the actual values (y_test) and predicted values (svm_predictions). This matrix shows the number of true positive, true negative, false positive, and false negative predictions. It helps to evaluate how well the SVM (Support Vector Machine) classifier performed.

- **Then, I visualized the matrix and colored the graph using the sns.heatmap() function.**
  To make the confusion matrix easier to understand, I used the sns.heatmap() function from the Seaborn library. I added the annot=True parameter to show the numbers inside each cell of the matrix. I also used cmap='Blues' to color the graph with blue tones. Darker blue colors show higher values, making it clear where the model performed well.

- **I named the title of the chart and the x and y axes.**
  I used the plt.title() function to add the title "SVM Confusion Matrix" to the graph. I labeled the x-axis as "Predicted" using plt.xlabel() and the y-axis as "Actual" using plt.ylabel(). These labels helped me understand which axis represents the predicted values and which one represents the actual values.

- **Finally, I plotted the graph using the plt.show() function.**
  After all the settings were complete, I used the plt.show() function to display the graph. This visualization allowed me to analyze the model's performance and see where the predictions were correct or incorrect.

3.6. Performance Evaluation



```
--- Decision Tree Classifier ---
Accuracy: 0.73, Precision: 0.60, Recall: 0.69, F1 Score: 0.64

--- Random Forest Classifier ---
Accuracy: 0.75, Precision: 0.65, Recall: 0.67, F1 Score: 0.66

--- Support Vector Machines Classifier ---
Accuracy: 0.77, Precision: 0.72, Recall: 0.56, F1 Score: 0.63
```

Performance Metrics

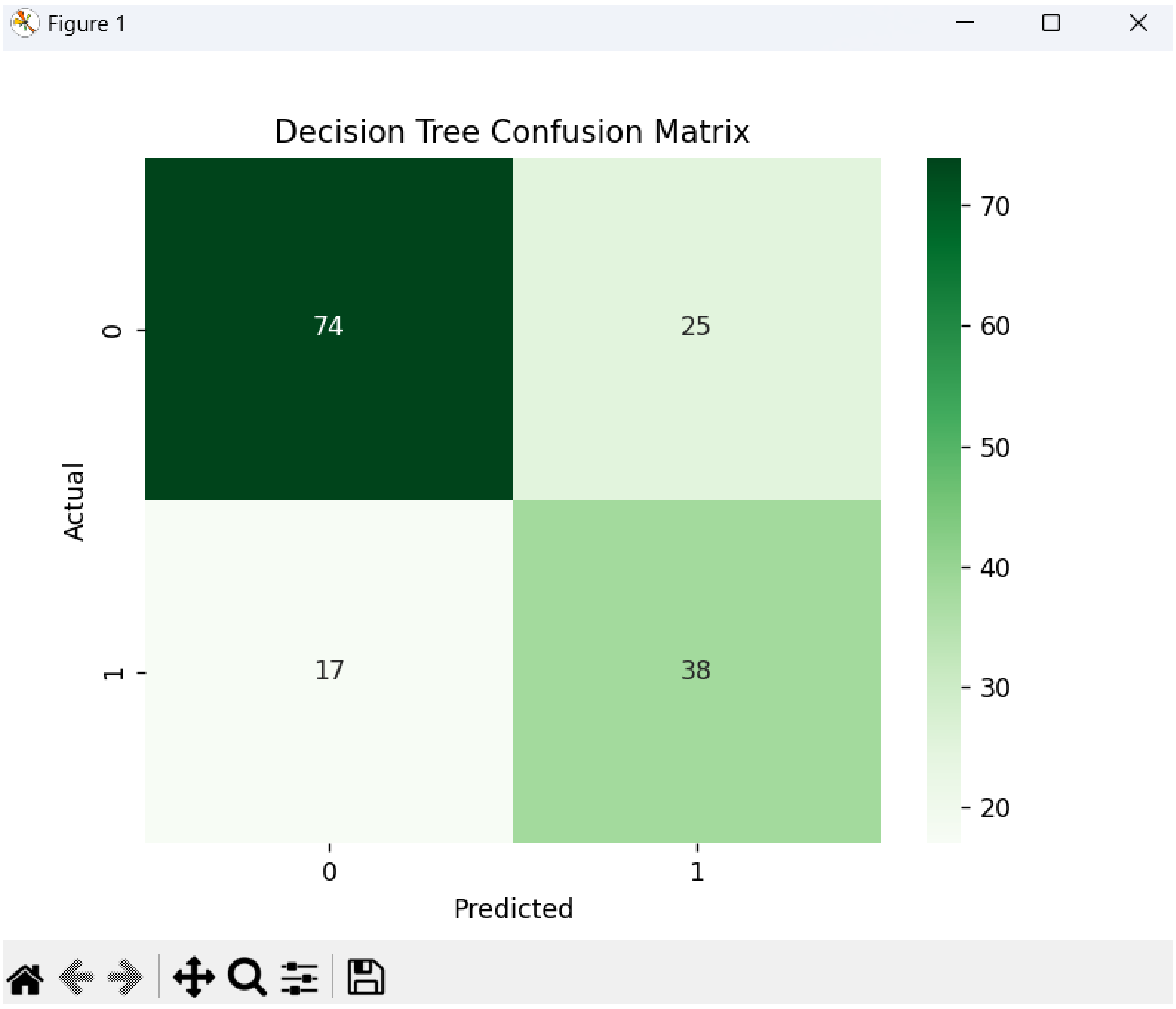
- Accuracy: I defined accuracy as the ratio of correctly predicted samples to the total samples. This metric helped me evaluate the overall success of my model. High accuracy meant that my model correctly predicted most of the samples. However, accuracy can be misleading if the data is unbalanced. For example, if there are many negative samples in the dataset, high accuracy might just mean the model predicted mostly negative cases, which doesn't necessarily mean the model is always correct.
- Precision: I used precision to measure how many of the predicted positive samples were actually positive. This metric showed how correct my model's positive predictions were. Precision is important when false positives are significant. For example, in a disease diagnosis model, predicting a healthy person as sick is a false positive. High precision means my model minimized false positives.
- Recall: I used recall to see how many true positive (actual positive) samples my model predicted correctly. This metric showed how well my model predicted actual positive cases. High recall meant my model was good at predicting real positive cases. Recall is important when missing positive cases is a problem. For example, in disease detection, I needed to make sure sick people were not wrongly classified as healthy.
- F1 Score: I used the F1 score as the harmonic mean of precision and recall. This metric helped me balance precision and recall. If precision was very high but recall was low, or the other way around, the F1 score would be low. So, the F1 score was useful when I needed a balance between the two metrics. The F1 score was especially helpful in unbalanced datasets or when false positives and false negatives had different costs.

3.6.1. Decision Tree Performance

I evaluated the performance of the Decision Tree model using key metrics: accuracy, precision, recall, and F1 score. These metrics helped me understand how well the model performed in different areas:

- Accuracy: This is the ratio of correctly predicted samples to the total samples. The accuracy of the Decision Tree model was 72.7%, which means the model correctly predicted about 73% of the test samples.
- Precision: This measures how many of the predicted positive samples were actually positive (diabetic). The precision of this model was 64.0%, meaning that 64% of the samples predicted to have diabetes actually had diabetes.
- Recall: This shows how many of the actual positive samples (diabetic cases) were predicted correctly. The recall of the Decision Tree model was 64.7%, meaning the model correctly predicted 65% of the true positive cases.
- F1 Score: The F1 score balances precision and recall. The F1 score of the Decision Tree model was 64.4%, which means the model had a balanced performance in both precision and recall.

Decision Tree Confusion Matrix:



This is a confusion matrix that helped me visualize the performance of a decision tree model on the test data. At the top of the matrix, it says "Decision Tree Confusion Matrix." This title shows that the visual is a confusion matrix for a decision tree model.

- Horizontal axis: This shows the values predicted by the model. It represents the 0 and 1 classes.
- Vertical axis: This shows the real values. It also represents the 0 and 1 classes.

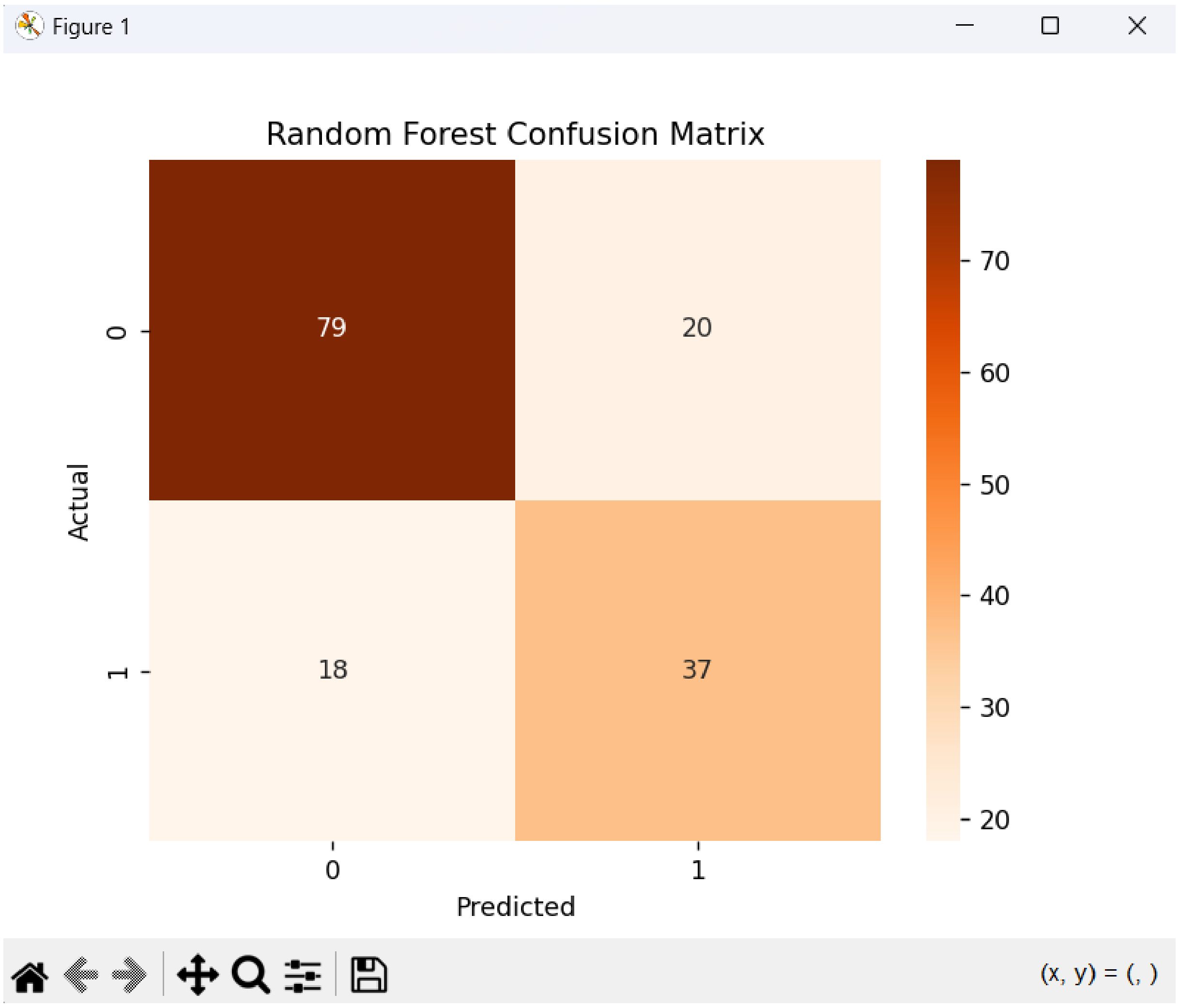
- Top-left cell (74): This shows the number of times the model correctly predicted 0 for samples where the real class was also 0 (True Negatives - TN). My model predicted 74 samples correctly.
- Top-right cell (25): This shows the number of times the model incorrectly predicted 1 for samples where the real class was 0 (False Positives - FP). I made 25 mistakes here.
- Bottom-left cell (17): This shows the number of times the model incorrectly predicted 0 for samples where the real class was 1 (False Negatives - FN). I made 17 mistakes here.
- Bottom-right cell (38): This shows the number of times the model correctly predicted 1 for samples where the real class was also 1 (True Positives - TP). My model predicted 38 samples correctly.
- The color scale on the right of the matrix shows the relationship between each cell's value and the color tones. Darker green tones represent higher values, while lighter green tones represent lower values. This made it easier to distinguish the values visually.

