RECEP TAYYIP ERDOGAN UNIVERSITY
FACULTY OF ENGINEERING AND ARCHITECTURE
COMPUTER ENGINEERING DEPARTMENT

Data Mining

Homework

Student Name Surname

Eren ŞİŞMAN

Student No

201401021

Question 1

Name Surname

Eren ŞİŞMAN

Student No

201401021

Project Requirements

Using a publicly available dataset of your choice (e.g., from Kaggle or UCI Machine Learning Repository), and apply a decision tree classifier to predict a target variable. Split your dataset into training, validation (if it is necessary) and test sets and evaluate the model's performance using accuracy, precision, recall, and F1 score. Compare the decision tree's performance with at least one other classifier (e.g., random forest, support vector machine etc).

Dataset Selection and Overview

For this assignment, the Palmer Penguins dataset was selected. This dataset is publicly available and designed to classify penguin species (Adelie, Chinstrap, and Gentoo) based on physical measurements and categorical variables. It is well-suited for classification tasks.

Dataset Features:
- Features: bill length, bill depth, flipper length, body mass, and sex.
- Target Variable: species (Adelie, Chinstrap, Gentoo).

The task involves applying a decision tree classifier, comparing it with a random forest classifier, and evaluating their performance with accuracy, precision, recall, and F1 score.

Why This Dataset?

The **Palmer Penguins dataset** was chosen because:

1. **Multi-Class Problem**: It has three species (*Adelie*, *Chinstrap*, *Gentoo*), ideal for classification tasks.
2. **Feature Variety**: Includes both numerical (e.g., bill_length_mm, body_mass_g) and categorical features (sex).
3. **Balanced Classes**: Species are well-distributed, preventing bias in model performance.
4. **Real-World Relevance**: Features like bill dimensions and body mass are meaningful in ecological studies.
5. **Manageable Size**: Small and easy to preprocess, making it ideal for quick analysis.

The goal of this task is to classify penguin species based on physical measurements and demographic features using **Decision Tree** and **Random Forest** models. By evaluating and comparing these models, we aim to understand how feature interactions and hyperparameter tuning impact classification accuracy and overall performance.

Code Explanation

1. Necessary Libraries

```
# 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.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

# import the dataset
from palmerpenguins import load_penguins
```

- Libraries used:
  - pandas and numpy for data manipulation.

- sklearn for machine learning models and performance evaluation.
- seaborn and matplotlib for visualization.
- palmerpenguins to load the dataset.

2. Data Loading and Preprocessing

```
# load the dataset
penguins = load_penguins()
print(penguins.head())
```

- Data Loading:
  - The dataset is loaded using `load_penguins()` and displayed using `head()` to examine its structure.

```
# select the necessary columns
penguins = penguins[["species", "bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g", "sex"]]

# check for missing values
imputer = SimpleImputer(strategy="mean")
penguins.iloc[:, 1:-1] = imputer.fit_transform(penguins.iloc[:, 1:-1])

# fill missing
penguins["sex"].fillna("Unknown", inplace=True)

# convert categorical variables to numerical
penguins = pd.get_dummies(penguins, columns=["sex"], drop_first=True)
penguins["species"] = penguins["species"].map({"Adelie": 0, "Chinstrap": 1, "Gentoo": 2})
```

- Preprocessing:
  - Feature selection: Only relevant columns are retained.
  - Handling missing values:
    - Numerical columns are filled with the mean value.
    - Missing values in the categorical column `sex` are filled with "Unknown".
  - Encoding categorical variables:
    - `sex` is converted to dummy variables.
    - The target variable `species` is mapped to numerical values.

```
# split the data into features and target
X = penguins.drop(columns=["species"])
y = penguins["species"]

# split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

- Data splitting and scaling:
  - Data is split into training (80%) and test (20%) sets.
  - Numerical features are standardized using `StandardScaler`.

3. Model Training

```
# first model training
# decision tree model
dt_model_1 = DecisionTreeClassifier(random_state=42)
dt_model_1.fit(X_train, y_train)
y_pred_dt_1 = dt_model_1.predict(X_test)

# random forest model
rf_model_1 = RandomForestClassifier( n_estimators=5, max_depth=3 ,random_state=42)
rf_model_1.fit(X_train, y_train)
y_pred_rf_1 = rf_model_1.predict(X_test)

# second model training
# decision tree model
dt_model_2 = DecisionTreeClassifier(max_depth=5, min_samples_split=10, random_state=42)
dt_model_2.fit(X_train, y_train)
y_pred_dt_2 = dt_model_2.predict(X_test)

# random forest model
rf_model_2 = RandomForestClassifier(n_estimators=200, max_depth=15, random_state=42)
rf_model_2.fit(X_train, y_train)
y_pred_rf_2 = rf_model_2.predict(X_test)
```

- Decision Tree:
  - First Training: Default parameters were used.
  - Second Training: Optimized hyperparameters were applied (max_depth=5, min_samples_split=10).
- Random Forest:
  - First Training: Hyperparameters were applied (n_estimators=5, max_depth=3).
  - Second Training: Optimized hyperparameters were applied (n_estimators=200, max_depth=10).

4. Model Evaluation

```python
# evaluate model performance
def evaluate_model(y_test, y_pred, model_name, training_phase):
    print(f"{model_name} - {training_phase} training")
    print("accuracy:", accuracy_score(y_test, y_pred))
    print("precision:", precision_score(y_test, y_pred, average='weighted'))
    print("recall:", recall_score(y_test, y_pred, average='weighted'))
    print("f1 score:", f1_score(y_test, y_pred, average='weighted'))
    print("\nclassification report:\n", classification_report(y_test, y_pred))
```

- A custom function `evaluate_model` computes performance metrics:
  - Accuracy
  - Precision
  - Recall
  - F1 Score
  - Classification Report

5. Visualization

