RECEP TAYYIP ERDOGAN UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

COMPUTER ENGINEERING DEPARTMENT

CEN411 DATA MINING

HOMEWORK

2024-2025

Due Date: 08.12.2024

Student Name Surname

Nasıf Can YAVUZ

Student No

201401018

Instructor

Abdulgani KAHRAMAN

RIZE

RTEU COMPUTER ENGINEERING DATA MINING HOMEWORK: QUESTION 1

---

1. DATASET

I used heart disase dataset (heart.csv) from Kaggle.

Source: Heart Disease Dataset

1.2 Reading The Dataset

I assigned the dataset to the variable 'df' in python environment as relative path.

Code:

```
# Load the dataset
data_path = '../datasets/heart.csv'
df = pd.read_csv(data_path)
```

1.3 Dataset Details

Info() function used to get basic information about the dataset.

Code:

```
# Basic information
print("\nDataset Information:")
print(df.info())
```

Console output:

```
Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1025 entries, 0 to 1024
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       1025 non-null   int64
 1   sex       1025 non-null   int64
 2   cp        1025 non-null   int64
 3   trestbps  1025 non-null   int64
 4   chol      1025 non-null   int64
 5   fbs       1025 non-null   int64
 6   restecg   1025 non-null   int64
 7   thalach   1025 non-null   int64
 8   exang     1025 non-null   int64
 9   oldpeak   1025 non-null   float64
 10  slope     1025 non-null   int64
 11  ca        1025 non-null   int64
 12  thal      1025 non-null   int64
 13  target    1025 non-null   int64
dtypes: float64(1), int64(13)
```

- **Records Count**: 1025
- **Attribute Count**: 14 (13 independent variables, 1 target variable)

- **Target Variable**: target (0 = no heart disease, 1 = heart disease)

Attribute	Description
Age	Age of the patient (in years).
Sex	Gender of the patient (1 = Male, 0 = Female).
Cp	Chest pain type (0 = Typical angina, 1 = Atypical angina, 2 = Non-anginal, 3 = Asymptomatic).
Trestbps	Resting blood pressure (in mm Hg).
Chol	Serum cholesterol level (in mg/dl).
Fbs	Fasting blood sugar (> 120 mg/dl, 1 = True, 0 = False).
Restecg	Resting electrocardiographic results (0 = Normal, 1 = ST-T wave abnormality, 2 = Left ventricular hypertrophy).
thalach	Maximum heart rate achieved.
exang	Exercise-induced angina (1 = Yes, 0 = No).
oldpeak	ST depression induced by exercise relative to rest.
slope	The slope of the peak exercise ST segment (0 = Downsloping, 1 = Flat, 2 = Upsloping).
ca	Number of major vessels (0-4) colored by fluoroscopy.
thal	Thalassemia status (1 = Fixed defect, 2 = Normal, 3 = Reversible defect).
target	Target variable (1 = Presence of heart disease, 0 = Absence of heart disease).

---

2. DATA PREPROCESSING

2.1 Missing Data Check

Empty cells in the dataframe were checked to check for the presence of missing data.

Code:

```
1. # Check for missing values
2. print("\nMissing Values:")
3. print(df.isnull().sum())
```

Console Output:

```
1. Missing Values:
2. age         0
3. sex         0
4. cp          0
5. trestbps    0
```

```
 6. chol        0
 7. fbs         0
 8. restecg     0
 9. thalach     0
10. exang       0
11. oldpeak     0
12. slope       0
13. ca          0
14. thal        0
15. target      0
```

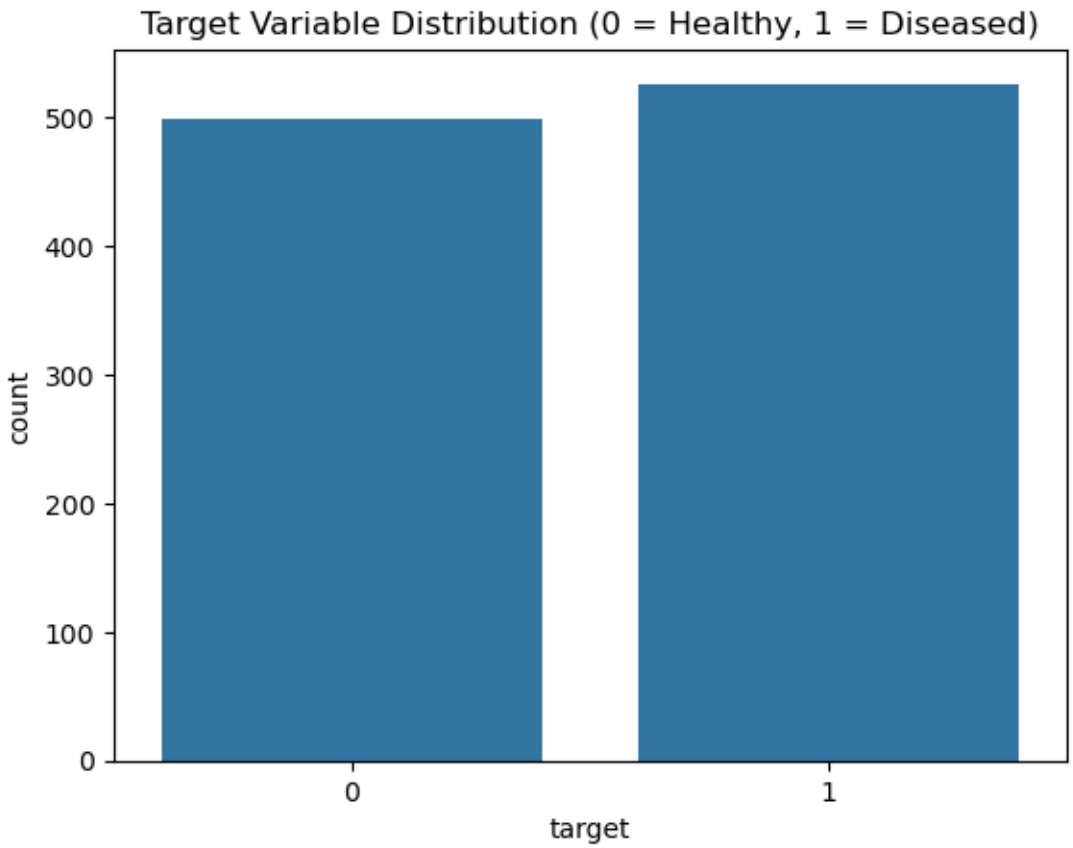
It was determined that there was no missing data in the data set. Therefore, no filling process was applied. If there were missing data, in order to train the model well, I would have to either completely remove the rows with missing data or fill the missing data with the arithmetic mean of the other data.