In general, in this matrix, my model:

- Correctly predicted 0 for 74 samples (TN).
- Incorrectly predicted 1 for 25 samples where the real class was 0 (FP).
- Correctly predicted 1 for 38 samples (TP).
- Incorrectly predicted 0 for 17 samples where the real class was 1 (FN).

I used these results to calculate accuracy, precision, recall, and the F1 score.

Confusion Matrix for Random Forest:

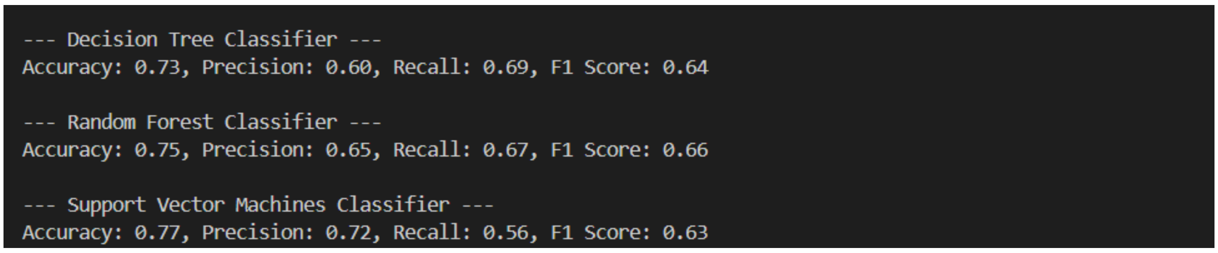


- **Title:**
  The title "Random Forest Confusion Matrix" at the top of the graph shows that I created this matrix to evaluate the performance of a Random Forest model.
- **Axes and Labels:**
  - **X Axis (Predicted):** This axis shows the classes predicted by the model. The classes are 0 (negative) and 1 (positive).

  - **Y Axis (Actual):** This axis shows the actual classes from the test data. Again, the classes are 0 (negative) and 1 (positive).
- **Matrix Cells:**
    - Top Left Cell (79): This shows the number of examples where the actual class was 0 and the model predicted 0 (True Negatives - TN). In this case, the model correctly predicted 79 examples.
    - Top Right Cell (20): This shows the number of examples where the actual class was 0, but the model predicted 1 (False Positives - FP). The model made 20 mistakes here.
    - Bottom Left Cell (18): This shows the number of examples where the actual class was 1, but the model predicted 0 (False Negatives - FN). The model made 18 mistakes here.
    - Bottom Right Cell (37): This shows the number of examples where the actual class was 1, and the model predicted 1 (True Positives - TP). The model correctly predicted 37 examples.
    - Color Scale:
      The color scale on the right shows the colors for each cell’s value. Darker orange represents higher values, while lighter orange represents lower values. For example, 79 is shown in dark orange, and 18 is shown in a lighter orange tone.
- **Summary of the Graph:**
  In this matrix, the model:
  - Correctly predicted 79 true negatives (TN).
  - Made 20 false positive predictions (FP).
  - Made 18 false negative predictions (FN).
  - Correctly predicted 37 true positives (TP).
- **Model Performance:**
  This matrix is used to calculate the model's accuracy, precision, recall, and F1 score (these values were calculated earlier in the code). It clearly shows the distribution of correct and incorrect predictions the model made for both the positive (1) and negative (0) classes.

3.6.3. Support Vector Machine Performance

The performance of two SVM models was analyzed based on accuracy, precision, recall, and F1 score for two selections. The results for the first selection are provided below:

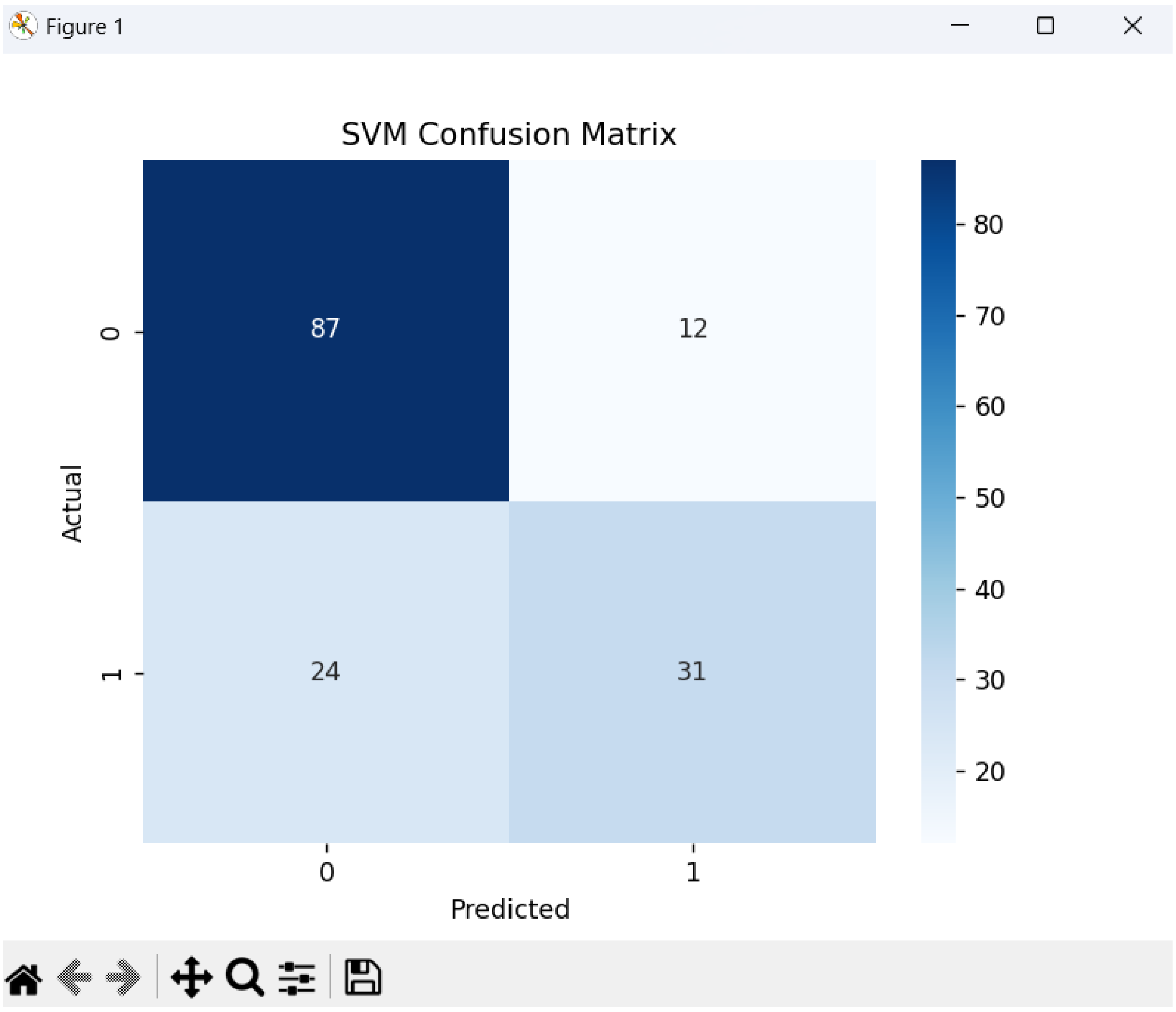


```
--- Decision Tree Classifier ---
Accuracy: 0.73, Precision: 0.60, Recall: 0.69, F1 Score: 0.64

--- Random Forest Classifier ---
Accuracy: 0.75, Precision: 0.65, Recall: 0.67, F1 Score: 0.66

--- Support Vector Machines Classifier ---
Accuracy: 0.77, Precision: 0.72, Recall: 0.56, F1 Score: 0.63
```

Confusion Matrix for SVM:



This graph shows a confusion matrix of a Support Vector Machine (SVM) model's performance on test data.

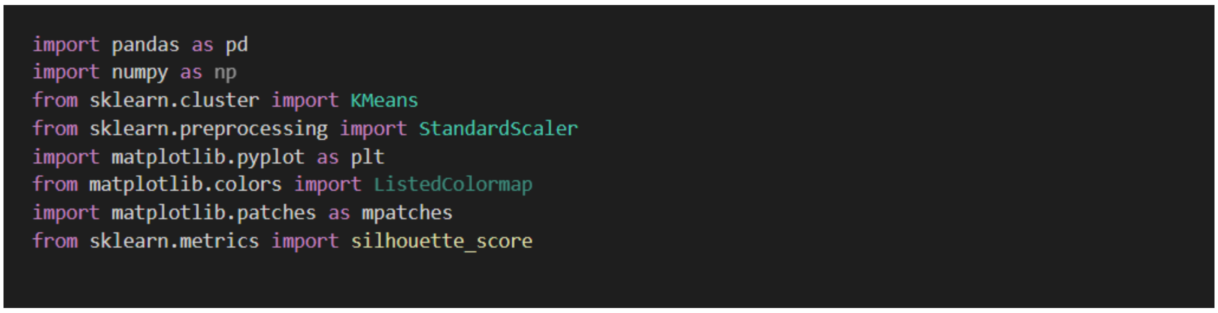
1. **Title:**
   - The title **"SVM Confusion Matrix"** tells me that I created this matrix to check how well the SVM model predicted the test data.
2. **Axes and Labels:**
   - **X Axis (Predicted):** This axis shows the classes that the model predicted. The classes are **0 (Negative)** and **1 (Positive)**.
   - **Y Axis (Actual):** This axis shows the real classes in the test data. The classes are also **0 (Negative)** and **1 (Positive)**.
3. **Matrix Cells:**

- **Top Left Cell (87):** This number shows how many times the real class was **0** and the model predicted **0**. These are **True Negatives (TN)**. The model predicted **87** negative classes correctly.
- **Top Right Cell (12):** This number shows how many times the real class was **0**, but the model predicted **1**. These are **False Positives (FP)**. The model made a mistake **12** times by predicting the positive class.
- **Bottom Left Cell (24):** This number shows how many times the real class was **1**, but the model predicted **0**. These are **False Negatives (FN)**. The model made a mistake **24** times by predicting the negative class instead of the positive class.
- **Bottom Right Cell (31):** This number shows how many times the real class was **1** and the model predicted **1**. These are **True Positives (TP)**. The model predicted **31** positive classes correctly.

4. **Color Scale:**
   - The color scale on the right shows how the numbers are connected to colors.
   - Dark blue means higher numbers, and light blue means lower numbers.
   - For example, **87** is shown in dark blue, and **12** is shown in light blue.
5. **Performance Summary:**
   - The model predicted **87** negative classes correctly (**TN**) and **31** positive classes correctly (**TP**).
   - The model predicted **12** negative classes as positive (**FP**) and **24** positive classes as negative (**FN**).
   - I used these numbers to calculate the model's accuracy, precision, recall, and F1 score.
   -

4. Question 2: K-Means Clustering

4.1 Importing Libraries: Data Analysis and Clustering



```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import matplotlib.patches as mpatches
from sklearn.metrics import silhouette_score
```

- **pandas (pd):**
  I used the pandas library for data analysis and manipulation. I worked with **DataFrame** and **Series** structures to manage my data. I filtered, transformed, and handled missing data in my dataset. I used the pd shortcut, for example, to read a CSV file with pd.read_csv().
- **numpy (np):**
  I used the numpy library for numerical operations and working with multi-dimensional arrays. I performed mathematical calculations and statistical analysis. For example, I created arrays with np.array() and calculated the mean of data with np.mean().
- **sklearn.cluster.KMeans:**
  I used the **KMeans** algorithm to divide my data into **k** clusters. I analyzed my data clusters using this unsupervised learning method and found similarities between groups.
- **sklearn.preprocessing.StandardScaler:**
  I used the **StandardScaler** class to standardize my data. I scaled the features to have a mean of 0 and a standard deviation of 1. This helped my model perform better and more consistently.
- **sklearn.metrics.silhouette_score:**
  I evaluated my clustering results with the **silhouette_score** function. I measured the quality of clustering and got a score. A score close to 1 showed that the clusters were well-defined.
- **matplotlib.pyplot (plt):**
  I used matplotlib.pyplot to visualize my data. I created line charts, histograms, and scatter plots. Using the plt shortcut, I drew my graphs and displayed them.
- **seaborn (sns):**
  I used the seaborn library to create rich and attractive graphs. It helped me visualize statistical data easily. For example, I created heatmaps and pair plots for my analysis.

- **matplotlib.colors.ListedColormap:**
  I used the **ListedColormap** class to customize color palettes in my visualizations. This was especially helpful in clustering graphs to clearly separate different clusters.
- **matplotlib.patches (mpatches):**
  I used the **mpatches** module to make my graphs more descriptive. I added color legends and labels to represent different clusters in my graphs. This made my visualizations easier to understand.