```python
# confusion matrix
def plot_confusion_matrix(y_test, y_pred, model_name, training_phase):
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(6, 4))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Adelie", "Chinstrap", "Gentoo"], yticklabels=["Adelie", "Chinstrap", "Gentoo"])
    plt.title(f"{model_name} - {training_phase} training")
    plt.ylabel("actual")
    plt.xlabel("predicted")
    plt.show()
```

The confusion matrix provides a clear visualization of model predictions versus actual labels.

Results

Performance Metrics:

1. **Decision Tree:**

- In the first training, the decision tree performed well but not perfectly. This was due to the default hyperparameters, which allowed some minor misclassifications.
- In the second training, hyperparameter optimization (limiting the depth of the tree and controlling the minimum samples per split) improved the model's ability to generalize. The

decision tree achieved more accuracy in this phase, demonstrating the importance of proper parameter tuning.

2. **Random Forest:**

- In the first training, the random forest model had intentionally limited capacity due to the hyperparameter settings (n_estimators=5 and max_depth=3). This led to underfitting, with lower accuracy and weaker overall performance.
- In the second training, increasing the number of estimators (n_estimators=200) and depth (max_depth=10) allowed the random forest to model the data more effectively. This resulted in a significant performance boost, matching the near-optimal behavior expected from a robust ensemble model.

```
decision tree - first training
accuracy: 0.9855072463768116
precision: 0.986086956521739
recall: 0.9855072463768116
f1 score: 0.9855401097637122

classification report:
              precision    recall  f1-score   support

           0       1.00      0.97      0.98        32
           1       1.00      1.00      1.00        13
           2       0.96      1.00      0.98        24

    accuracy                           0.99        69
   macro avg       0.99      0.99      0.99        69
weighted avg       0.99      0.99      0.99        69

random forest - first training
accuracy: 0.9565217391304348
precision: 0.9646739130434783
recall: 0.9565217391304348
f1 score: 0.9577014771302873

classification report:
              precision    recall  f1-score   support

           0       1.00      0.91      0.95        32
           1       0.81      1.00      0.90        13
           2       1.00      1.00      1.00        24

    accuracy                           0.96        69
   macro avg       0.94      0.97      0.95        69
weighted avg       0.96      0.96      0.96        69
```

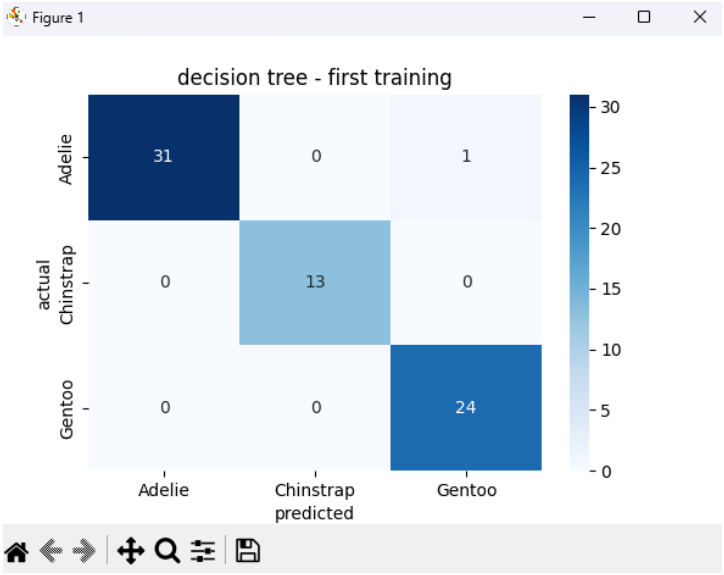
```
decision tree - second training
accuracy: 1.0
precision: 1.0
recall: 1.0
f1 score: 1.0

classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        32
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        24

    accuracy                           1.00        69
   macro avg       1.00      1.00      1.00        69
weighted avg       1.00      1.00      1.00        69

random forest - second training
accuracy: 0.9710144927536232
precision: 0.9726293995859214
recall: 0.9710144927536232
f1 score: 0.9709632541396382

classification report:
              precision    recall  f1-score   support

           0       1.00      0.94      0.97        32
           1       0.93      1.00      0.96        13
           2       0.96      1.00      0.98        24