2.2 Target variable distribution

In this section, the distribution of the target variable is analysed.

Code:

```
1. # Target variable distribution
2. print("\nTarget Variable Distribution:")
3. print(df['target'].value_counts())
4.
5. # Visualize target distribution
6. sns.countplot(x='target', data=df)
7. plt.title("Target Variable Distribution (0 = Healthy, 1 = Diseased)")
8. plt.show()
```



Class 1 (Heart Disease): 526 samples

Class 0 (No Heart Disease): 499 samples

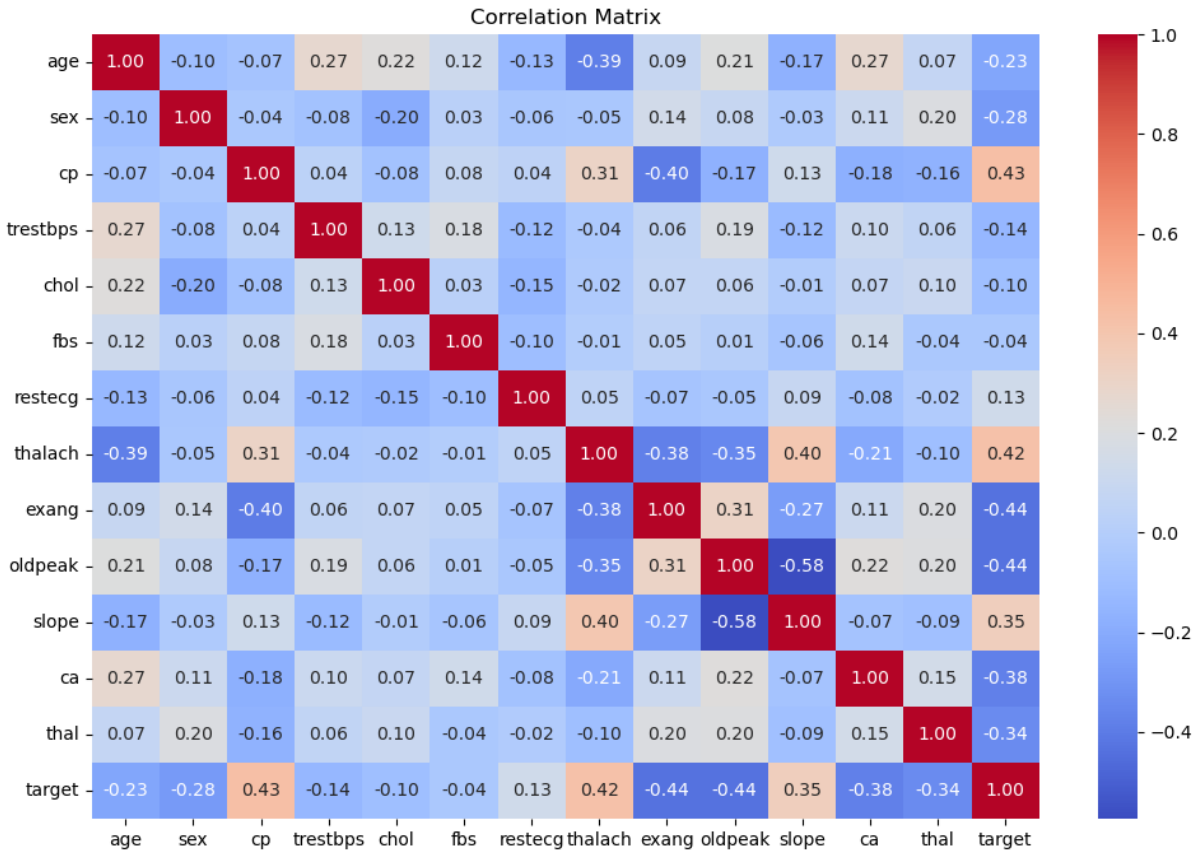
The distribution of the target variable classes was found to be balanced, therefore no resampling was required.

2.3 Feature Selection

A correlation matrix was created to examine the relationships between the features. Features with strong correlations were taken into consideration.

Code:

```
1. # Correlation matrix
2. correlation_matrix = df.corr()
3.
4. # Visualize correlation matrix
5. plt.figure(figsize=(12, 8))
6. sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap="coolwarm")
7. plt.title("Correlation Matrix")
8. plt.show()
```



Analyzed the correlation between features and the target variable.

Code:

```
# Examine correlation with target
target_corr = correlation_matrix["target"].sort_values(ascending=False)
print("\nCorrelation with Target:")
print(target_corr)

# Features to remove based on correlation analysis (example: low correlation features)
remove_features = ["trestbps", "chol", "fbs", "restecg"]

# Separate features and target variable
X = df.drop(columns=["target"] + remove_features)
y = df["target"]
```

Features with low correlation (trestbps, chol, fbs, and restecg) were removed to improve model performance.

3 MODEL EVALUATION PROCESS

3.1 Splitting the data to test and train data

The dataset was split into training (80%) and test (20%) sets using the train_test_split function, ensuring the class distribution was preserved (stratified) and the sizes of the training and test sets were printed

Code:

```
# Split the dataset into training and test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42,
stratify=y)
print(f"Training set size: {X_train.shape}")
print(f"Test set size: {X_test.shape}")
```

3.2 Normalize The Features

The features were scaled using StandardScaler function, which standardizes the data by removing the mean and scaling to unit variance. The fit_transform method was used on the training set, while the transform method was applied to the test set.

Code:

```
# Normalize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

```

3.3 Decision Tree Model

A Decision Tree model was created using the DecisionTreeClassifier and trained on the training data (X_train, y_train).

Code:

```
# Decision Tree Model
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)
```

3.4 Decision Tree Evaluate performance

Predictions were made on the test set, and the performance of the Decision Tree model was evaluated using the classification_report, which includes accuracy, precision, recall, and F1-score.