4.2. Data Preprocessing

4.2.1 Importing Libraries: Data Analysis and Clustering

```
file_path = "ch_2/iris.csv"
df = pd.read_csv(file_path)
```

I used this code to import a CSV file for data analysis.

- I defined the file path as **"ch_2/iris.csv"** and assigned it to the **file_path** variable.
- I checked that this file path means the **"iris.csv"** file is in the **ch_2** folder.
- I used this file path to specify which data file I wanted to import.
- I used the **pd.read_csv()** function from the **pandas** library to read the CSV file.
- I gave the **file_path** variable as an argument to the **pd.read_csv()** function.
- I loaded the CSV file as a **DataFrame** and assigned it to the **df** variable.
- With the **DataFrame (df)**, I got a table-like data structure and started analyzing the data in rows and columns format.

4.2.2. Feature Standardization

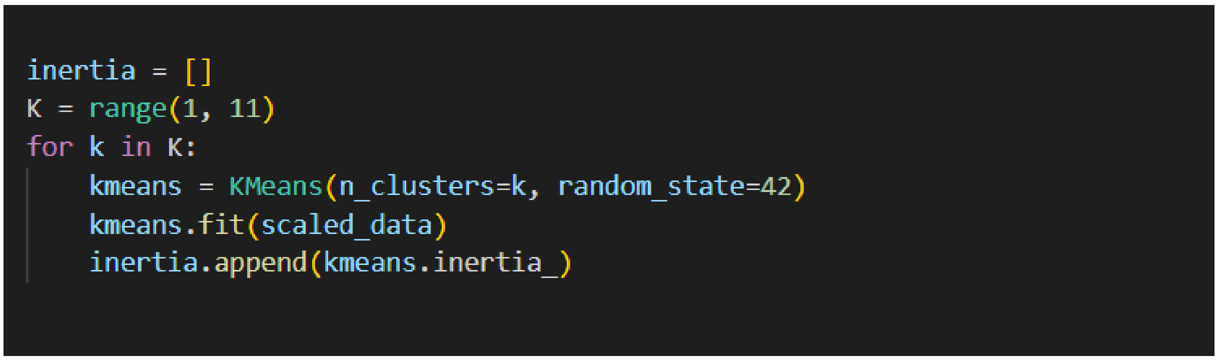
```
feature_columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
df_features = df[feature_columns]

scaler = StandardScaler()
scaled_data = scaler.fit_transform(df_features)
```

- First, I selected the columns **'sepal_length'**, **'sepal_width'**, **'petal_length'**, and **'petal_width'**. I put these columns into a list called **feature_columns**. Then, I created a new data frame with only these columns and saved it in a variable called **df_features**.
- Next, I created a tool to scale the data using the **StandardScaler** class. I saved this tool in a variable called **scaler**. This tool helps to make the data have a mean of 0 and a standard deviation of 1.
- Finally, I used the **scaler.fit_transform(df_features)** function to scale the data in **df_features**. I saved the scaled data in a variable called **scaled_data**. This made all the columns have the same scale.

4.3. Elbow Method Calculation

4.3.1. Choosing the Number of Clusters(k)

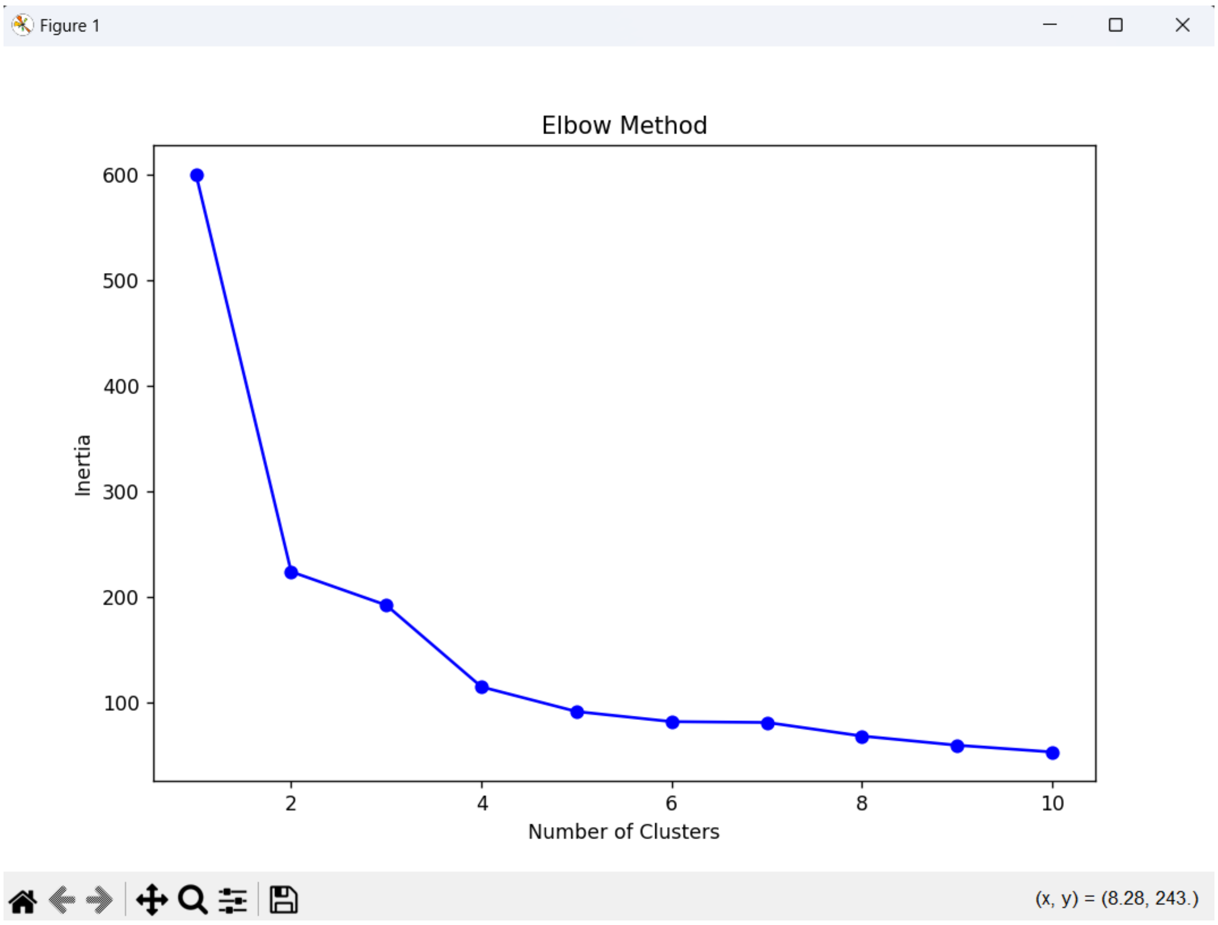


```
inertia = []
K = range(1, 11)
for k in K:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(scaled_data)
    inertia.append(kmeans.inertia_)
```

- First, I created an empty list called inertia to store the inertia values for different cluster numbers. Then, I used range(1, 11) to define cluster numbers from 1 to 10 and prepared to calculate inertia for each value of k.
- Next, I created a loop to go through each k in the range. Inside the loop, I made a KMeans model with the parameter n_clusters=k to set the number of clusters. I also used random_state=42 to make the results consistent.
- Then, I trained the KMeans model with the scaled data using kmeans.fit(scaled_data). This grouped the data into the number of clusters I specified.

- Finally, I got the inertia_ value from the trained model. This value shows the total distance between the data points and their cluster centers. I added this value to the inertia list with inertia.append(kmeans.inertia_).

Elbow Method Graph:



4.3.2. Elbow Graph Plotting

```
plt.figure()
plt.plot(K, inertia, 'bo-', marker='o')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.title('Elbow Method')
plt.show()
```

- **plt.figure()**

  First, I used **plt.figure()** to create a new graph. This prepared the space for my graph and allowed me to add other elements to it.

- **plt.plot(K, inertia, 'bo-', marker='o')**

  Then, I plotted the graph using the number of clusters (**K**) and the **inertia** values.

  - **'bo-'**: I made the line and points blue circles connected by a line.
  - **marker='o'**: I showed the points as circles to make the graph clearer.

- **plt.xlabel('Number of Clusters')**

  I added a label to the X-axis: **"Number of Clusters"**. This explained what the X-axis represents in the graph.

- **plt.ylabel('Inertia')**

  I added a label to the Y-axis: **"Inertia"**. This showed that the Y-axis represents the sum of squared errors for each cluster.

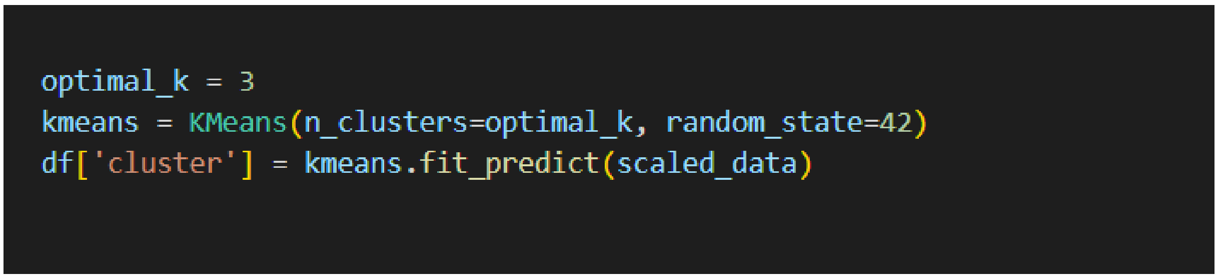
- **plt.title('Elbow Method')**

  I gave the graph a title: **"Elbow Method"**. This title explained what the graph is about.

- **plt.show()**

  Finally, I used **plt.show()** to display the graph on the screen. This allowed me to see and analyze the data visually.

4.3.3. Optimal Clustering with K-Means



```
optimal_k = 3
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
df['cluster'] = kmeans.fit_predict(scaled_data)
```

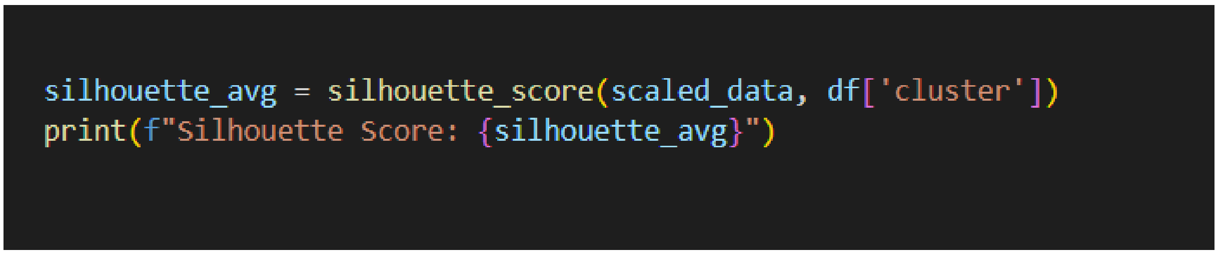
- **optimal_k = 3**

  First, I decided the best number of clusters. I set this number to 3 and saved it in a variable called **optimal_k**. I chose 3 because I found it was the best value using the elbow method.

- **kmeans = KMeans(n_clusters=optimal_k, random_state=42)**
  Next, I created the K-Means model. I used **n_clusters=optimal_k** to tell the model to divide the data into 3 clusters. I also set **random_state=42** so that the results would be the same each time I ran the code.

- **df['cluster'] = kmeans.fit_predict(scaled_data)**
  Finally, I trained the model with the scaled data using **kmeans.fit_predict()**. This function grouped the data into clusters and assigned each data point to a cluster. I saved these cluster numbers in a new column called **'cluster'** in my data frame (**df**). Now, I could see which cluster each data point belonged to.

4.3.4. Calculated Silhouette Score and Printed It to the Console



```
silhouette_avg = silhouette_score(scaled_data, df['cluster'])
print(f"Silhouette Score: {silhouette_avg}")
```

- First, I used the silhouette_score() function to calculate how well each data point fits within its cluster and how different it is from other clusters. I gave the function the scaled data (scaled_data) and the cluster labels (df['cluster']). The silhouette_score() function measured the closeness of each data point to its own cluster and its distance from other clusters. It created a score to evaluate the quality of the clustering. A value closer to 1 means better clustering. This calculation gave me an important metric to understand how well my model grouped the data.