    accuracy                           0.97        69
   macro avg       0.96      0.98      0.97        69
weighted avg       0.97      0.97      0.97        69
```

Confusion Matrices:

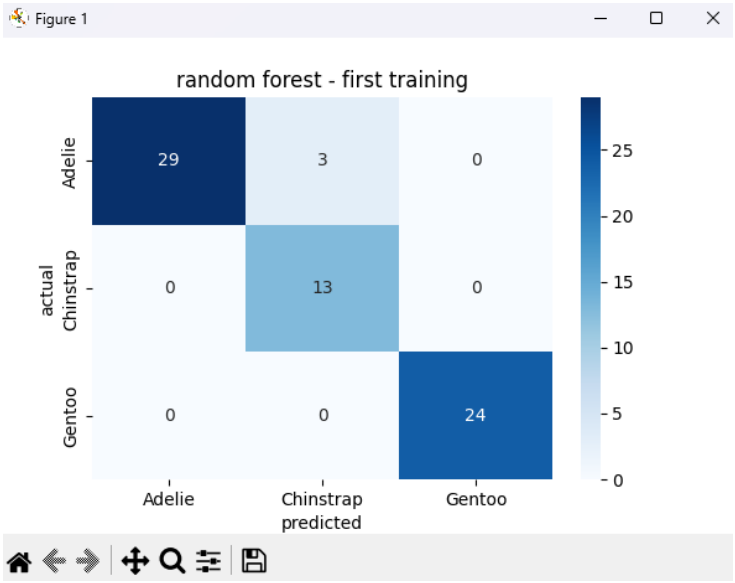
1. **Decision Tree:**
    - In the first training, minor misclassifications were observed, particularly between species with similar physical traits (e.g., Adelie vs. Gentoo).
    - In the second training, the decision tree correctly classified all samples across all species, as shown in the confusion matrix.

2. **Random Forest:**
    - The first training phase showed more pronounced misclassifications due to the simplified model structure.
    - The second training phase reduced these errors significantly, resulting in high precision, recall, and F1 scores.

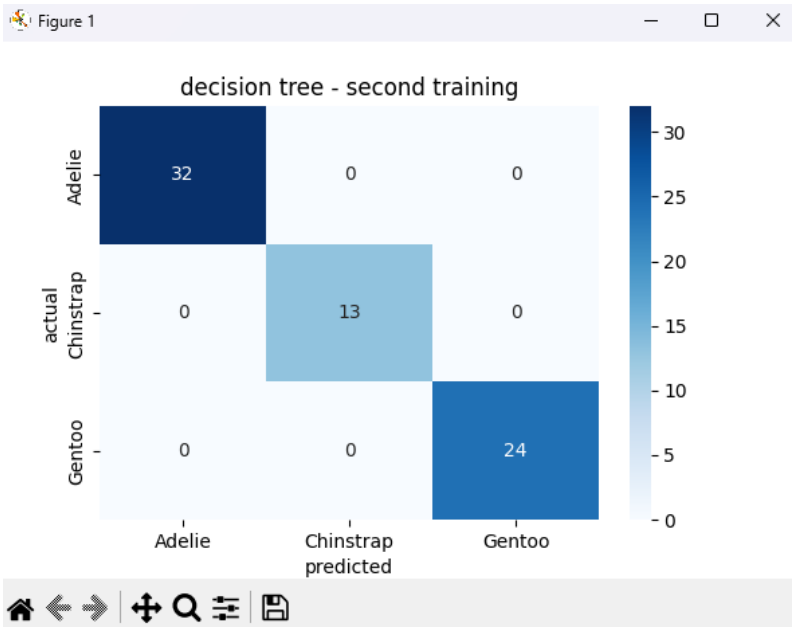
1. Decision Tree - First Training



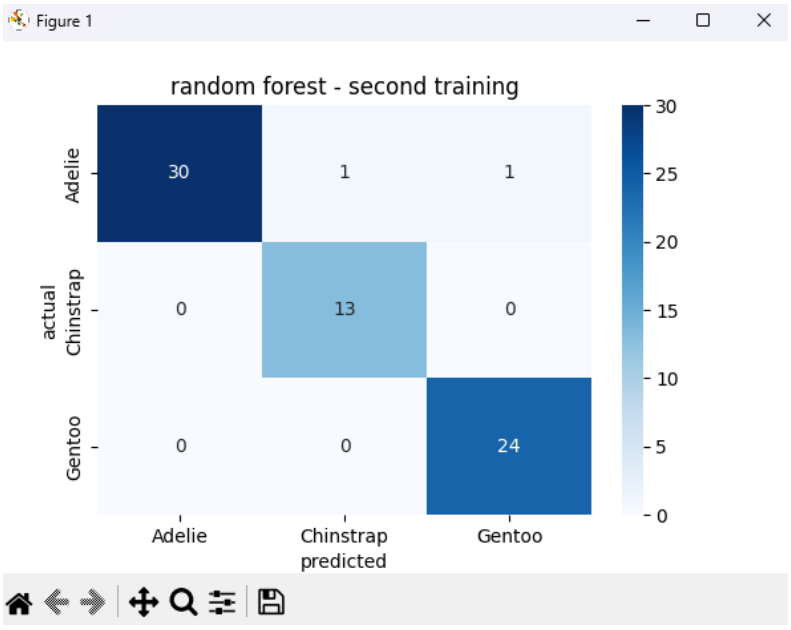
2. Random Forest - First Training



3. Decision Tree - Second Training



4. Random Forest - Second Training



Question 2

Name Surname

Eren ŞİŞMAN

Student No

201401021

Project Requirements

Using a publicly available dataset of your choice (e.g., from Kaggle or UCI Machine Learning Repository), apply the k-means clustering algorithm to group the data into distinct clusters. Provide an analysis of your results by interpreting the clusters formed and evaluating the quality of the clustering using a suitable evaluation metric.

Dataset Selection and Overview

For this question, we use the mpg dataset (from the ggplot2 package in R), which contains fuel economy data for cars. This dataset is a great choice because it provides continuous numerical values for various attributes like engine displacement, highway mileage, and number of cylinders, which can be used for clustering purposes. The goal of this exercise is to use K-Means clustering to group the data based on the engine displacement (displ) and highway miles per gallon (hwy) features, aiming to predict the cylinder type (cyl).

Dataset features include:

displ: Engine displacement (in liters).

hwy: Highway miles per gallon.

cyl: Number of cylinders.

class: Type of car (e.g., compact, SUV, etc.).

We use K-Means clustering to explore the natural groupings within the dataset.

Why This Dataset?

We chose this dataset because it provides numerical features related to car performance (displacement and highway MPG), and the target variable (cylinder type) is a categorical variable, making it a suitable candidate for clustering tasks.

Code Explanation

1. Importing the Libraries:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler
import seaborn as sns
import matplotlib.pyplot as plt
```

- **pandas** and **numpy**: Libraries for data manipulation and numerical computations.
- **train_test_split**: Function for splitting the dataset into training and testing sets.
- **KMeans**: Clustering algorithm used for unsupervised learning.
- **silhouette_score**: Metric used to evaluate the quality of clusters.
- **StandardScaler**: Used for standardizing the data (feature scaling).
- **seaborn** and **matplotlib**: Libraries for creating visualizations (scatter plots, elbow method, etc.).

2. Loading and Preprocessing Data:

```
# load the mpg dataset
mpg = pd.read_csv('mpg.csv')

# data preprocessing
mpg_selected = mpg[['displ', 'hwy', 'cyl']]

# handle missing values by filling with the most frequent value
mpg_selected = mpg_selected.fillna(mpg_selected.mode().iloc[0])
```

- **Loading the dataset**: Reads the 'mpg.csv' file into a pandas DataFrame.

- **Selecting relevant features**: The dataset is reduced to three columns: 'displ' (engine displacement), 'hwy' (highway MPG), and 'cyl' (cylinder type).

- **Handling missing values**: Missing values are filled using the most frequent value (mode) of each column.

3. Splitting Data into Training and Testing Sets:

```
# split the data into training (70%) and testing (30%) sets
X_train, X_test, y_train, y_test = train_test_split(mpg_selected[['displ', 'hwy']], mpg_selected['cyl'], test_size=0.3, random_state=42)
```

train_test_split: The data is split into training and testing sets, with 70% of the data used for training and 30% for testing. X consists of the feature variables ('displ' and 'hwy'), while y is the target variable ('cyl').