Code:

```
# Evaluate performance
y_pred_dt = dt_model.predict(X_test)
print("Decision Tree Model Performance:")
print(classification_report(y_test, y_pred_dt))
```

Console Output:

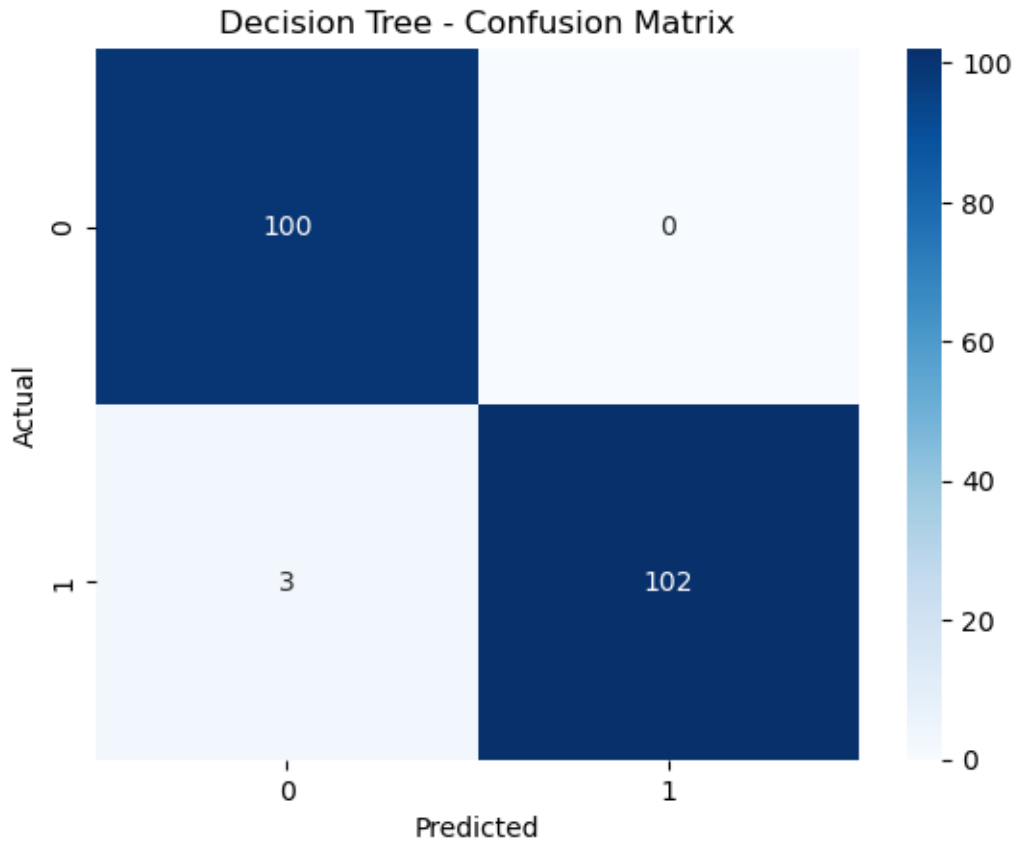
```
Decision Tree Model Performance:
              precision    recall  f1-score   support

           0       0.97      1.00      0.99       100
           1       1.00      0.97      0.99       105

    accuracy                           0.99       205
   macro avg       0.99      0.99      0.99       205
weighted avg       0.99      0.99      0.99       205
```

3.5 Decision Tree Model Confusion Matrix

The confusion matrix for the Decision Tree model was visualized using a heatmap.

Code:

```
1. # Confusion Matrix
2. sns.heatmap(confusion_matrix(y_test, y_pred_dt), annot=True, fmt="d", cmap="Blues")
3. plt.title("Decision Tree - Confusion Matrix")
4. plt.xlabel("Predicted")
5. plt.ylabel("Actual")
6. plt.show()
```



3.6 SVM Model

An SVM model was created using the SVC class and trained on the scaled training data (X_train_scaled, y_train).

Code:

```
1. # SVM Model
2. svm_model = SVC(random_state=42)
3. svm_model.fit(X_train_scaled, y_train)
```

3.7 SVM Model Evaluate performance

Predictions were made on the scaled test data, and the performance of the SVM model was evaluated using the classification_report.

Code:

```
# Evaluate performance
y_pred_svm = svm_model.predict(X_test_scaled)
print("SVM Model Performance:")
print(classification_report(y_test, y_pred_svm))
```

Console Output:

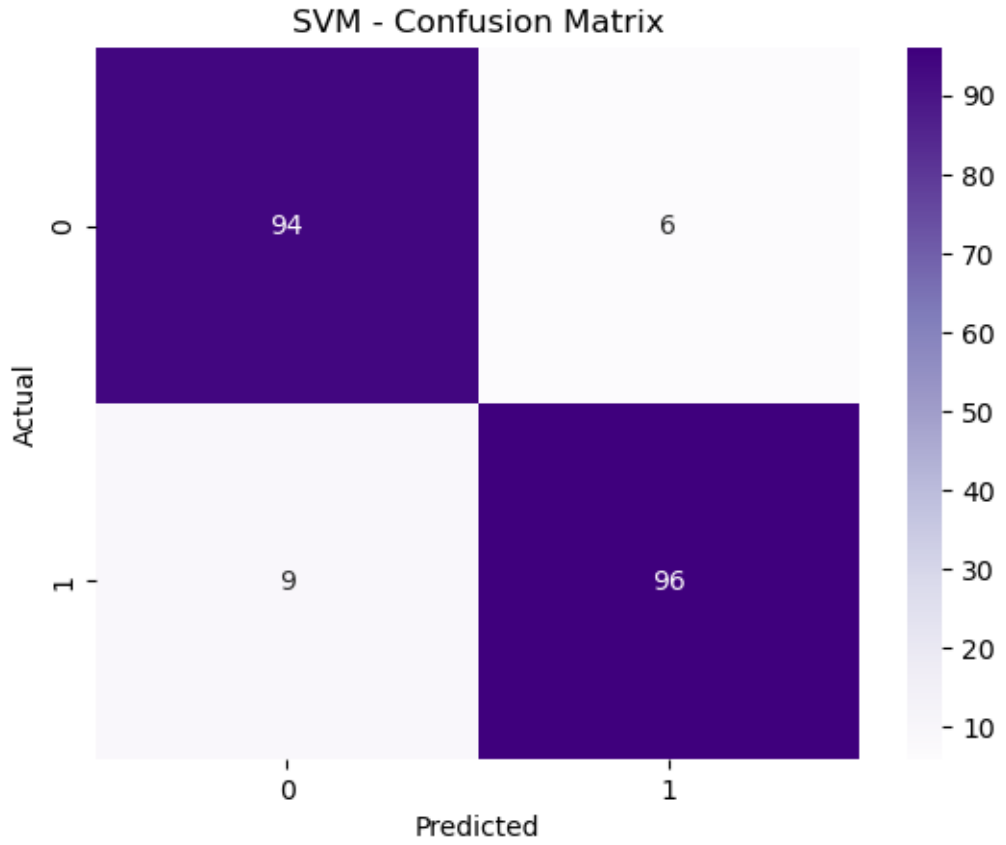
```
SVM Model Performance:
              precision    recall  f1-score   support

           0       0.91      0.94      0.93       100
           1       0.94      0.91      0.93       105

    accuracy                           0.93       205
   macro avg       0.93      0.93      0.93       205
weighted avg       0.93      0.93      0.93       205

```

3.8 SVM Model Confusion Matrix

The confusion matrix for the SVM model was visualized using a heatmap.