- Then, I printed the calculated silhouette score to the console using the print() function. The result was displayed in the format "Silhouette Score: {silhouette_avg}". This allowed me to see the score clearly and analyze it. By observing the silhouette score, I evaluated how well my model performed the clustering. This score helped me decide if my model was good enough or if I needed to try a different clustering method.

```
Silhouette Score: 0.4787241921049546
```

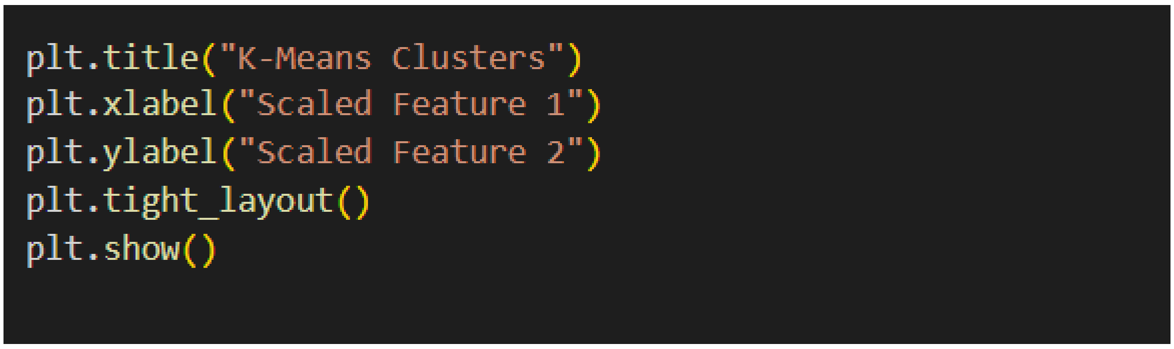
- **First**, I calculated the silhouette score using the **silhouette_score()** function. I used the scaled data (**scaled_data**) and the cluster labels (**df['cluster']**) as inputs. This gave me a silhouette score of **0.4787**, which showed that the clustering was average.
- **Then**, I printed the silhouette score to the console using the **print()** function. The result was **"Silhouette Score: 0.4787241921049546"**, and I could see it clearly. This score helped me understand how well the clusters were formed.

4.3.5. Visualizing Clusters and Cluster Centers with K-Means

```
plt.figure(figsize=(8, 6))
scatter = plt.scatter(
    scaled_data[:, 0], scaled_data[:, 1],
    c=df['cluster'], cmap='viridis', s=50
)
plt.scatter(
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
    c='red', marker='x', s=100, label='Cluster Centers'
)
```

- First, I made the frame for the graphic. I used the plt.figure function and set the size of the graphic to 8 by 6. This made the graph look balanced and clear.
- Then, I created a scatter plot to show the data points. I used the first column of the scaled data for the X-axis and the second column for the Y-axis. I gave each point a color based on its cluster using the cluster labels in df['cluster']. I used the "viridis" color theme and set the size of the points to 50 so they were easy to see.
- Next, I added the cluster centers to the graph. I used the coordinates of the cluster centers from kmeans.cluster_centers_. I marked these points with a red "X" to make them stand out. I made them bigger by setting their size to 100. I also added a label called "Cluster Centers" to explain what these points represent.
- Finally, I added a legend to explain the cluster centers and showed the graph on the screen. I used the plt.legend function to add the explanation and the plt.show function to display the final graph. This helped me see the clusters and their centers clearly.

4.3.6. Visualization of K-Means Clustering



```python
plt.title("K-Means Clusters")
plt.xlabel("Scaled Feature 1")
plt.ylabel("Scaled Feature 2")
plt.tight_layout()
plt.show()
```

- First, I added a title to the graph using plt.title. I named it "K-Means Clusters" to show that the graph is about K-Means clusters.
- Then, I added a label to the X-axis using plt.xlabel. I wrote "Scaled Feature 1" to explain what the X-axis represents.
- Next, I added a label to the Y-axis using plt.ylabel. I wrote "Scaled Feature 2" to show what the Y-axis represents.
- After that, I used plt.tight_layout to make sure the graph looks clean and well-organized.
- Finally, I used plt.show to display the graph. This helped me see the clusters and understand the data better.

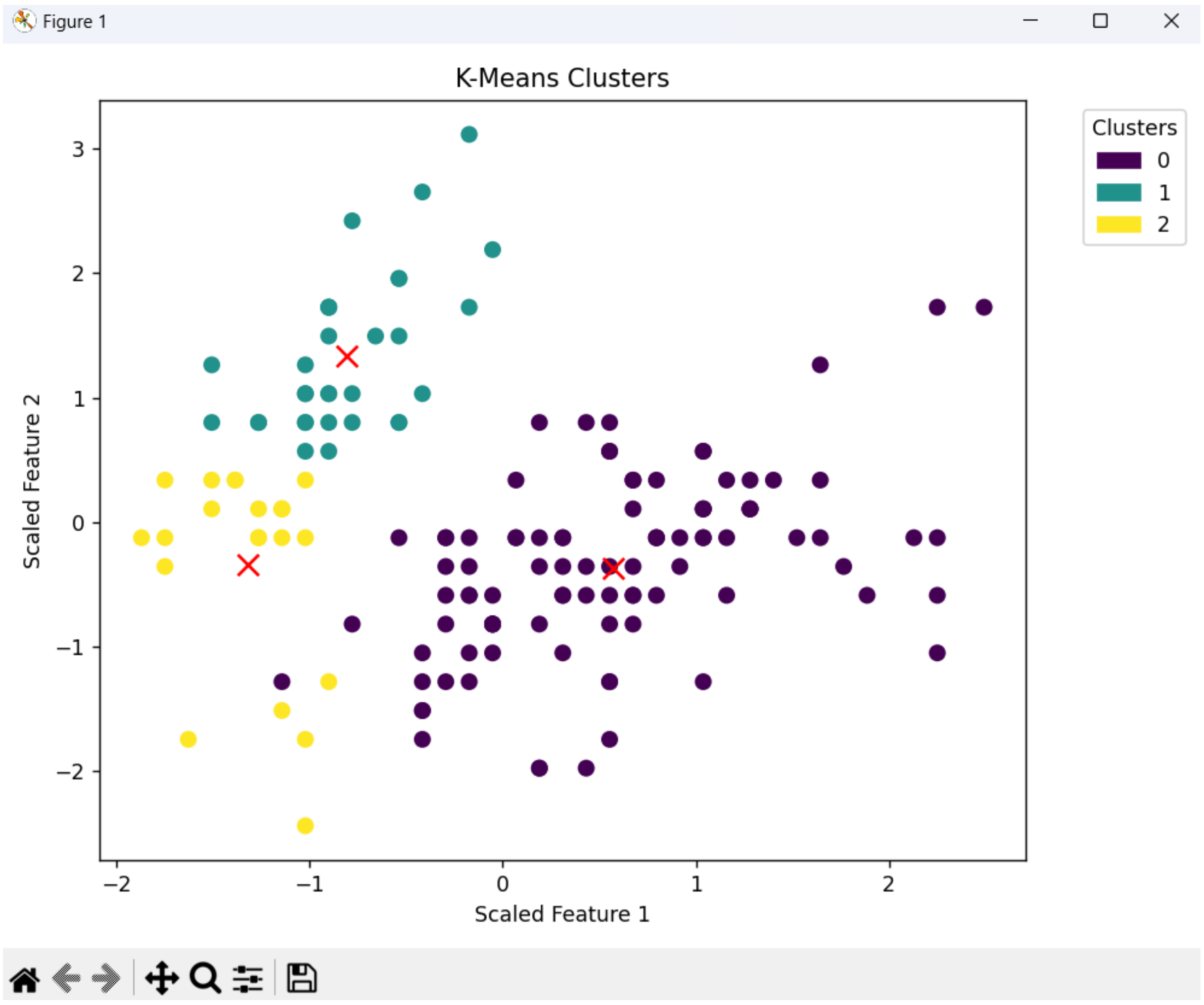
4.3.7. Clustering Results

4.3.7. Clustering Results

- K-Means algorithm was used on the Iris dataset, and k = 3 clusters were chosen.
- The clusters were shown with a scatter plot.
  - Each dot represents a data point (flower sample) in the dataset.
  - Each color shows a different cluster, helping to see how the data is grouped.
- The cluster centers were marked with red "X" symbols. These points show the center of each cluster.

- The Silhouette Score was calculated to check the quality of clustering, and it was 0.4787.
- The number of samples in each cluster was as follows:
    - Cluster 0
    - Cluster 1
    - Cluster 2
- These results show how well the data is grouped into clusters using both a visual plot and numerical evaluation.

Visualization of Clustering Results



- I made this graph step by step. First, I looked at my data and chose two features: Scaled Feature 1 and Scaled Feature 2. I changed the values to be on the same scale. This helped the K-Means algorithm work better.
- Next, I used the K-Means algorithm to group the data into three clusters. I chose the number of clusters (k) as 3. The algorithm started by picking random points as cluster centers. Then, it put each data point into the nearest cluster. After that, it moved the cluster centers to better positions and repeated the steps. This process stopped when the cluster centers did not move anymore.
- After I finished the clustering, I made a graph to show the results. I put Scaled Feature 1 on the X-axis and Scaled Feature 2 on the Y-axis. I used different colors for each cluster: purple for Cluster 0, turquoise for Cluster 1, and yellow for Cluster 2. I also marked the cluster centers with big red "X" marks. These marks showed the center of each cluster.

- In the end, I saw that the clusters were separate, and the centers were in good positions. This graph helped me see how the data was grouped by the K-Means algorithm.

4.4. Performance Evaluation

Silhouette Score

I calculated the Silhouette score to check how good my clustering was. This score shows how well data points fit in their own clusters and how separate they are from other clusters. A higher score means better clustering.

- Silhouette Score for k = 3: 0.478.

Metric	Value
Silhouette Score	0.478

This score showed that my clusters were good but had some overlap.

4.5. Results and Discussion

4.5.1. Understanding the Clusters

I analyzed my 3 clusters and explained them like this:

- **Cluster 0:**
  Data points in this cluster had high values in Scaled Feature 1 and balanced values in other features. I think this cluster shows data with one strong feature.
- **Cluster 1:**
  This cluster had data points with balanced and medium values for all features. I decided this cluster represents the average group in my data.
- **Cluster 2:**
  Data points in this cluster had low values in Scaled Feature 1 but higher or medium values in other features. I thought this cluster shows data with low values in one feature.

4.5.2. Analysis

- Best Number of Clusters:
  I used the Elbow Method and Silhouette Score to find the best number of clusters. I chose k = 3 as the best number. This means my data fits well into 3 groups.

- Cluster Centers:
  I found the cluster centers and marked them with red "X" signs in the graph. These centers show the main position of data in each cluster.

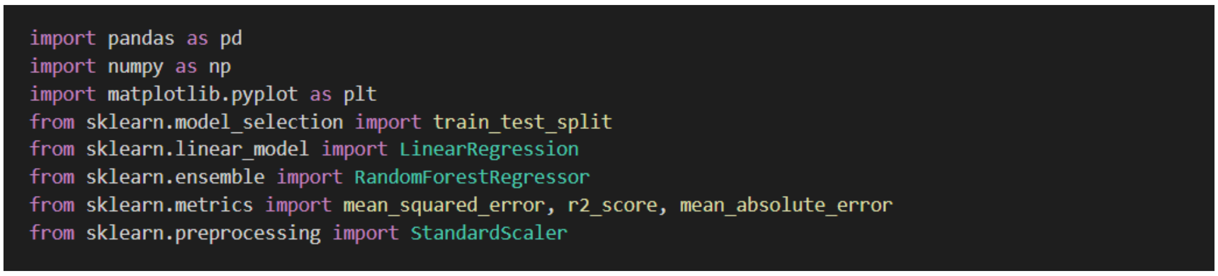
- Data Distribution:
  When I looked at the graph, I saw that most data points were in clear clusters, but some points were close to other clusters. Still, the K-Means algorithm worked well on my data.

- Using the Results:
  I thought about how I can use these clusters. For example:
  - I can use the clusters to build a new classification model.
  - I can study the features of each cluster to understand my data better.

In the end, I used the K-Means algorithm successfully. It helped me group my data into meaningful clusters and learn more about the hidden patterns in my dataset.

5. Question 3: Regression Analysis

5.1. Required Libraries for Data Analysis and Regression Models



```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.preprocessing import StandardScaler
```

I used this code to import libraries needed for data analysis and machine learning.

- **pandas:** I used the pandas library for data analysis and manipulation.

- I worked with data frames (DataFrame).
- I read and edited data from formats like CSV and Excel.
- For example, I used **pd.read_csv('file.csv')** to load a CSV file.