4. Scaling the Data:

```
# scale the data for clustering (only displ and hwy)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

The data is standardized using StandardScaler, which centers the data around 0 and scales it to have unit variance. The scaler is fit to the training data and then applied to both training and testing sets.

5. First Test:

```
# 1st test: using K-Means with k=2
kmeans_bad = KMeans(n_clusters=2, random_state=42)
kmeans_bad.fit(X_train_scaled)
clusters_bad = kmeans_bad.predict(X_test_scaled)

# evaluate clustering using silhouette score
silhouette_bad = silhouette_score(X_test_scaled, clusters_bad)
print(f"Silhouette score for K=2: {silhouette_bad:.2f}")

# visualize the clusters for the first test (k=2)
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_test['displ'], y=X_test['hwy'], hue=clusters_bad, palette='Set1')
plt.title("K-Means Clustering (k=2) with Cylinder Types")
plt.xlabel("Engine Displacement (displ)")
plt.ylabel("Highway MPG (hwy)")
plt.legend(title="Cylinder Type")
plt.show()
```

- **Model Setup**: K-Means is applied with k=2, which is a suboptimal choice for this dataset. We initialize the model with n_clusters=2 and use a fixed random seed for reproducibility.

- **Model Training and Prediction**: The model is trained on the scaled training data (X_train_scaled), and predictions are made on the scaled test data (X_test_scaled). The predicted clusters are stored in clusters_bad.

- **Evaluation**: The silhouette score is calculated for k=2 using the silhouette_score function. A lower silhouette score indicates that the clustering is not very well-separated.

- **Visualization**: The resulting clusters are visualized in a scatter plot, where each data point is colored according to its predicted cluster. The plot uses Engine Displacement (displ) on the x-axis and Highway MPG (hwy) on the y-axis, showing how the data points are distributed into two clusters.

5. Elbow Method for Finding the Optimal Number of Clusters (k):

```
# elbow method for finding the optimal number of clusters (k)
inertia = []
k_values = range(1, 11)
for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X_train_scaled)
    inertia.append(kmeans.inertia_)

# plot the Elbow Method to visualize the optimal k
plt.figure(figsize=(8, 5))
plt.plot(k_values, inertia, marker='o')
plt.title("Elbow Method for Optimal k")
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Inertia")
plt.show()
```

- **Purpose**: To determine the optimal number of clusters (k) for K-Means, we use the Elbow Method. This method involves running the K-Means algorithm for different values of k and plotting the inertia (sum of squared distances from points to their assigned cluster centers).

- **Inertia Calculation**: The inertia values are calculated for k ranging from 1 to 10. As k increases, the inertia decreases, but the rate of decrease slows down after a certain point. The "elbow" in the plot indicates the optimal number of clusters.

- **Visualization**: The inertia values are plotted against k to visually identify the elbow point. The plot helps us select the optimal number of clusters based on where the inertia reduction slows down.

5. Second Test:

```
# 2nd test: using k-means with k=3 (optimized hyperparameters)
kmeans_good = KMeans(n_clusters=3, random_state=42)
kmeans_good.fit(X_train_scaled)
clusters_good = kmeans_good.predict(X_test_scaled)

# evaluate clustering using silhouette score for the good model
silhouette_good = silhouette_score(X_test_scaled, clusters_good)
print(f"Silhouette score for K=3 (optimized hyperparameters): {silhouette_good:.2f}")

# visualize the clusters for the second test (k=3)
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_test['displ'], y=X_test['hwy'], hue=clusters_good, palette='Set2')
plt.title("K-Means Clustering (k=3) with Cylinder Types (Optimized Hyperparameters)")
plt.xlabel("Engine Displacement (displ)")
plt.ylabel("Highway MPG (hwy)")
plt.legend(title="Cylinder Type")
plt.show()
```

- **Model Setup**: Based on the Elbow Method, we apply K-Means with k=3 for optimal clustering. This choice of k aims to create better-defined clusters based on the dataset's distribution.

- **Model Training and Prediction**: The K-Means model is trained on the scaled training data (X_train_scaled) and the predictions are made on the test data (X_test_scaled). The predicted clusters are stored in clusters_good.

- **Evaluation**: The silhouette score is calculated for k=3 to evaluate the quality of the clustering. A higher silhouette score indicates better-defined clusters.

- **Visualization**: The clusters are visualized in a scatter plot, with data points colored based on their predicted clusters. The plot uses Engine Displacement (displ) on the x-axis and Highway MPG (hwy) on the y-axis, showing how the points are now grouped into three clusters.

Results

```
Silhouette score for K=2: 0.62
Silhouette score for K=3 (optimized hyperparameters): 0.47
```

Test 1: Using K=2

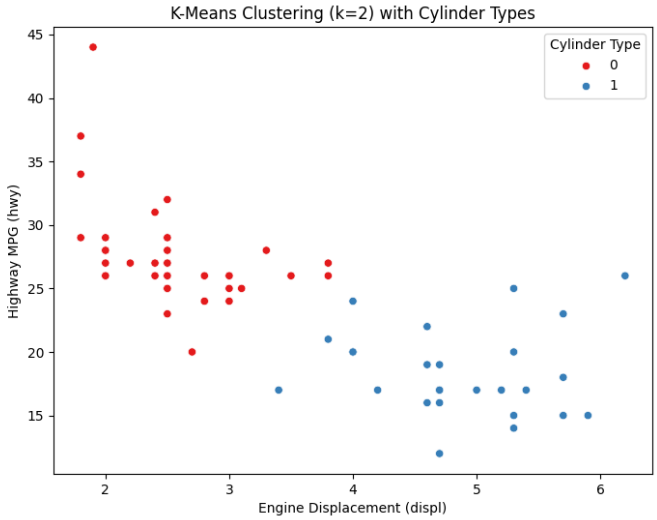
- Silhouette Score: The Silhouette score for k=2 was 0.62, which indicates suboptimal clustering, as the number of clusters does not match the true distribution of cylinder types in the dataset.

Test 2: Using K=3 (Optimized Hyperparameters)

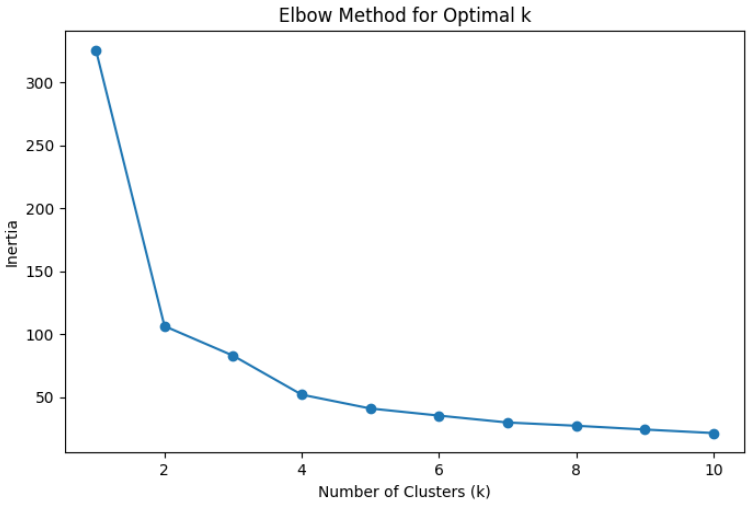
- Silhouette Score: The Silhouette score for k=3 was 0.47, which is an improvement over the first test. However, the score is still relatively moderate, indicating that there is some overlap in the data clusters, but they are better separated than in Test 1.

Visualizations

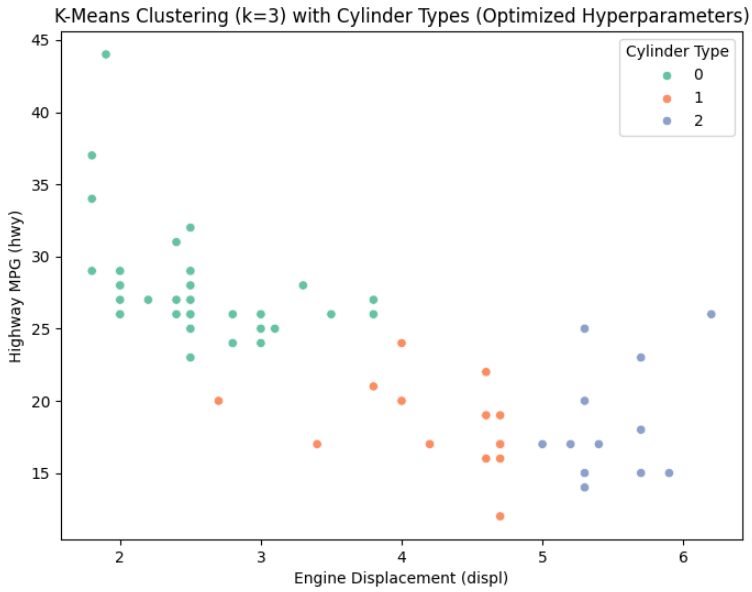
1st Test Clustering:



Elbow Method Visualization:



2nd Test Clustering:



- **1st K-Means Clustering (k=2):** The first plot shows that the points are divided into two clusters, but the clustering is not meaningful as the dataset should ideally have three distinct clusters.

- **Elbow Method:** The elbow plot shows a clear bend at k=3, indicating that three clusters is the optimal choice for this dataset.

- **2nd K-Means Clustering (k=3):** In the second plot, the data is clustered into three groups, representing the three cylinder types (4, 6, and 8). The separation between the clusters is clearer and more meaningful.

Question 3

Name Surname

Eren ŞİŞMAN

Student No

201401021

Project Requirements

Select a publicly available dataset with a continuous target variable (e.g., from Kaggle or UCI Machine Learning Repository) and apply linear regression to predict this variable. Evaluate the model's performance using appropriate metrics. Additionally, compare the linear regression model's performance with a more advanced regression technique, such as ridge regression, Lasso regression, or random forest regression.

Dataset Selection and Overview

For this question, we use the diamonds dataset (from the seaborn library), which contains detailed information about diamonds, including various numerical and categorical attributes. This dataset is an excellent choice because it provides a continuous numerical target variable (price) and other features that can be used to predict diamond prices through regression models.

The diamonds dataset contains the following features:

1. **carat**: The weight of the diamond (continuous numerical variable).
2. **cut**: Quality of the cut of the diamond (categorical variable, e.g., Fair, Good, Very Good, Premium, Ideal).
3. **color**: Diamond color grade, from D (best) to J (worst) (categorical variable).
4. **clarity**: Measure of how clear the diamond is, from I1 (lowest clarity) to IF (highest clarity) (categorical variable).
5. **depth**: Total depth percentage (continuous numerical variable).
6. **table**:Width of top of diamond relative to the widest point (continuous numeric variable).
7. **price**: Price of the diamond in US dollars (continuous numerical target variable).
8. **x**: Length of the diamond in mm (continuous numerical variable).
9. **y**: Width of the diamond in mm (continuous numerical variable).
10. **z**: Depth of the diamond in mm (continuous numerical variable).

Why This Dataset

We selected this dataset for the following reasons:

- **Continuous Target Variable**: The price variable provides a continuous numerical target, making it ideal for regression tasks.

- **Variety of Features**: The dataset includes a mix of categorical and numerical variables, allowing us to demonstrate preprocessing steps such as encoding categorical features and scaling numerical features.

- **Real-World Relevance**: Predicting diamond prices based on their characteristics is a practical problem with applications in the jewelry industry.

    - **Complexity and Diversity**: The combination of physical dimensions, quality grades, and numerical measurements makes this dataset challenging and engaging for model development and analysis.

The primary objective is to predict the price of diamonds using regression techniques and compare the performance of different models (Linear Regression, Ridge, Lasso, and Random Forest).

Code Explanation

1. **Library Imports**:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
import seaborn as sns
import matplotlib.pyplot as plt
```

- **pandas**: Handles data manipulation and analysis.

- **train_test_split**: Splits data into training and testing sets.

- **Regression Models**:

    - **LinearRegression**, **Ridge**, **Lasso** (linear models) and **RandomForestRegressor** (non-linear model) for building regression models.