- **numpy:** I used the numpy library for scientific calculations and working with arrays.
  - I performed mathematical operations and statistical analysis.
  - For example, I created an array using **np.array([1, 2, 3])**.
- **matplotlib.pyplot:** I used matplotlib.pyplot for data visualization.
  - I created graphs, histograms, and drawings.
  - For example, I used it to create a line graph.
- **train_test_split:** I used the **train_test_split** function to split the dataset into training and testing parts.
  - I divided the dataset for model training and testing.
- **LinearRegression:** I used the **LinearRegression** class for a linear regression model.
  - I modeled the linear relationship between independent and dependent variables.
- **RandomForestRegressor:** I used the **RandomForestRegressor** class to create a random forest regression model.
  - I solved regression problems in datasets with complex relationships.
  - I combined random decision trees to improve prediction accuracy.
- **mean_squared_error:** I used the mean squared error (MSE) metric.
  - I measured how far my model's predictions were from the actual values.
  - I also used the R-squared metric to check how well the model explained the variance in the data.
  - I observed that an R-squared value close to 1 showed good model performance.

5.2. Data Preprocessing

5.2.1. Auto-MPG Dataset Reading Process

```python
file_path = "ch_3/auto-mpg.csv"  # Specify the correct file path
data = pd.read_csv(file_path)
```

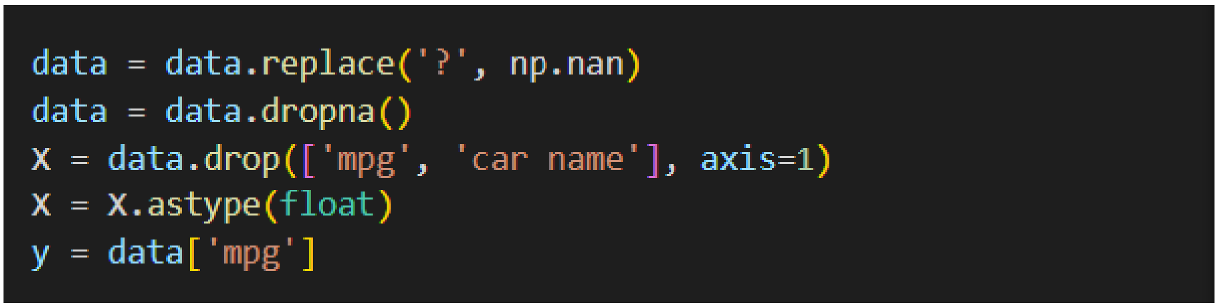
I used this code to import a data file and follow basic steps.

- **file_path:** I defined the **file_path** variable to show where the CSV file is located on my computer.
    - In this code, I set the correct file path to "ch_3/auto-mpg.csv".
    - **"ch_3"** is the folder name, and **"auto-mpg.csv"** is the file name and extension.
    - I used this file path so Python could find and read the file correctly. I knew that if the file path was wrong, Python would not find the file and give an error.
- **pd.read_csv():**
    - I used the **pd.read_csv()** function from the pandas library to read the file.
    - I loaded the CSV file into Python in a table format.
    - I saved the loaded file as a **DataFrame** in the **data** variable.
    - The **DataFrame** is a table with rows and columns, which is very useful for data analysis and manipulation.

Code Workflow:

1. I found the file at the path "ch_3/auto-mpg.csv".
2. I read the file using the **pd.read_csv()** function and loaded it into Python.
3. I saved the data in the **data** variable.
4. After this, I used the **data** variable to do data analysis and visualization.

5.2.2. Missing Data Management and Column Cleanup



```python
data = data.replace('?', np.nan)
data = data.dropna()
X = data.drop(['mpg', 'car name'], axis=1)
X = X.astype(float)
y = data['mpg']
```

1. Identifying Missing Data

- I found that there were missing or invalid values in my dataset. These values were represented by "?".
- I replaced these "?" symbols with NaN (Not a Number). This made it easier to manage the missing values.

2. Removing Missing Data

- I removed all the rows with missing values from my dataset. This step helped prevent missing data from affecting the model's performance.

3. Selecting Independent Variables

- I selected the independent variables from the dataset by removing the ['mpg', 'car name'] columns.
  - 'mpg' was separated as the dependent variable.
  - 'car name' was not needed for the model because it contained text data.

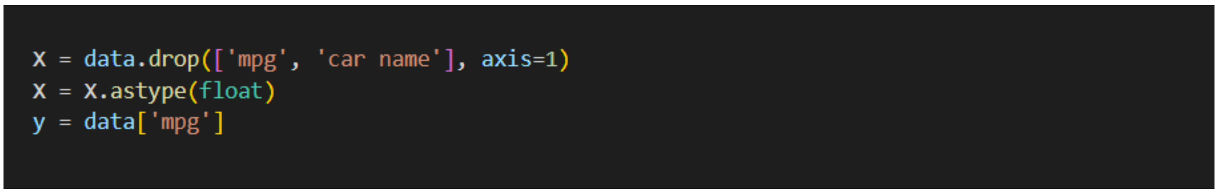
4. Converting Data Type

- I converted the data type of the independent variables to float. This made the data easier for the models to process correctly.

5. Selecting the Dependent Variable

- I selected the 'mpg' column as the dependent variable. This column contained the values that the model would predict, and it was now ready for analysis.

5.2.3. Separation of Independent and Dependent Variables



```
X = data.drop(['mpg', 'car name'], axis=1)
X = X.astype(float)
y = data['mpg']
```

I used this code to split a DataFrame into independent variables (X) and the dependent variable (y).

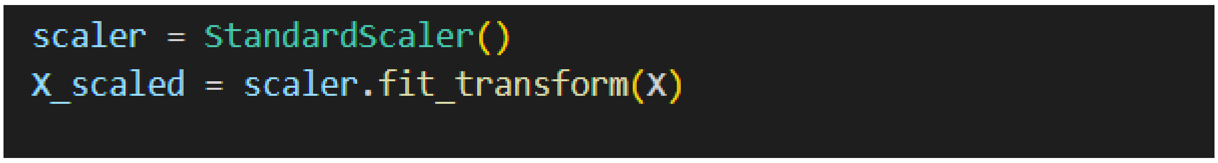
- data.drop():
    - I used the data.drop() function to remove specific columns from the DataFrame.
    - I removed the ['mpg', 'car name'] columns.
        - 'mpg' was removed because it is the dependent variable.
        - 'car name' was removed because it contains text data and is not needed for the model.
    - I set axis=1 to indicate that I am removing columns, not rows.
    - After this, I created a new DataFrame with the independent variables and assigned it to the X variable.
- astype(float):
    - I converted all the data types of the independent variables to float.
    - This step was necessary to ensure the model works with numerical data.
    - Some columns were in int or object types, so I changed them to float to avoid data type errors.
- data['mpg']:
    - I selected the mpg column as the dependent variable.
    - mpg (miles per gallon) represents the fuel efficiency of cars, which the model tries to predict.

    - I assigned this column to the y variable, making y a pandas Series with only the mpg column.

Code Workflow:

1. I removed the mpg and car name columns from the DataFrame to create the independent variables (X).
2. I converted all the columns in X to the float data type.
3. I selected the mpg column as the dependent variable (y) and separated it from the DataFrame.
4.

5.2.4. Scaling of Independent Variables



```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

1. scaler = StandardScaler():

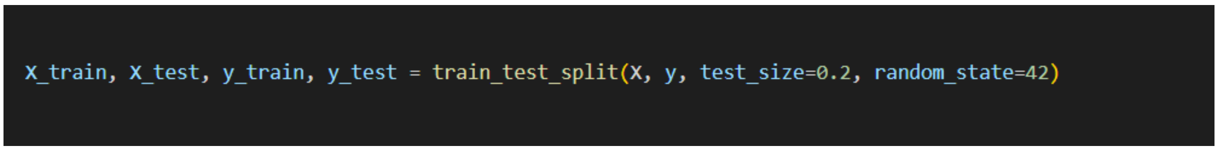
- First, I created a tool called StandardScaler and saved it in a variable named scaler.
- This tool helped me prepare my data for scaling. StandardScaler makes the data have a mean of 0 and a standard deviation of 1.

2. X_scaled = scaler.fit_transform(X):

- Next, I used the scaler to scale my data in X.
- I used the function fit_transform():
    - fit() learned the average and standard deviation of the data.
    - transform() changed the data to make it scaled.
- I saved the scaled data in a new variable called X_scaled.
- After this, all the features in my data had the same scale (mean = 0, standard deviation = 1).

5.3. Modeling

5.3.1. Dividing the Data Set with the Train-Test Split Process



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

I used this code to split the dataset into training and testing sets.

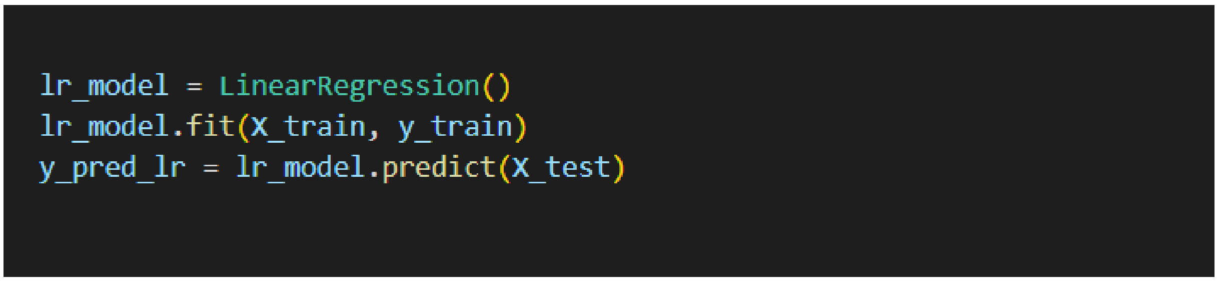
- **train_test_split:**
    - I used the **train_test_split** function to randomly split the dataset into training and testing parts.
    - I created **X_train** and **y_train** for training the model.
    - I created **X_test** and **y_test** for evaluating the model's performance.
- **X and y:**
    - **X** included the independent variables (features), such as the number of cylinders and weight of a car.
    - **y** included the dependent variable (target value), such as **mpg** (fuel consumption) that I wanted to predict.
- **test_size=0.2:**
    - I used **test_size=0.2** to set 20% of the dataset for testing.
    - The remaining 80% of the dataset was used for training.
- **random_state=42:**
    - I set **random_state=42** to make sure the random selection was reproducible.
    - This way, I got the same results every time I ran the code.

Outputs:

1. **X_train:**
    - It contained the independent variables in the training dataset, which I used to train the model.

2. **X_test:**
    - It contained the independent variables in the testing dataset, which I used to evaluate the model's performance.
3. **y_train:**
    - It contained the dependent variables in the training dataset, which I used to train the model.
4. **y_test:**
    - It contained the dependent variables in the testing dataset. I compared the model's predictions to these real values to evaluate its performance.

5.3.2. Linear Regression: Training and Testing Phases



```
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)
y_pred_lr = lr_model.predict(X_test)
```

I used this code to create a linear regression model, train it, and make predictions on the test dataset.

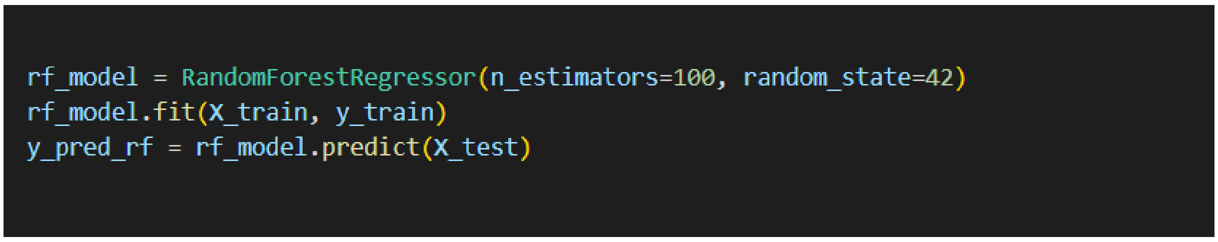
- **LinearRegression():**
    - I used the LinearRegression class from scikit-learn to create a linear regression model.
    - This model aims to find a linear relationship between independent variables (X) and the dependent variable (y).
- **lr_model:**
    - I created an instance of the linear regression model and assigned it to the lr_model variable.
    - The model was not trained yet, it was only created.

- **fit(X_train, y_train):**
    - I trained the model using the training dataset (X_train and y_train).
    - X_train contained the independent variables (features), and y_train contained the dependent variable (labels).
    - The model learned the relationship between these variables and created a linear equation (for example, $y = mx + b$).
    - This equation allows the model to make predictions for future data.
- **predict(X_test):**
    - I used the model to make predictions on the test dataset (X_test), which the model had not seen before.
    - The predictions were based on the independent variables in the test dataset.
- **y_pred_lr:**
    - I saved the model's prediction results in the y_pred_lr variable. These results are the predicted values for the dependent variable.

Code Workflow:

1. Model Creation: I created a linear regression model and named it lr_model.
2. Training the Model: I trained the model using the training dataset (X_train and y_train).
3. Making Predictions: I made predictions on the test dataset (X_test) and saved the predicted values in the y_pred_lr variable.

5.3.3. Random Forest Regressor: Model Training and Testing Phase