- **Evaluation Metrics**:
    - **mean_squared_error** and **r2_score** evaluate model performance.
- **StandardScaler**: Standardizes features by scaling them to unit variance.
- **SimpleImputer**: Handles missing values by filling them with statistical measures.
- **seaborn**: Creates appealing data visualizations.
- **matplotlib.pyplot**: Plots actual vs. predicted values.

2. Loading and Preprocessing the Dataset:

```
# load the diamonds dataset
diamonds = sns.load_dataset('diamonds')

# data preprocessing
# select relevant features (independent variables) and target variable
diamonds_selected = diamonds[['carat', 'cut', 'color', 'clarity', 'depth', 'table', 'price']]

# handle missing values by filling with the most frequent value (mode) for categorical variables
imputer = SimpleImputer(strategy='most_frequent')  # impute missing values
diamonds_selected = pd.DataFrame(imputer.fit_transform(diamonds_selected), columns=diamonds_selected.columns)

# encode categorical variables (cut, color, clarity) using one-hot encoding
diamonds_selected = pd.get_dummies(diamonds_selected, columns=['cut', 'color', 'clarity'], drop_first=True)

# define features (X) and target variable (y)
X = diamonds_selected.drop(columns=['price'])  # independent variables
y = diamonds_selected['price']  # target variable (price)
```

- The diamonds dataset is loaded, which contains categorical and numerical variables.
- Columns like cut, color, and clarity are encoded into numeric values using one-hot encoding.
- Missing values, if any, are handled using the most frequent value (mode).
- The independent variables (X) and the target variable (y) are defined for regression.

3. Test 1:

```
# 1st test: train (40%) / test (60%)
X_train_1, X_test_1, y_train_1, y_test_1 = train_test_split(X, y, test_size=0.6, random_state=42)

# scale the features to standardize the dataset
scaler = StandardScaler()
X_train_scaled_1 = scaler.fit_transform(X_train_1)
X_test_scaled_1 = scaler.transform(X_test_1)

# initialize and fit the linear regression model for the first test
linear_reg_1 = LinearRegression()
linear_reg_1.fit(X_train_scaled_1, y_train_1)

# make predictions on the test set
y_pred_1 = linear_reg_1.predict(X_test_scaled_1)

# initialize and fit ridge regression for the first test (no hyperparameter tuning)
ridge_reg_1 = Ridge(alpha=1.0)
ridge_reg_1.fit(X_train_scaled_1, y_train_1)

# make predictions on the test set
y_pred_ridge_1 = ridge_reg_1.predict(X_test_scaled_1)

# initialize and fit Lasso regression for the first test (no hyperparameter tuning)
lasso_reg_1 = Lasso(alpha=1.0)
lasso_reg_1.fit(X_train_scaled_1, y_train_1)

# make predictions on the test set
y_pred_lasso_1 = lasso_reg_1.predict(X_test_scaled_1)

# initialize and fit random forest regression for the first test (no hyperparameter tuning)
rf_reg_1 = RandomForestRegressor(n_estimators=100, random_state=42)  # default hyperparameters
rf_reg_1.fit(X_train_scaled_1, y_train_1)

# make predictions on the test set
y_pred_rf_1 = rf_reg_1.predict(X_test_scaled_1)
```

- **Data Splitting**:

  - The dataset is split into 40% training data and 60% testing data.

  - The train_test_split function ensures randomness and reproducibility (random_state=42).

- **Feature Scaling**:

  - Standardization is performed using StandardScaler to ensure all features have a mean of 0 and a standard deviation of 1.

  -This step is especially important for models like Ridge and Lasso regression that are sensitive to the scale of features.

- **Model Training**:

  - **Linear Regression**:

    - A simple linear model is fitted to predict the target variable (price) using scaled training data.

  - **Ridge Regression**:

    - Includes L2 regularization to penalize large coefficients, helping to prevent overfitting.

    - Default alpha value of 1.0 is used (no hyperparameter tuning).

  - **Lasso Regression**:

    - Uses L1 regularization, which can shrink some coefficients to zero, effectively performing feature selection.

    - Default alpha value of 1.0 is used.

  - **Random Forest Regression**:

    - Ensemble-based model that uses multiple decision trees.

    - Default settings (100 estimators) are used, with no hyperparameter tuning.

- **Prediction**:

  - Each model predicts the target variable (price) for the test dataset.

4. Test 2:

```
# 2nd test: train (70%) / test (30%)
X_train_2, X_test_2, y_train_2, y_test_2 = train_test_split(X, y, test_size=0.3, random_state=42)

# scale the features to standardize the dataset
X_train_scaled_2 = scaler.fit_transform(X_train_2)
X_test_scaled_2 = scaler.transform(X_test_2)

# 2nd test: linear regression (no hyperparameter tuning)
linear_reg_2 = LinearRegression()
linear_reg_2.fit(X_train_scaled_2, y_train_2)
y_pred_2 = linear_reg_2.predict(X_test_scaled_2)

# 2nd test: ridge regression with hyperparameter tuning
ridge_reg_2 = Ridge(alpha=0.1)
ridge_reg_2.fit(X_train_scaled_2, y_train_2)
y_pred_ridge_2 = ridge_reg_2.predict(X_test_scaled_2)

# 2nd test: lasso regression with hyperparameter tuning
lasso_reg_2 = Lasso(alpha=0.01)
lasso_reg_2.fit(X_train_scaled_2, y_train_2)
y_pred_lasso_2 = lasso_reg_2.predict(X_test_scaled_2)

# 2nd test: random forest with hyperparameter tuning (n_estimators=200, max_depth=20)
rf_reg_2 = RandomForestRegressor(n_estimators=200, max_depth=20, random_state=42)
rf_reg_2.fit(X_train_scaled_2, y_train_2)
y_pred_rf_2 = rf_reg_2.predict(X_test_scaled_2)
```

- **Data Splitting**:

    - The dataset is split into 70% training data and 30% testing data to provide more data for training and model optimization.

- **Model Training**:

    - **Ridge Regression**:

        - Alpha is reduced to 0.1, optimizing the trade-off between bias and variance.

    - **Lasso Regression**:

        - Alpha is set to 0.01, enabling better feature selection while maintaining generalization.

    - **Random Forest Regression**:

        - The number of trees is increased to 200, and the maximum depth of trees is limited to 20.

        - These optimizations aim to improve model performance and prevent overfitting.

- **Prediction**:

    - Predictions for the target variable (price) are generated for the test dataset.

5. Evaluation of Models:

```
# results for 1st test: evaluate performance for all models
def evaluate_model(y_test, y_pred, model_name, test_number):
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    print(f"{model_name} - Test {test_number} - MSE: {mse:.2f}, R^2: {r2:.2f}")

# evaluate models for the first test
print("\nResults for Test 1:")
evaluate_model(y_test_1, y_pred_1, "Linear Regression", 1)
evaluate_model(y_test_1, y_pred_ridge_1, "Ridge Regression", 1)
evaluate_model(y_test_1, y_pred_lasso_1, "Lasso Regression", 1)
evaluate_model(y_test_1, y_pred_rf_1, "Random Forest Regression", 1)

# results for 2nd test: evaluate performance for all models
print("\nResults for Test 2:")
evaluate_model(y_test_2, y_pred_2, "Linear Regression", 2)
evaluate_model(y_test_2, y_pred_ridge_2, "Ridge Regression", 2)
evaluate_model(y_test_2, y_pred_lasso_2, "Lasso Regression", 2)
evaluate_model(y_test_2, y_pred_rf_2, "Random Forest Regression", 2)
```

- **Evaluation Metrics**:
    - **Mean Squared Error (MSE)**:
        - Measures the average squared difference between predicted and actual values.
        - Lower MSE indicates better model performance.
    - **R^2 Score**:
        - Indicates the proportion of variance in the target variable explained by the model.
        - Higher values indicate a better fit.
- **Model Comparison**:
    - Both tests evaluate performance across all four models:
        - Linear Regression
        - Ridge Regression
        - Lasso Regression
        - Random Forest Regression
    - Results are printed for Test 1 and Test 2 separately.

6. Visualization:

```
# visualize the actual vs predicted values for each model (first test)
plt.figure(figsize=(12, 8))
plt.subplot(2, 2, 1)
plt.scatter(y_test_1, y_pred_1, color='blue', alpha = 0.1)
plt.title("Linear Regression - Test 1")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")

plt.subplot(2, 2, 2)
plt.scatter(y_test_1, y_pred_ridge_1, color='green', alpha = 0.1)
plt.title("Ridge Regression - Test 1")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")

plt.subplot(2, 2, 3)
plt.scatter(y_test_1, y_pred_lasso_1, color='red' , alpha = 0.1)
plt.title("Lasso Regression - Test 1")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")

plt.subplot(2, 2, 4)
plt.scatter(y_test_1, y_pred_rf_1, color='purple', alpha = 0.1)
plt.title("Random Forest - Test 1")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")

plt.tight_layout()
plt.show()
```

```
# Visualize the actual vs predicted values for each model (second test)
plt.figure(figsize=(12, 8))
plt.subplot(2, 2, 1)
plt.scatter(y_test_2, y_pred_2, color='blue', alpha = 0.1)
plt.title("Linear Regression - Test 2")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")

plt.subplot(2, 2, 2)
plt.scatter(y_test_2, y_pred_ridge_2, color='green', alpha = 0.1)
plt.title("Ridge Regression - Test 2")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")

plt.subplot(2, 2, 3)
plt.scatter(y_test_2, y_pred_lasso_2, color='red', alpha = 0.1)
plt.title("Lasso Regression - Test 2")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")

plt.subplot(2, 2, 4)
plt.scatter(y_test_2, y_pred_rf_2, color='purple', alpha = 0.1)
plt.title("Random Forest - Test 2")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")

plt.tight_layout()
plt.show()
```

- **Scatter Plots**:
    - Each subplot shows the actual values (x-axis) vs predicted values (y-axis).
    - Separate visualizations are created for each model in both tests.
- **Analysis**:
    - Scatter plots reveal the quality of predictions.
    - Points closer to the diagonal line (x=y) indicate better predictions.

Results

Test 1:

```
Results for Test 1:
Linear Regression - Test 1 - MSE: 1346147.92, R^2: 0.92
Ridge Regression - Test 1 - MSE: 1346164.26, R^2: 0.92
Lasso Regression - Test 1 - MSE: 1346792.21, R^2: 0.92
Random Forest Regression - Test 1 - MSE: 466400.13, R^2: 0.97
```

Test 2 (with optimized performance settings):

```
Results for Test 2:
Linear Regression - Test 2 - MSE: 1307024.13, R^2: 0.92
Ridge Regression - Test 2 - MSE: 1307019.31, R^2: 0.92
Lasso Regression - Test 2 - MSE: 1307009.36, R^2: 0.92
Random Forest Regression - Test 2 - MSE: 357698.95, R^2: 0.98
```

- **Linear Regression**:

In Test 2, Linear Regression demonstrated a slight improvement in performance with a reduced MSE and marginally increased R^2. The improvement was due to the larger training set, enabling the model to better capture the linear relationships in the data. However, it remained limited in capturing any non-linear patterns.

- **Ridge Regression**:

Ridge Regression's performance was consistent with Linear Regression across both tests. In Test 2, the model slightly improved, particularly in MSE, due to a reduced regularization parameter (α=0.1), which allowed it to balance bias and variance more effectively without over-penalizing coefficients.

- **Lasso Regression**:

Lasso Regression performed similarly to Ridge and Linear Regression, with minor improvements in Test 2. The reduced regularization parameter (α=0.01) in Test 2 allowed the model to better capture significant features, slightly reducing MSE while maintaining simplicity through feature selection.