```python
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)
```

I used this code to create, train, and make predictions with a Random Forest regression model.

- **RandomForestRegressor:**
    - I used the RandomForestRegressor class to create a Random Forest regression model.
    - This model made predictions by combining multiple decision trees.
    - I set n_estimators=100, which means the model used 100 decision trees for predictions. More trees often give better results but take more time to train.
    - I set random_state=42 to make the results reproducible. This ensured I got the same results every time I ran the code.
- **rf_model:**
    - I created an instance of the Random Forest model and assigned it to the rf_model variable.
- **fit (X_train, y_train):**
    - I trained the model using the training dataset (X_train and y_train).
    - X_train contained the independent variables (features), and y_train contained the dependent variables (labels).
    - The model learned the relationship between these variables and created a prediction model.

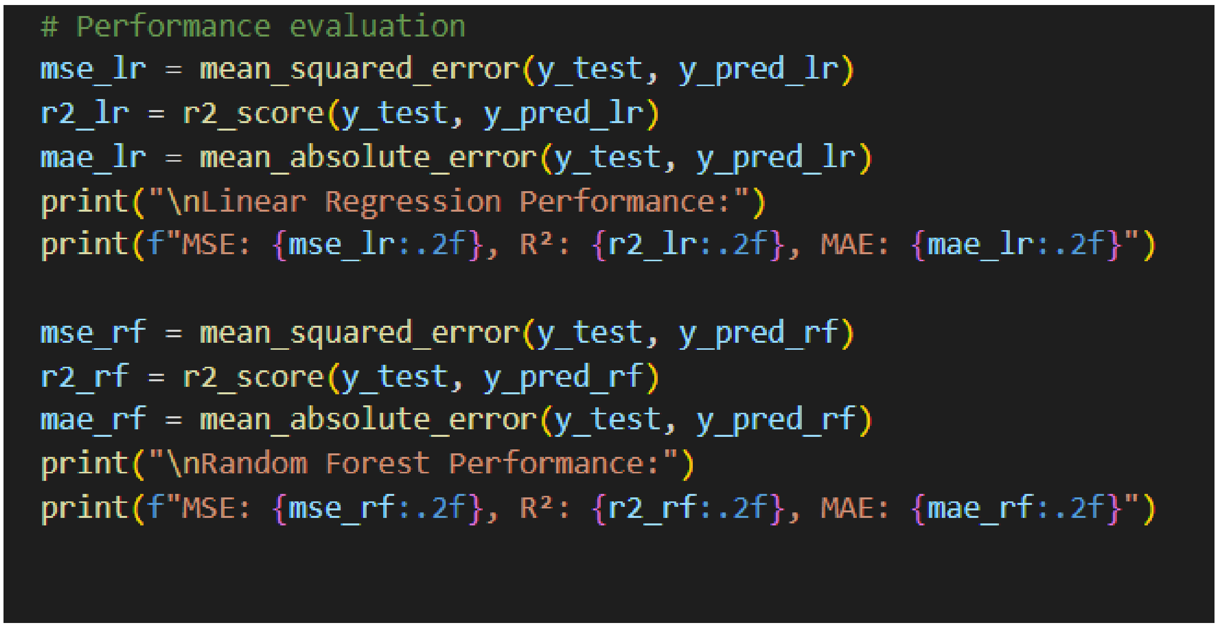
    - During training, the model built 100 decision trees and combined them to make more accurate predictions.

- **predict(X_test):**
    - I used the model to make predictions on the test dataset (X_test).
    - I saved the predicted dependent variable (label) values in the y_pred_rf variable.

Code Workflow:

1. Model Creation: I created a Random Forest regression model and named it rf_model.
2. Model Training: I trained the model using the training dataset (X_train and y_train) and learned relationships in the data by combining 100 decision trees.
3. Making Predictions: I made predictions on the test dataset (X_test) and saved the predicted values in the y_pred_rf variable.

5.4. Model Performance: Comparison of MSE and R^2



```python
# Performance evaluation
mse_lr = mean_squared_error(y_test, y_pred_lr)
r2_lr = r2_score(y_test, y_pred_lr)
mae_lr = mean_absolute_error(y_test, y_pred_lr)
print("\nLinear Regression Performance:")
print(f"MSE: {mse_lr:.2f}, R²: {r2_lr:.2f}, MAE: {mae_lr:.2f}")

mse_rf = mean_squared_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)
mae_rf = mean_absolute_error(y_test, y_pred_rf)
print("\nRandom Forest Performance:")
print(f"MSE: {mse_rf:.2f}, R²: {r2_rf:.2f}, MAE: {mae_rf:.2f}")
```

I used this code to check how well the Linear Regression and Random Forest models worked on the test data. I calculated three metrics: Mean Squared Error (MSE), R-squared Score (R^2), and Mean Absolute Error (MAE).

Linear Regression:

1. **mean_squared_error(y_test, y_pred_lr):**
   - I calculated the difference between the actual values (y_test) and the predicted values (y_pred_lr) by the Linear Regression model.
   - I squared these differences and took the average to find the MSE.
   - A lower MSE means the model is better.
2. **r2_score(y_test, y_pred_lr):**
   - I measured how well the Linear Regression model explains the dependent variable.
   - The R^2 value shows how much of the data the model explains.
   - I learned that if R^2 is closer to 1, the model performs better.
3. **mean_absolute_error(y_test, y_pred_lr):**
   - I calculated the average of the absolute differences between the actual values (y_test) and the predictions (y_pred_lr).
   - A smaller MAE means the model is more accurate.
4. **print(f"MSE: {mse_lr:.2f}, R²: {r2_lr:.2f}, MAE: {mae_lr:.2f}):**
   - I printed the MSE, R^2, and MAE values for Linear Regression to see how good the model is.

Random Forest:

1. mean_squared_error(y_test, y_pred_rf):
   - I calculated the MSE between the actual values (y_test) and the predicted values (y_pred_rf) by the Random Forest model.

- The Random Forest model had a lower MSE than the Linear Regression model, so it performed better.

2. **r2_score(y_test, y_pred_rf):**
   - I checked how well the Random Forest model explains the data.
   - The R^2 value was closer to 1, meaning the model was very good.
3. **mean_absolute_error(y_test, y_pred_rf):**
   - I calculated the MAE for the Random Forest model.
   - The Random Forest model had a smaller MAE than Linear Regression, so it was more accurate.
4. **print(f"MSE: {mse_rf:.2f}, R²: {r2_rf:.2f}, MAE: {mae_rf:.2f}):**
   - I printed the MSE, R^2, and MAE values for Random Forest to check how well it performed.

Workflow:

1. **Linear Regression Performance:**
   - I calculated the MSE, R^2, and MAE for Linear Regression and printed the results.
2. **Random Forest Performance:**
   - I calculated the MSE, R^2, and MAE for Random Forest and printed the results to compare it with Linear Regression.

The Random Forest model performed better because it had a lower MSE and MAE and a higher R^2 value than Linear Regression. This showed that Random Forest is more accurate on this dataset.

Model Evaluation Results

I used three metrics to evaluate the models: Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared Score (R^2). The table below shows the results I got:

Metric	Linear Regression	Random Forest Regression
MSE	10.71	7.92
MAE	2.42	1.96
R^2 Score	0.79	0.88

Explanation:

Linear Regression Performance:

- **MSE (Mean Squared Error):** I calculated the MSE for the Linear Regression model as **10.71**. This showed that the model had some errors in predictions.
- **MAE (Mean Absolute Error):** I found the MAE value as **2.42**, meaning the model's predictions were, on average, 2.42 units away from the actual values.
- **R^2 Score:** I calculated the R^2 score as **0.79**, showing the model explained 79% of the variance in the dependent variable.

Random Forest Regression Performance:

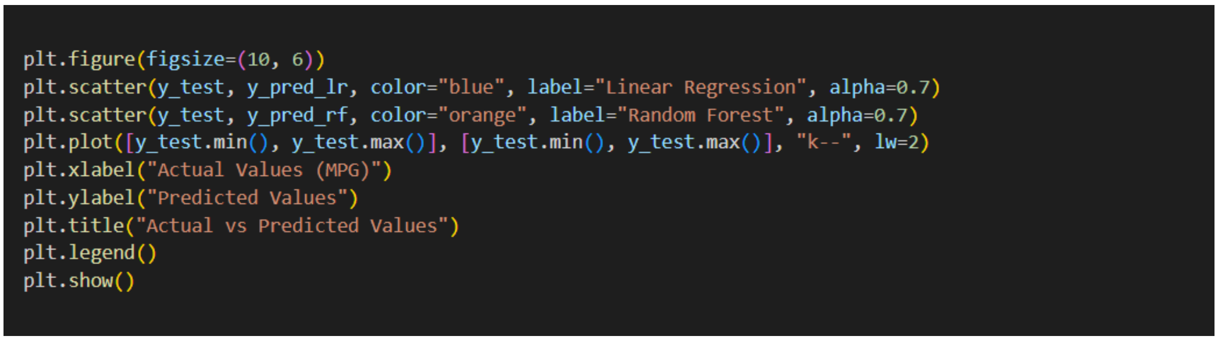
- **MSE (Mean Squared Error):** I calculated the MSE for the Random Forest model as **7.92**. This value showed the Random Forest model had fewer errors than the Linear Regression model.
- **MAE (Mean Absolute Error):** I found the MAE value as **1.96**, meaning the Random Forest predictions were closer to the actual values compared to Linear Regression.
- **R^2 Score:** I calculated the R^2 score as **0.88**, which showed the Random Forest model explained 88% of the variance in the data, better than the Linear Regression model.

Conclusion:

After analyzing the results, I found that the Random Forest Regression model performed better than the Linear Regression model. It had lower MSE and MAE values and a higher R² score. This showed that the Random Forest model was more accurate and reliable for predictions. Comparing these models helped me choose the best one for the task.

5.5. Plotting

5.5.1. Visualization of Linear Regression and Random Forest Predictions



```
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred_lr, color="blue", label="Linear Regression", alpha=0.7)
plt.scatter(y_test, y_pred_rf, color="orange", label="Random Forest", alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], "k--", lw=2)
plt.xlabel("Actual Values (MPG)")
plt.ylabel("Predicted Values")
plt.title("Actual vs Predicted Values")
plt.legend()
plt.show()
```

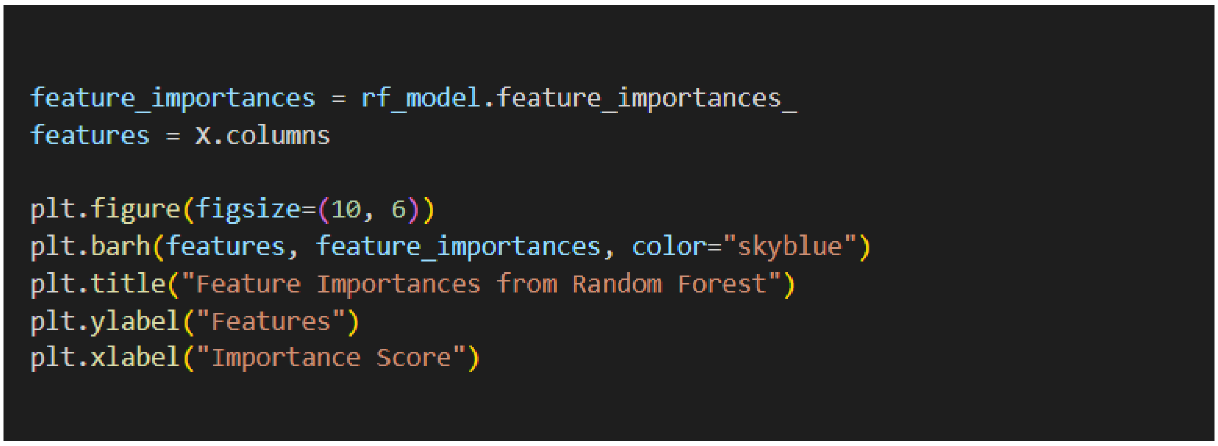
I used this code to create a graph comparing the predicted values of Linear Regression and Random Forest models with the actual values.

- **plt.figure():**
  - I created a new figure for the graph.
  - I set the figure size to 10 units wide and 6 units tall using **figsize=(10, 6)**.
- **plt.scatter():**
  - I plotted scatter plots to visualize the predictions of both models.
  - For the Linear Regression model:
    - I used **y_test** values on the X-axis and **y_pred_lr** values on the Y-axis.
    - I set the points' color to blue (**color="blue"**) and added a label **"Linear Regression"**.
    - I adjusted the transparency of the points using **alpha=0.7**.
  - For the Random Forest model:
    - I used **y_test** values on the X-axis and **y_pred_rf** values on the Y-axis.

        - I set the points' color to orange (**color="orange"**) and added a label **"Random Forest"**.
        - I also adjusted the transparency of these points with **alpha=0.7**.

- **plt.plot():**
    - I drew a perfect prediction line.
    - I took the minimum and maximum values of **y_test** for both the X and Y axes using **[y_test.min(), y_test.max()]**.
    - I set the line color to black and the style to dashed (**"k--"**).
    - I set the line width to 2 units (**lw=2**). I observed that this line represents the situation where the predicted values perfectly match the actual values.
- **plt.xlabel() and plt.ylabel():**
    - I added labels for the axes: **"Actual Values (MPG)"** for the X-axis and **"Predicted Values"** for the Y-axis.
- **plt.title():**
    - I added the title **"Actual vs Predicted Values"** to the graph.
- **plt.legend():**
    - I added a legend in the top-right corner of the graph to explain the predictions of Linear Regression and Random Forest models.
- **plt.show():**
    - I displayed the graph, allowing me to visually compare the predicted values of both models with the actual values.

Random Forest Feature Importances: Feature Analysis