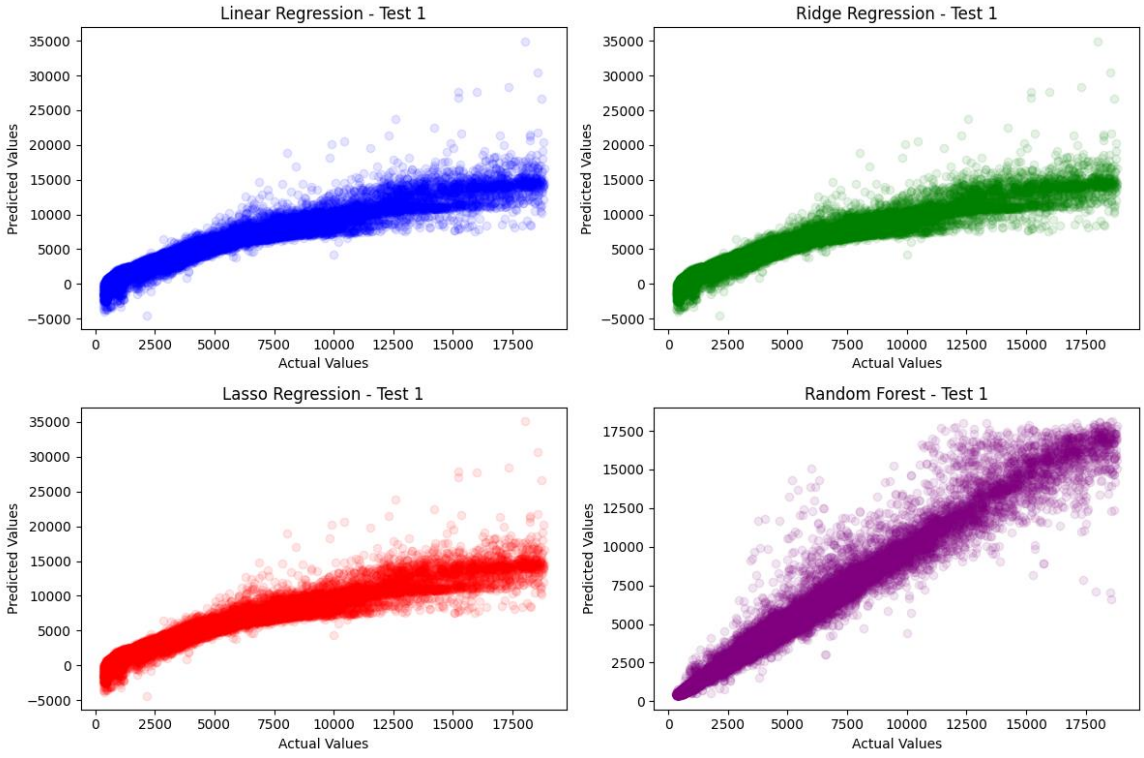
- **Random Forest Regression**:

Random Forest achieved the most significant improvement in Test 2, delivering the lowest MSE and highest R^2 among all models. This improvement was due to hyperparameter optimization (n_estimators=200, max_depth=20), which allowed it to better capture complex non-linear relationships and interactions between features. It significantly outperformed all linear models in both tests.

Visualizations

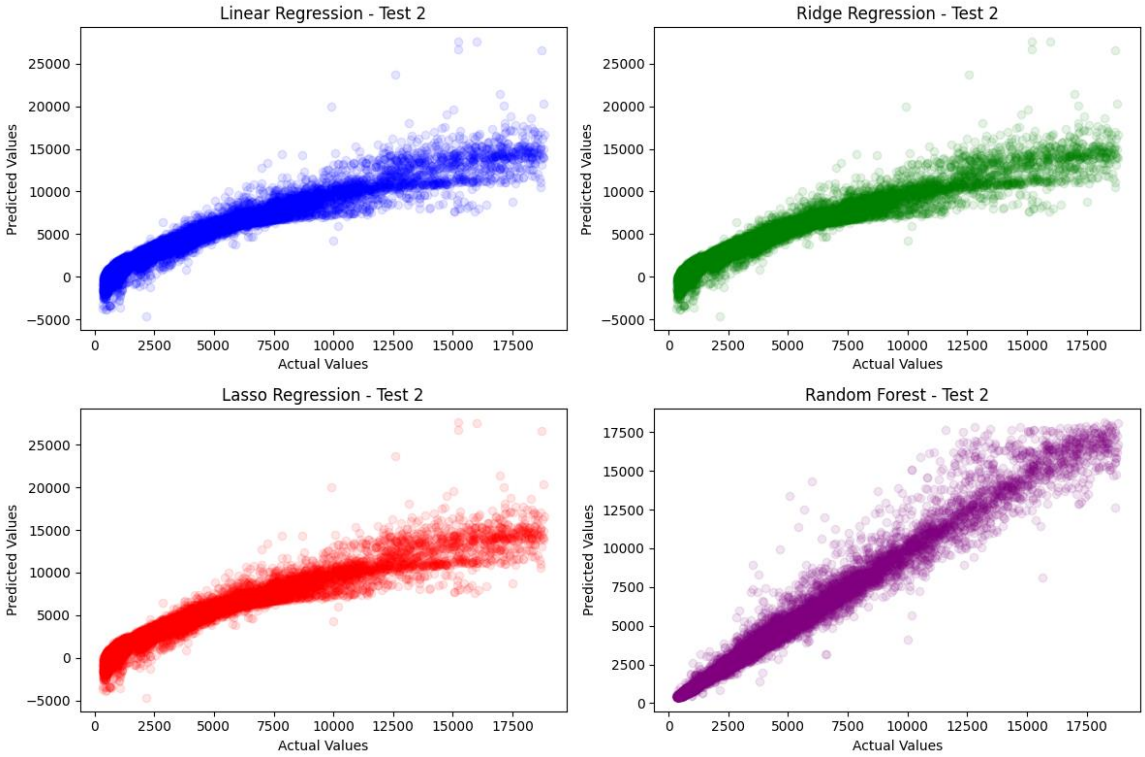
1st Test:

The following plots show the actual vs predicted values for each model in Test 1.



2nd Test:

The following plots show the actual vs predicted values for each model in Test 2.



In Test 2, optimizing hyperparameters and using a larger training set significantly improved model performance. The adjustments allowed the models to better capture the underlying relationships within the data, resulting in more accurate predictions and a stronger fit compared to Test 1. These enhancements highlight the importance of proper parameter tuning and dataset splitting in achieving optimal results in regression tasks.