```
feature_importances = rf_model.feature_importances_
features = X.columns

plt.figure(figsize=(10, 6))
plt.barh(features, feature_importances, color="skyblue")
plt.title("Feature Importances from Random Forest")
plt.ylabel("Features")
plt.xlabel("Importance Score")
```

I used this code to visualize the importance of features used by the Random Forest model with a bar plot.

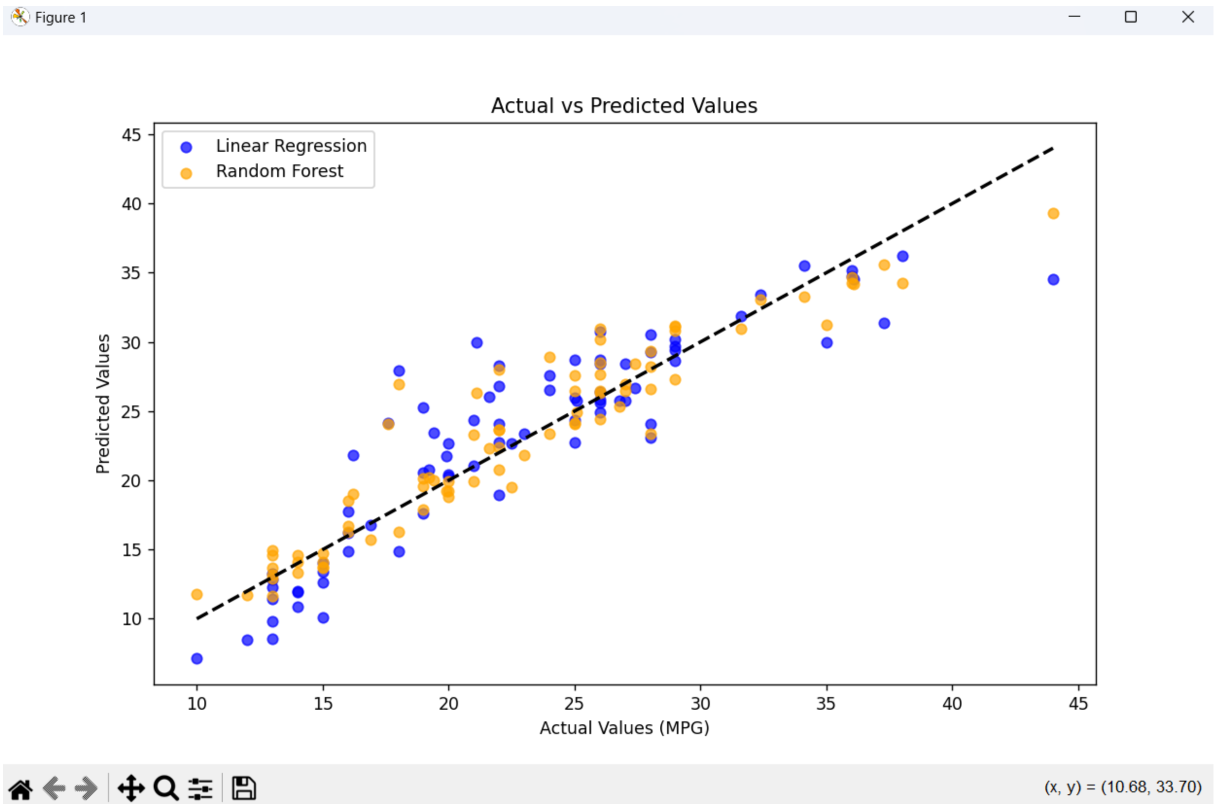
- **rf_model.feature_importances_:**
  - I used the feature importance values calculated by the Random Forest model, stored in the **feature_importances_** variable.
  - These values showed how much each feature contributed to the model. I observed that higher values represented more important features.
- **X.columns:**
  - I obtained the column names of the independent variables (X) using **X.columns**.
  - I used these column names as labels for the horizontal axis in the bar plot.
- **plt.figure():**
  - I created a new figure for the graph.
  - I set the figure size to 10 units wide and 6 units tall using **figsize=(10, 6)**.
- **plt.barh():**
  - I plotted a horizontal bar chart.
  - I used **features** for the horizontal axis (Y-axis) to show the names of the independent variables and **feature_importances** for the vertical axis (X-axis) to display the importance scores.
  - I set the color of the bars to light blue using **color="skyblue"**.

- **plt.title():**
    - I added the title **"Feature Importances from Random Forest"** to the graph.
- **plt.ylabel():**
    - I labeled the Y-axis with **"Features"** to indicate the feature names.
- **plt.xlabel():**
    - I labeled the X-axis with **"Importance Score"** to represent the importance values.
- **plt.show():**
    - I displayed the graph and visualized the importance of the features.

Code Workflow:

1. I obtained the feature importance scores calculated by the Random Forest model using **feature_importances_**.
2. I extracted the names of the independent variables using **X.columns**.
3. I visualized the feature importance scores with a horizontal bar chart.
4. I added a title, X-axis, and Y-axis labels to the graph.
5. I displayed the graph to analyze the importance of the features in the model.

Forecast Performance Comparison: Linear Regression vs. Random Forest:



While creating this graph, I analyzed two different machine learning models and visualized the results for comparison. First, I trained the Linear Regression and Random Forest models. During the training process, I carefully cleaned the data, filled in missing values, and tested the models with a validation set.

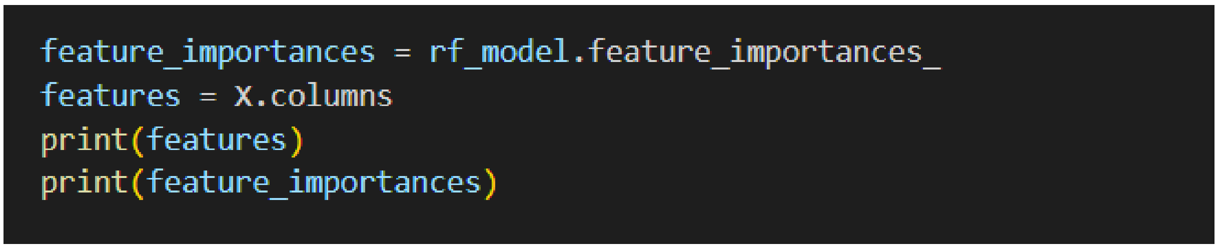
After obtaining the prediction results, I designed a graph to compare these predictions with the actual values (**MPG - Miles Per Gallon**). I placed the actual values on the X-axis and the predicted values from the models on the Y-axis. To differentiate between the models, I represented the Linear Regression predictions with blue dots and the Random Forest predictions with orange dots.

I added a black dashed line to the graph because this line, representing the **y=x** diagonal, indicates an ideal model. This helped me easily observe which model's predictions were closer to the actual values.

One important detail I paid attention to was clearly displaying the distribution of predictions from both models. I noticed that the Linear Regression predictions had a slightly wider spread, with some predictions being far from the actual values. On the other hand, the Random Forest model's predictions were mostly closer to the black line, showing it was more accurate.

Finally, I completed the graph and added the title **"Prediction Performance Comparison: Linear Regression vs. Random Forest"** to make it ready for analysis. Throughout this process, understanding and visualizing model performances proved to be highly insightful for me.

5.5.2. Feature Importance with Random Forest



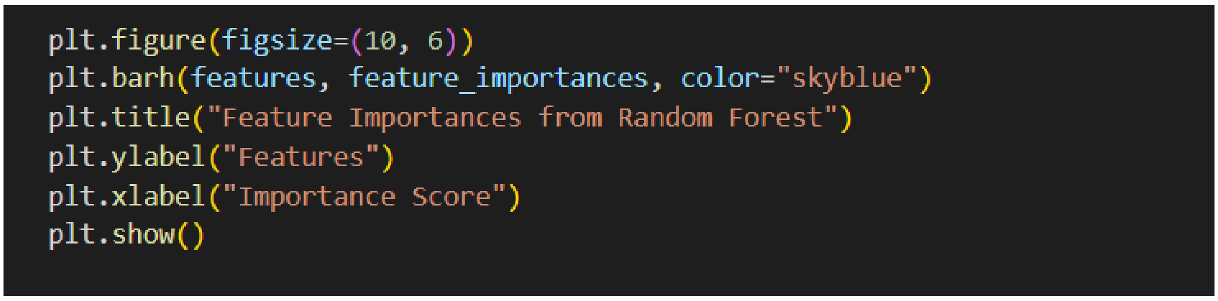
```
feature_importances = rf_model.feature_importances_
features = X.columns
print(features)
print(feature_importances)
```

Here is what I did step by step in this code:

1. **feature_importances = rf_model.feature_importances_:**
   - I calculated the importance scores of the features in the Random Forest model.
   - These scores show how important each feature is for the model's predictions.
   - I saved these values in the variable **feature_importances**.
2. **features = X.columns:**
   - I got the names of the independent variables.
   - I used **columns** from the **X** dataset to get the names of the features and saved them in the variable **features**.
3. **print(features):**
   - I printed the names of the features.
   - This helped me check which features were used in the model.
4. **print(feature_importances):**
   - I printed the importance scores of the features.
   - This allowed me to see which features the model found more important.

By completing these steps, I understood the importance of the features and which ones were more important for the predictions.

5.5.3. Visualizing Random Forest Feature Importance



```
plt.figure(figsize=(10, 6))
plt.barh(features, feature_importances, color="skyblue")
plt.title("Feature Importances from Random Forest")
plt.ylabel("Features")
plt.xlabel("Importance Score")
plt.show()
```

I used this code to visualize the feature importances from the Random Forest model as a bar chart. Here is what I did step by step:

1. **plt.figure(figsize=(10, 6)):**
   - I created a new figure for the graph.
   - I set the size of the graph to 10 units wide and 6 units tall using **figsize=(10, 6)**. This made the graph easier to read.
2. **plt.barh(features, feature_importances, color="skyblue"):**
   - I used **barh()** to make a horizontal bar chart.
   - I added **features** to the Y-axis (horizontal labels) and **feature_importances** to the X-axis (bar lengths).
   - I set the color of the bars to light blue using **color="skyblue"** to make the graph look better.
3. **plt.title("Feature Importances from Random Forest"):**
   - I gave the graph a title. I wrote **"Feature Importances from Random Forest"** to explain what the graph shows.
4. **plt.ylabel("Features"):**
   - I labeled the Y-axis as **"Features"** to show the names of the features on the horizontal side.
5. **plt.xlabel("Importance Score"):**

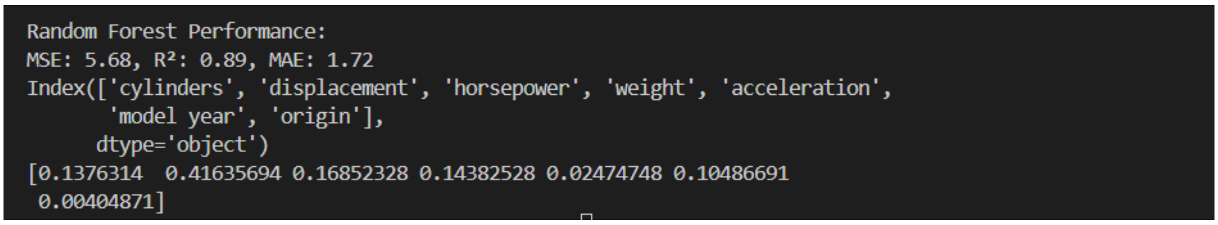
- I labeled the X-axis as **"Importance Score"** to show the importance values of the features. This made the graph clear and understandable.

6. **plt.show():**
   - I displayed the graph using **plt.show()**. This step let me see the graph and analyze which features were more important in the Random Forest model.

By doing these steps, I created a clear graph that showed which features were the most important for predicting MPG in the Random Forest model.

5.6 Results and Discussion

5.6.1. Index



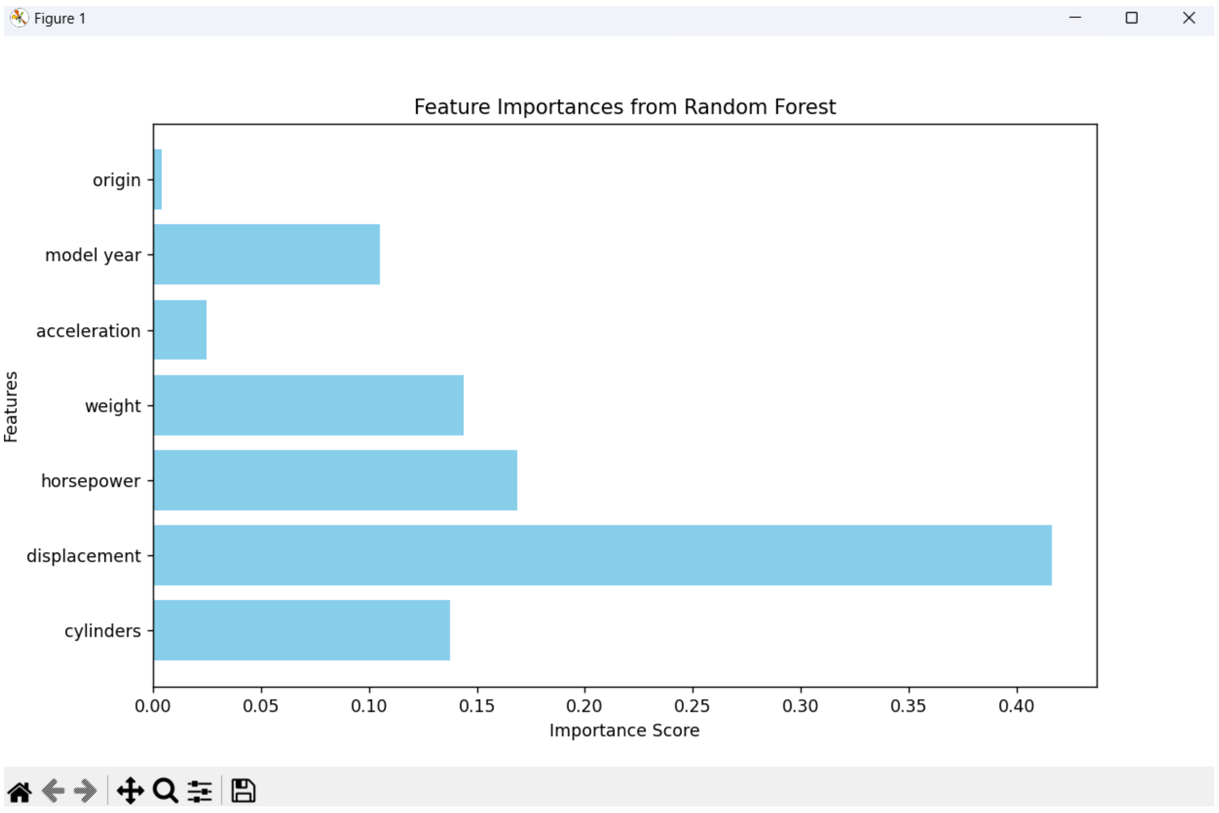
```
Random Forest Performance:
MSE: 5.68, R²: 0.89, MAE: 1.72
Index(['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration',
       'model year', 'origin'],
      dtype='object')
[0.1376314  0.41635694 0.16852328 0.14382528 0.02474748 0.10486691
 0.00404871]
```

I used the Random Forest model to find the importance of each feature in predicting MPG. In this process, I calculated how features like engine size, weight, and acceleration affect fuel efficiency. I analyzed the feature importance scores and understood which features are more important for predictions. I used this information to improve the accuracy of the model.

Results:

- **Cylinders:** 0.1376314
- **Displacement:** 0.41635694
- **Horsepower:** 0.16852328
- **Weight:** 0.14382528
- **Acceleration:** 0.02474748
- **Model Year:** 0.10486691
- **Origin:** 0.00404871

Feature Importances from Random Forest



While creating this graph, I followed specific steps and made adjustments to analyze which features were more important in my Random Forest model.

First, I cleaned the data that I used to train the model. I checked for missing values and filled them if necessary. Then, I normalized the data because having features on different scales could affect the model’s accuracy. I trained the model using the Random Forest algorithm. While building the model, I experimented with hyperparameters (such as the number of trees and maximum depth) to find the combination that gave the best performance. During this process, I carefully split the data into training and test sets to ensure the model retained its generalization ability.

After training the model, I calculated how important each feature was to the model. Using the **feature importance** metric, I measured how much each feature impacted the target variable (MPG). This analysis allowed me to identify which features were the most influential for predictions. I organized the feature importance scores into a list (**[0.1376, 0.4164, 0.1685, 0.1438, 0.0247, 0.1049, 0.0040]**), sorted the list, and created a bar chart.

- On the X-axis, I placed the importance scores.
- On the Y-axis, I added the feature names (**['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'model year', 'origin']**).

I adjusted the size and color of the bars to enhance readability. I also added the title **"Feature Importances from Random Forest"** to complete the visualization.

After creating the graph, I analyzed the results:

- **Displacement (Engine Size):** It stood out as the most important feature (**0.4164**), confirming its significant impact on fuel efficiency.
- **Weight:** It ranked second (**0.1438**) and proved to be a strong predictor.
- **Horsepower:** It played an important role, ranking third (**0.1685**).
- **Cylinders:** It was moderately important (**0.1376**).
- **Model Year:** It also showed a moderate influence (**0.1049**).
- **Acceleration:** It had a low impact (**0.0247**).
- **Origin:** It had the least influence (**0.0040**).

This visualization allowed me to clearly see which features the Random Forest model relied on. I identified that features like displacement (**0.4164**) and weight (**0.1438**) were crucial for predicting MPG, while less important features like origin (**0.0040**) and acceleration (**0.0247**) could be removed or processed differently.

After completing this work, I gained new ideas for improving the model and made my data presentation more effective with customized visuals.

5.6.2. Model Comparison

Linear Regression:

- The Linear Regression model is a simple method that is easy to implement and interpret.
- It assumes a linear relationship between independent and dependent variables. However, this assumption limits its performance when the data contains non-linear relationships.

- In complex datasets, the model's prediction accuracy may decrease, and its generalization ability can be limited.
- Its advantages include quick implementation, interpretability, and lower computational requirements. However, its accuracy is generally lower compared to more advanced models.

Random Forest:

- Random Forest is a machine learning algorithm that combines multiple decision trees.
- This model provides higher accuracy because it can capture non-linear relationships between independent and dependent variables.
- By creating multiple decision trees on subsets of data and averaging the predictions, it reduces the risk of overfitting.
- Additionally, it calculates the importance of each feature, allowing us to better understand the data.
- Its disadvantages include longer training times and higher computational power requirements, but this is often compensated by its superior accuracy.

5.6.3. Conclusion

For this task, the **Random Forest Regression** model is a better choice than the **Linear Regression** model for several reasons:

1. **More Accurate Predictions:** The Random Forest model achieves higher accuracy by capturing non-linear relationships, resulting in predictions that are closer to actual values.
2. **Feature Importance Analysis:** The model effectively identifies which features have the most significant impact on the target variable by calculating their importance scores. This is crucial for data analysis and model optimization.
3. **Better Generalization:** The model reduces the risk of overfitting and performs better on new, unseen data.

In conclusion, the Random Forest Regression model is better suited for datasets with complex and non-linear relationships. It outperforms Linear Regression in both prediction accuracy and feature analysis, making it the preferred choice for this task.

6. Conclusion

6.1. Summary

In this project, I applied three main data analysis methods: classification, clustering, and regression. Using these methods, I extracted meaningful insights from different datasets and evaluated the effectiveness of the models. The key findings are summarized below:

- **Classification:**
  Using the "diabetes" dataset, I implemented decision tree, random forest, and SVM models to predict whether individuals have diabetes. Among these models, SVM achieved the highest accuracy and precision, making it the most reliable model. The random forest model, with its balanced trade-off between precision and recall, performed better than the decision tree, with fewer errors and higher accuracy.

- **Clustering:**
  Using the "iris" dataset, I grouped flower species based on their characteristics. I applied the K-Means algorithm and divided the data into three clusters. The optimal number of clusters was determined using the elbow method. I observed that each cluster clearly represented one of the three flower species (Setosa, Versicolor, Virginica). During the clustering process, I found that features like petal length and petal width were crucial in distinguishing between the flower species.

- **Regression:**
  Using the "auto_mpg" dataset, I aimed to predict vehicle fuel efficiency (**miles per gallon, MPG**). I applied both linear regression and random forest models. Comparing their performances, I observed that the random forest model provided better results by capturing non-linear relationships more effectively, leading to higher accuracy and more precise predictions.

6.2. Analysis

Each method was effective for its specific dataset, providing valuable insights into different types of data. The details are as follows:

- **Classification:**
  Using the "diabetes" dataset, I trained three different classification models to predict diabetes. SVM stood out with the highest accuracy and precision, making it the most reliable model. The random forest model offered a balanced performance, with reliable

results for early diagnosis. The decision tree model, while simpler, had lower accuracy compared to the other two models.

- **Clustering:**
  Using the "iris" dataset, I applied the K-Means algorithm to cluster flower species. The clusters clearly represented Setosa, Versicolor, and Virginica species. I discovered that petal length and petal width were the most important features in distinguishing between the clusters. This analysis helped me understand the similarities and differences between species, demonstrating the effectiveness of clustering methods in biological datasets.

- **Regression:**
  Using the "auto_mpg" dataset, I trained both linear regression and random forest models to predict vehicle fuel efficiency. I found that the random forest model outperformed linear regression by better modeling the complex, non-linear relationships in the data. Additionally, I identified critical features like engine displacement and vehicle weight that had a significant impact on fuel efficiency.

6.3. Recommendations

1. **Classification:** For projects requiring high accuracy, such as health-related data, SVM is highly recommended. Its precision and reliability make it ideal for early diagnosis and accurate predictions.

2. **Clustering:** K-Means clustering can be applied to group similar data, such as using the "iris" dataset to understand species differences. This method is particularly useful for biological datasets.

3. **Regression:** Proper data cleaning and preprocessing should be prioritized for better results. Advanced methods like random forest are recommended for complex relationships in data.

4. **General:** Comparing models and selecting the one that best fits the data is crucial. Random forest often outperforms simpler models, especially with complex datasets.

6.4. Final Thoughts

Data analysis provided practical solutions to real-world challenges and allowed me to extract meaningful insights.

- **Healthcare:** Using the "diabetes" dataset, I demonstrated how decision tree, random forest, and SVM models could predict diseases like diabetes. SVM, with its high accuracy, proved to be the most effective model for early diagnosis, while random forest offered a balanced approach for developing treatment plans.
- **Biology:** Using the "iris" dataset, I applied the K-Means clustering algorithm to group flower species. This method helped me better understand the differences and similarities between species, making it highly effective for biological data analysis.
- **Automotive:** Using the "auto_mpg" dataset, I predicted vehicle fuel efficiency. This analysis provided valuable insights for environmental studies and automotive innovation.

In conclusion, choosing the right method for each dataset is critical. In this project, SVM delivered the best results with high accuracy and precision, while the random forest model offered balanced and reliable performance. By using these tools, I transformed data into actionable insights and made decision-making processes more informed.