Code:

```
# Confusion Matrix
sns.heatmap(confusion_matrix(y_test, y_pred_svm), annot=True, fmt="d", cmap="Purples")
plt.title("SVM - Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```



4 MODEL COMPARISON

4.1 Performance Metrics Calculation

A function was created to calculate performance metrics (accuracy, precision, recall, F1 score) for any model.

Code:

```
1. # Function to calculate performance metrics
2. def calculate_metrics(y_test, y_pred):
3.     accuracy = accuracy_score(y_test, y_pred)
4.     precision = precision_score(y_test, y_pred)
5.     recall = recall_score(y_test, y_pred)
6.     f1 = f1_score(y_test, y_pred)
7.     return accuracy, precision, recall, f1
8.
```

4.2 Decision Tree Metrics

The performance metrics for the Decision Tree model were calculated and printed. (Accuracy, precision, recall, f1 score)

Code:

```
1. # Decision Tree Metrics
2. dt_metrics = calculate_metrics(y_test, y_pred_dt)
3. print("Decision Tree Metrics:", dt_metrics)
```

Console Output:

```
1. Decision Tree Metrics: (0.9853658536585366, 1.0, 0.9714285714285714, 0.9855072463768116)
```

4.3 SVM Metrics

The performance metrics for the SVM model were calculated and printed. (Accuracy, precision, recall, f1 score)

Code:

```
1. # SVM Metrics
2. svm_metrics = calculate_metrics(y_test, y_pred_svm)
3. print("SVM Metrics:", svm_metrics)
```

Console Output:

```
1. SVM Metrics: (0.926829268292683, 0.9411764705882353, 0.9142857142857143, 0.927536231884058)
```

4.4 Model Performance Comparison

A DataFrame was created to compare the performance metrics (accuracy, precision, recall, F1 score) of the Decision Tree and SVM models and the model performance comparison table was printed.

Code:

```
 1. # Comparison Table
 2. comparison_df = pd.DataFrame({
 3.     "Model": ["Decision Tree", "SVM"],
 4.     "Accuracy": [dt_metrics[0], svm_metrics[0]],
 5.     "Precision": [dt_metrics[1], svm_metrics[1]],
 6.     "Recall": [dt_metrics[2], svm_metrics[2]],
 7.     "F1 Score": [dt_metrics[3], svm_metrics[3]]
 8. })
 9.
10. print("Model Performance Comparison:")
11. print(comparison_df)
```

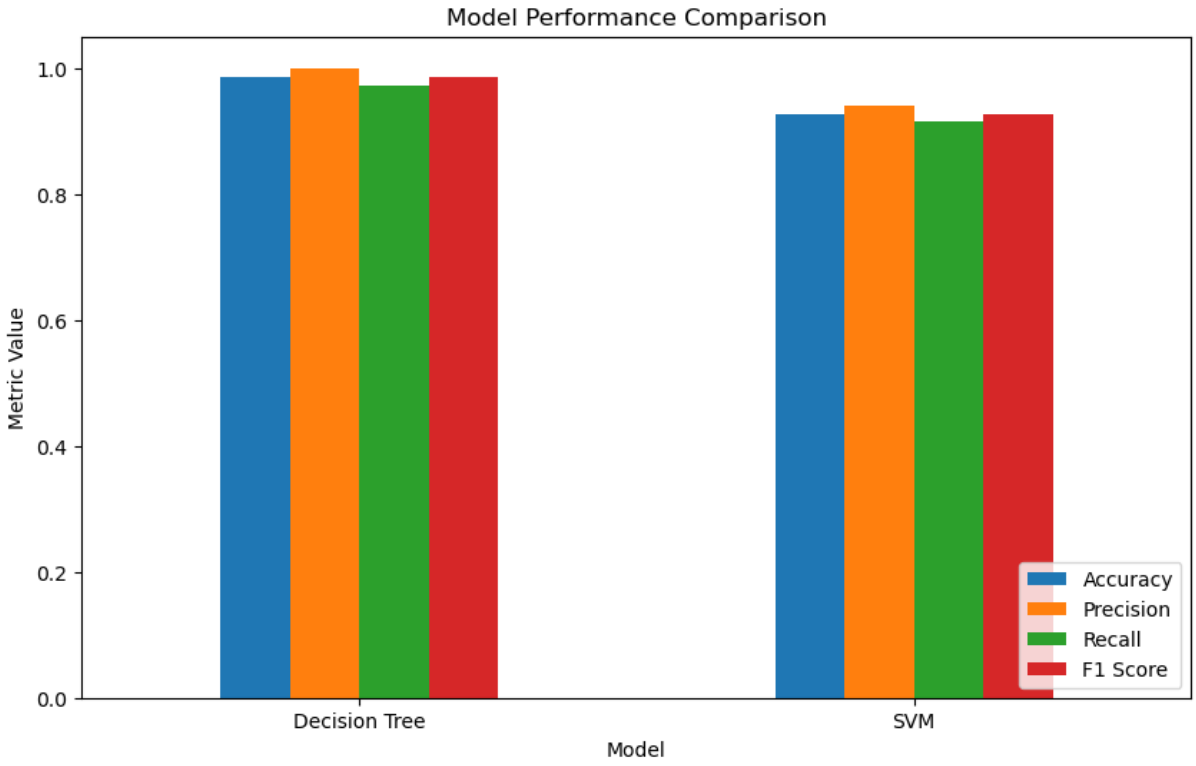
Console Output:

```
1. Model Performance Comparison:
2.            Model  Accuracy  Precision    Recall  F1 Score
3. 0  Decision Tree  0.985366   1.000000  0.971429  0.985507
4. 1            SVM  0.926829   0.941176  0.914286  0.927536
```

---

5 SUMMARY AND COMPARISON RESULTS

The aim of this study is to train decision tree and support vector machine models on heart.csv data and to compare the performances of these models. Firstly, the dataset was made suitable for the comparison of the two models and then the accuracy, precision, recall and f1 score values were compared between the two models.



5.1 Accuracy

Accuracy is the proportion of correct predictions out of the total predictions.

Formula: (True Positives+True Negatives)/Total Instances

High accuracy indicates good overall performance but may be misleading for imbalanced datasets. Thanks to our dataset is not imbalanced.

The Decision Tree achieves an accuracy of 0.985, outperforming the SVM (0.927). This suggests the Decision Tree makes fewer overall errors.

5.2 Precision

Precision is the proportion of positive predictions that are actually correct.

Formula: True Positives/(True Positives+False Positives)

High precision indicates the model effectively minimizes false positives.

The Decision Tree's precision is 1.000, meaning all positive predictions are correct. In comparison, the SVM achieves a precision of 0.941, indicating it made some false positive predictions.

5.3 Recall

The proportion of actual positives correctly identified by the model.

Formula: True Positives/(True Positives+False Negatives)

High recall indicates the model captures most of the positive examples.

The Decision Tree has a recall of 0.971, higher than the SVM's 0.914. This indicates the Decision Tree captures more true positives effectively.

5.4 F1 Score

The harmonic mean of Precision and Recall, balancing both metrics.

Formula: 2x(Precision·Recall)/(Precision+Recall)

Particularly useful for imbalanced datasets.

The Decision Tree's F1 Score (0.986) is superior to the SVM's (0.928). This highlights the Decision Tree's balanced performance in both precision and recall.

5.5 Which Model Performs Better?

The Decision Tree performs better across all metrics. Its perfect precision and high recall indicate a strong ability to correctly classify positive instances while avoiding false positives. The SVM, while delivering acceptable results, is outperformed by the Decision Tree in this analysis.

---

1. DATASET

I used wine dataset from UCI Machine Learning Repository.

Source: Wine - UCI Machine Learning Repository

1.2 Reading The Dataset

The wine dataset is loaded directly from the UCI repository using a URL. The dataset doesn't come with predefined column names, so they are assigned manually for better readability.

Code:

```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data"
columns = [
    "Class label", "Alcohol", "Malic acid", "Ash", "Alcalinity of ash", "Magnesium",
    "Total phenols", "Flavanoids", "Nonflavanoid phenols", "Proanthocyanins",
    "Color intensity", "Hue", "OD280/OD315 of diluted wines", "Proline"
]
wine_data = pd.read_csv(url, header=None, names=columns)
```

1.4 Dataset Details

Info() function used to get basic information about the dataset.

Code:

```
# Basic information
print("\nDataset Information:")
print(wine_data.info())
```

Console output:

```
Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 178 entries, 0 to 177
Data columns (total 14 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   Class label                   178 non-null    int64
 1   Alcohol                       178 non-null    float64
 2   Malic acid                    178 non-null    float64
 3   Ash                           178 non-null    float64
 4   Alcalinity of ash             178 non-null    float64
 5   Magnesium                     178 non-null    int64
 6   Total phenols                 178 non-null    float64
 7   Flavanoids                    178 non-null    float64
 8   Nonflavanoid phenols          178 non-null    float64
 9   Proanthocyanins               178 non-null    float64
 10  Color intensity               178 non-null    float64
 11  Hue                           178 non-null    float64
 12  OD280/OD315 of diluted wines  178 non-null    float64
 13  Proline                       178 non-null    int64
```

Attribute	Description
Class label	The alcohol content in the wine (measured as a percentage).
Alcohol	The amount of malic acid in the wine, a compound contributing to the wine's tartness.
Malic acid	The ash content in the wine, representing the inorganic material after combustion.
Ash	Resting blood pressure (in mm Hg).
Alcalinity of ash	A measure of the alkalinity (basicity) of the ash in the wine.
Magnesium	The magnesium content in the wine (measured in mg/l).
Total phenols	The total amount of phenolic compounds in the wine, which contribute to its flavor and antioxidant properties.
Flavanoids	A subclass of phenolic compounds, specifically flavonoids, contributing to bitterness and antioxidant activity.
Nonflavanoid phenols	Phenolic compounds in the wine that are not flavonoids.
Proanthocyanins	Phenolic compounds related to tannins, influencing the wine's color and mouthfeel.
Color intensity	The intensity or depth of the wine's color.
Hue	The shade or tint of the wine's color.
OD280/OD315 of diluted wines	A ratio of absorbance at 280 nm and 315 nm, often used to measure phenolic concentration and quality.
Proline	An amino acid found in wine, often related to its sweetness and quality.

---

2. DATA PREPROCESSING

4.3 Missing Data Check

Empty cells in the dataframe were checked to check for the presence of missing data.

Code:

```
# Check for missing values
print("\nMissing Values:")
print(wine_dataset.isnull().sum())
```

Console Output:

```
Missing Values:
Class label                     0
Alcohol                         0
Malic acid                      0
Ash                             0
Alcalinity of ash               0
Magnesium                       0
Total phenols                   0
Flavanoids                      0
Nonflavanoid phenols            0
Proanthocyanins                 0
Color intensity                 0
Hue                             0
OD280/OD315 of diluted wines    0
Proline                         0
```

It was determined that there was no missing data in the data set. Therefore, no filling process was applied. If there were missing data, in order to train the model well, I would have to either completely remove the rows with missing data or fill the missing data with the arithmetic mean of the other data.

4.4 Separating Features and Target

Code:

```
X = wine_data.drop("Class label", axis=1)  # Retain only the features
```

Class label: This column contains target labels that are not used in clustering (unsupervised learning).

X: The dataset now consists only of features (Alcohol, Malic acid, etc.) to be used for clustering.

4.5 Standardizing the Features

Features like Proline and Alcohol have different scales and magnitudes. Standardizing ensures all features contribute equally by rescaling them to a mean of 0 and a standard deviation of 1.

Code:

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

fit_transform: Learns the scaling parameters (mean and variance) and applies them to the dataset.

Why standardization?

- Features like Alcohol and Proline differ in magnitude (e.g., Proline values are in the hundreds, while others might be between 0–10).
- KMeans uses Euclidean distance to form clusters, and unstandardized features with large magnitudes dominate the distance calculation.
- Standardization ensures each feature contributes equally to the clustering.

---

3. Testing Different Values for k (Number of Clusters)

Code:

```
1. silhouette_scores = []
2. k_values = range(2, 11)
3.
4. for k in k_values:
5.     kmeans = KMeans(n_clusters=k, random_state=42)
6.     kmeans.fit(X_scaled)
7.     score = silhouette_score(X_scaled, kmeans.labels_)
8.     silhouette_scores.append(score)
```

- **Purpose**: Test different values of k to determine the optimal number of clusters.
- **Range of k**: Clusters are tested from k = 2 to k = 10.
- **Silhouette Score**:
    - Measures how well-separated and compact the clusters are.
    - A higher score indicates better-defined clusters.
- **For Loop**:
    - For each k, a KMeans model is initialized and fitted on the standardized data.
    - The silhouette score is calculated based on the resulting cluster labels and stored in a list.

Why multiple k values?

- KMeans requires the number of clusters (k) as input, but this value is often unknown beforehand.
- Testing different k values allows us to find the optimal number of clusters.

3.1 Visualizing the Silhouette Scores

What is the Silhouette Score?

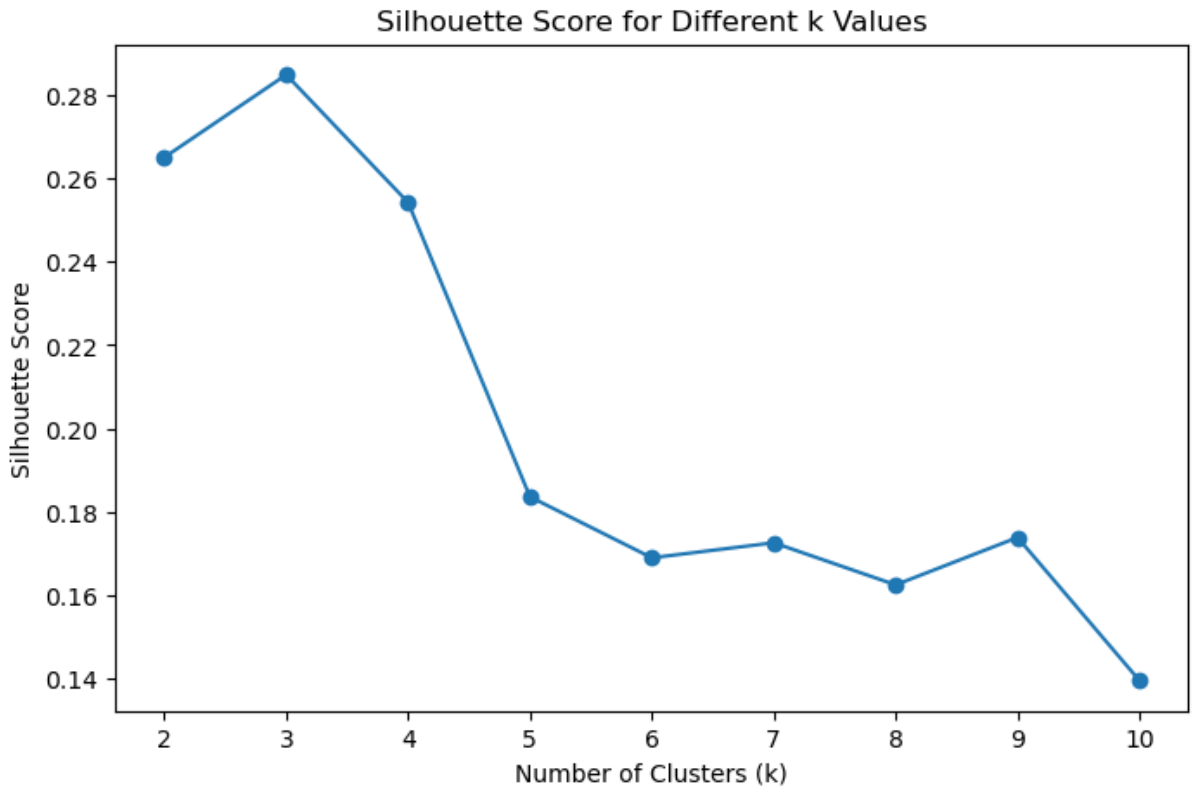
- A measure of how well clusters are defined.
- Ranges from -1 to 1:
    - A score close to 1 means clusters are well-separated and compact.
    - A score close to 0 means clusters overlap.
    - Negative values indicate misclassified points.

Why Silhouette Score?

- It evaluates both cohesion (compactness of clusters) and separation (distance between clusters).
- Alternatives like the **Elbow Method** (plotting within-cluster variance) could also be used but may require subjective judgment, while the Silhouette Score provides a quantifiable metric.

Code:

```
1. plt.figure(figsize=(8, 5))
2. plt.plot(k_values, silhouette_scores, marker="o")
3. plt.title("Silhouette Score for Different k Values")
4. plt.xlabel("Number of Clusters (k)")
5. plt.ylabel("Silhouette Score")
6. plt.show()
```



Line Plot:

- Visualizes the relationship between k and the Silhouette Score.
- Peaks in the plot suggest optimal values for k.

Purpose:

- Helps to visually identify the k that produces the highest Silhouette Score.

We can see that k=3 value produces the highest Silhouette Score.

---

4. Determining the Optimal k and Applying K-Means

Centroid Initialization:

KMeans starts by randomly placing cluster centroids, then iteratively refines them by minimizing the variance within clusters.

Random State:

Ensures reproducibility. The clustering process is stochastic, and setting a seed ensures consistent results across runs.

Code:

```
1. best_k = k_values[np.argmax(silhouette_scores)]
2. kmeans = KMeans(n_clusters=best_k, random_state=42)
3. kmeans.fit(X_scaled)
```

Optimal k:

- The code finds the k value that gives the maximum Silhouette Score using np.argmax().

Fitting KMeans:

- A new KMeans model is created with the optimal number of clusters (best_k).
- The model is trained on the standardized dataset.

The code finds the k value that gives the maximum Silhouette Score as 3.

---

5 Adding Cluster Labels to the Data

The data is clustered into groups, and each point is assigned to a specific cluster.

Code:

```
1. wine_data["Cluster"] = kmeans.labels_
```

Cluster Assignments: Each data point is assigned a cluster label by the KMeans model.

Adding to the DataFrame: The cluster labels are added as a new column, Cluster, in the original dataset.

5.1 Analyzing Cluster Characteristics

Why calculate means?

It gives insight into the defining features of each cluster. For example, Cluster 0 might have higher values for Alcohol and lower values for Malic acid compared to other clusters.

Code:

```
1. group_means = wine_data.groupby("Cluster").mean() print(group_means)
```

Console Output:

```
1.          Class label    Alcohol  ...  OD280/OD315 of diluted wines      Proline
2. Cluster                          ...
3. 0           2.000000  12.250923  ...                      2.803385   510.169231
4. 1           2.941176  13.134118  ...                      1.696667   619.058824
5. 2           1.048387  13.676774  ...                      3.163387  1100.225806
6.
7. [3 rows x 14 columns]
```

Grouping by Clusters:

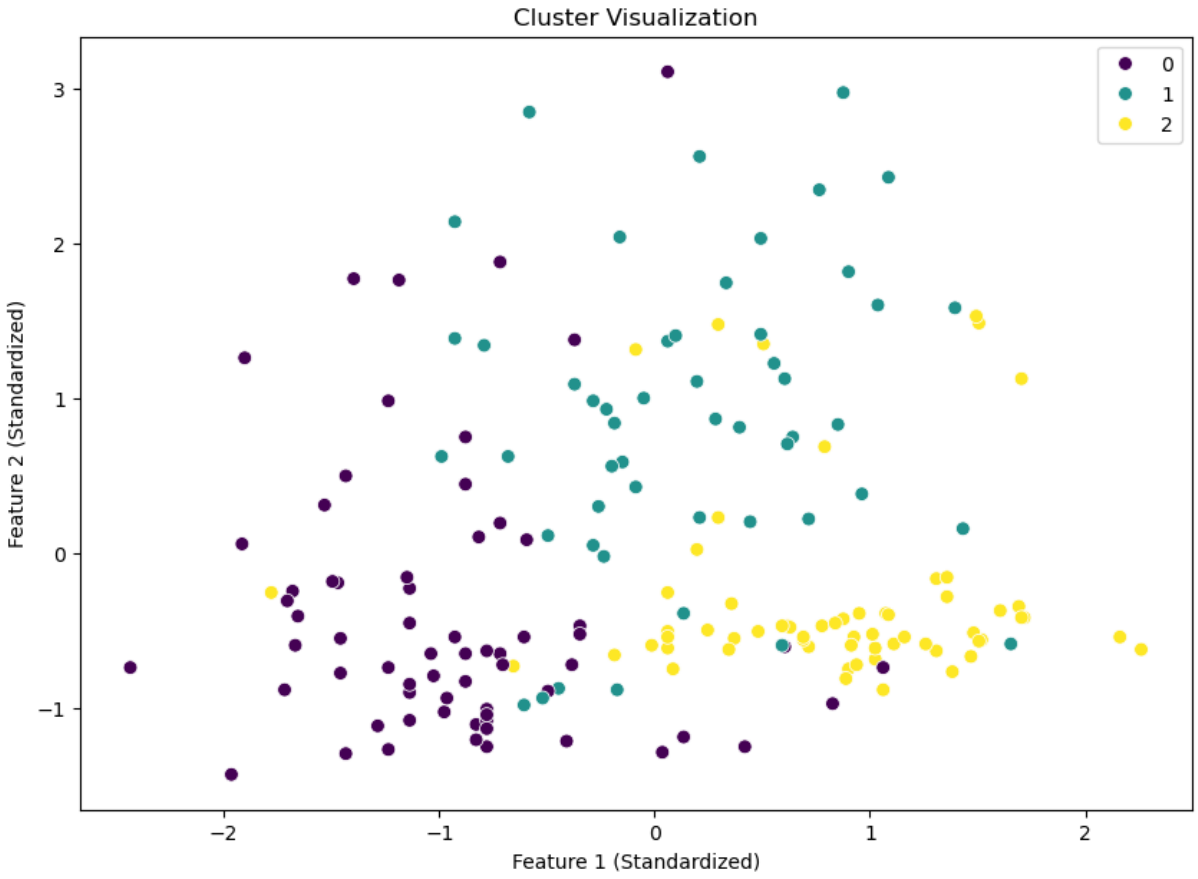
- Groups the dataset by cluster labels.
- Calculates the mean of each feature for each cluster.

Purpose:

- Provides insights into the characteristics of each cluster.
- Helps understand the differences between clusters.

5.2 Visualizing the Clusters

```
1. plt.figure(figsize=(10, 7))
2. sns.scatterplot(
3.     x=X_scaled[:, 0], y=X_scaled[:, 1], hue=kmeans.labels_, palette="viridis", s=50
4. )
5. plt.title("Cluster Visualization")
6. plt.xlabel("Feature 1 (Standardized)")
7. plt.ylabel("Feature 2 (Standardized)")
8. plt.show()
```



Scatter Plot:

- Plots the first two standardized features on the x- and y-axes.
- Points are colored based on their cluster labels.

Purpose:

- Provides a visual representation of how the clusters are distributed.
- Shows if the clusters are well-separated or overlapping.

---

6 CLUSTERING SUMMARY AND RESULTS

1. **Cluster Separation:**
    - The clusters show distinguishable groupings with some overlap between clusters, particularly between clusters 1 (teal) and 2 (yellow).
    - Cluster 0 (purple) forms a more compact and distinct group, suggesting that the wines in this cluster have more homogeneous chemical properties compared to the other two clusters.
2. **Interpreting Overlaps:**

- The overlap between clusters 1 and 2 could indicate similarity in chemical composition for wines in these clusters, suggesting that these wines share common attributes that blur their separation.

3. **Cluster Characteristics:**
    - Each cluster represents a subset of wine samples that share similar patterns in their chemical attributes, such as alcohol content, acidity, or phenolic composition.
    - Cluster centers (not shown in the image) provide the average standardized feature values for each group, enabling deeper analysis of the wine profiles.
4. **Dimensionality Reduction Impact:**
    - PCA has effectively reduced the 13-dimensional dataset to two dimensions for visualization. However, some variance and cluster-specific information might have been lost during this process, which could contribute to the overlaps seen.
5. **Silhouette Scores:**
    - If the clustering evaluation included Silhouette Scores, this could quantitatively validate the visual impression. A higher Silhouette Score for cluster 0 would confirm its tight cohesion, whereas lower scores for clusters 1 and 2 would reflect their overlap.
6. **Insights**:
    - Cluster 0 might represent a distinct wine cultivar with unique characteristics, whereas clusters 1 and 2 might be more similar cultivars with overlapping chemical profiles.
    - Such clustering could guide wineries in understanding chemical similarities among wine samples, leading to potential improvements in production or quality control processes.
7. **Limitations and Next Steps:**
    - The visualization is limited to two dimensions. Examining clustering performance in higher-dimensional space might provide a clearer separation.
    - To validate the clustering, comparing these results against the original class labels (if available) would help evaluate how well the clustering aligns with known groupings.

This analysis provides valuable insights into the underlying structure of the Wine dataset and highlights areas for further exploration, including refinement of clustering methods or integration of domain knowledge to enhance interpretation.

RTEU COMPUTER ENGINEERING DATA MINING HOMEWORK: QUESTION 3

---

1. DATASET

I used boston housing dataset.

Source: BostonHousing.csv

1.2 Reading The Dataset

The Boston Housing dataset is loaded from a URL into a Pandas DataFrame using pd.read_csv. The dataset contains information about housing prices and associated features.

Code:

```
1. url = "https://raw.githubusercontent.com/selva86/datasets/master/BostonHousing.csv"
2. data = pd.read_csv(url)
```

1.5 Dataset Details

Info() function used to get basic information about the dataset.

Code:

```
# Basic dataset information
print("\nDataset Info:")
print(data.info())
```

Console output:

```
Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   crim     506 non-null    float64
 1   zn       506 non-null    float64
 2   indus    506 non-null    float64
 3   chas     506 non-null    int64
 4   nox      506 non-null    float64
 5   rm       506 non-null    float64
 6   age      506 non-null    float64
 7   dis      506 non-null    float64
 8   rad      506 non-null    int64
 9   tax      506 non-null    int64
 10  ptratio  506 non-null    float64
 11  b        506 non-null    float64
 12  lstat    506 non-null    float64
 13  medv     506 non-null    float64
```

Attribute	Description
Crim	Per capita crime rate by town.
Zn	Proportion of residential land zoned for lots over 25,000 sq. ft.
Indus	Proportion of non-retail business acres per town.
chas	Charles River dummy variable (1 if tract bounds river; 0 otherwise).
nox	Nitric oxide concentration (parts per 10 million).
rm	Average number of rooms per dwelling.
Age	Proportion of owner-occupied units built prior to 1940.
Dis	Weighted distances to five Boston employment centers.
Rad	Index of accessibility to radial highways.
tax	Full-value property tax rate per $10,000.
Ptratio	Pupil-teacher ratio by town.
b	1000(Bk - 0.63)^2, where Bk is the proportion of Black people by town.
lstat	Percentage of lower-status population.
medv	Median value of owner-occupied homes in $1000s (target variable).

---

2. DATA PREPROCESSING

4.6 Missing Data Check

Empty cells in the dataframe were checked to check for the presence of missing data.

Code:

```
print("\nMissing Values:")
print(data.isnull().sum())
```

Console Output:

```
Missing Values:
crim       0
zn         0
indus      0
chas       0
nox        0
rm         0
age        0
dis        0
rad        0
tax        0
ptratio    0
b          0
lstat      0
medv       0
dtype: int64

```

It was determined that there was no missing data in the data set. Therefore, no filling process was applied. If there were missing data, in order to train the model well, I would have to either completely remove the rows with missing data or fill the missing data with the arithmetic mean of the other data.

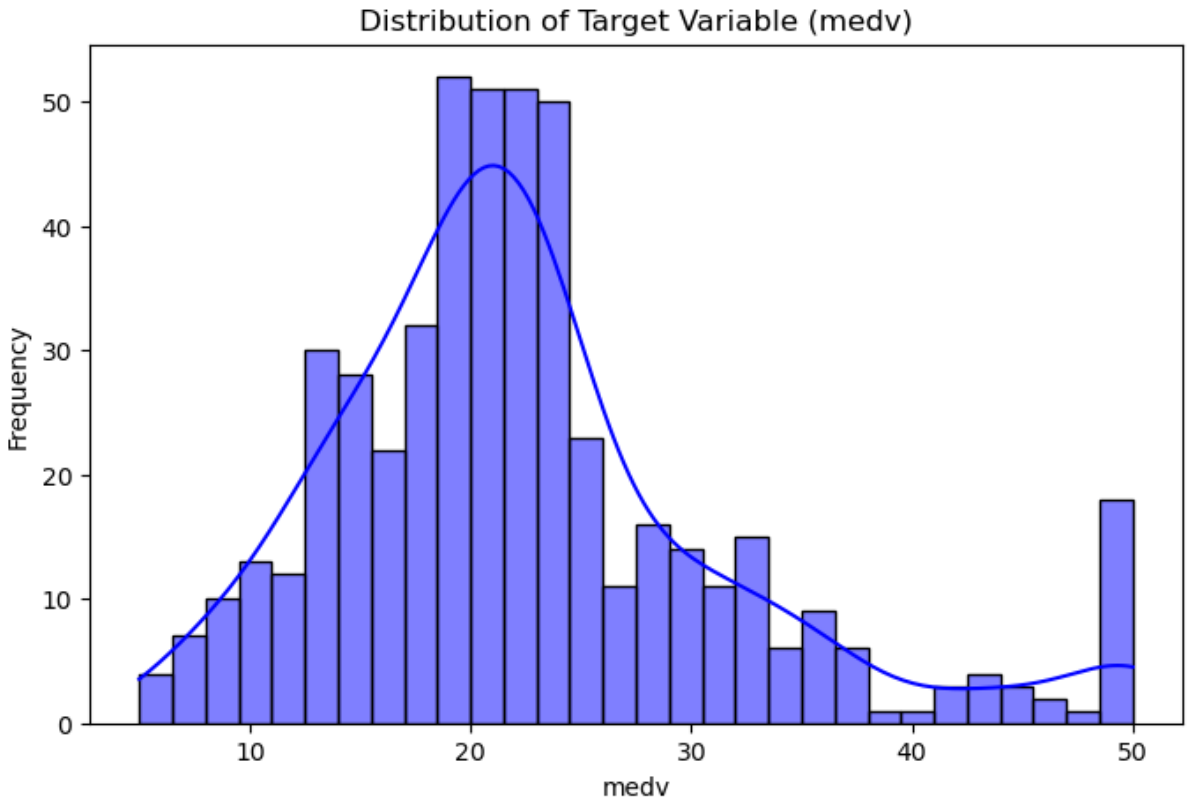
4.7 Distribution of target variable

The target variable (medv) is visualized using a histogram with a Kernel Density Estimate (KDE) to show its distribution.

Purpose: Understand the spread of housing prices (medv).

Code:

```
# Distribution of target variable
plt.figure(figsize=(8, 5))
sns.histplot(data["medv"], kde=True, bins=30, color="blue")
plt.title("Distribution of Target Variable (medv)")
plt.xlabel("medv")
plt.ylabel("Frequency")
plt.show()
```



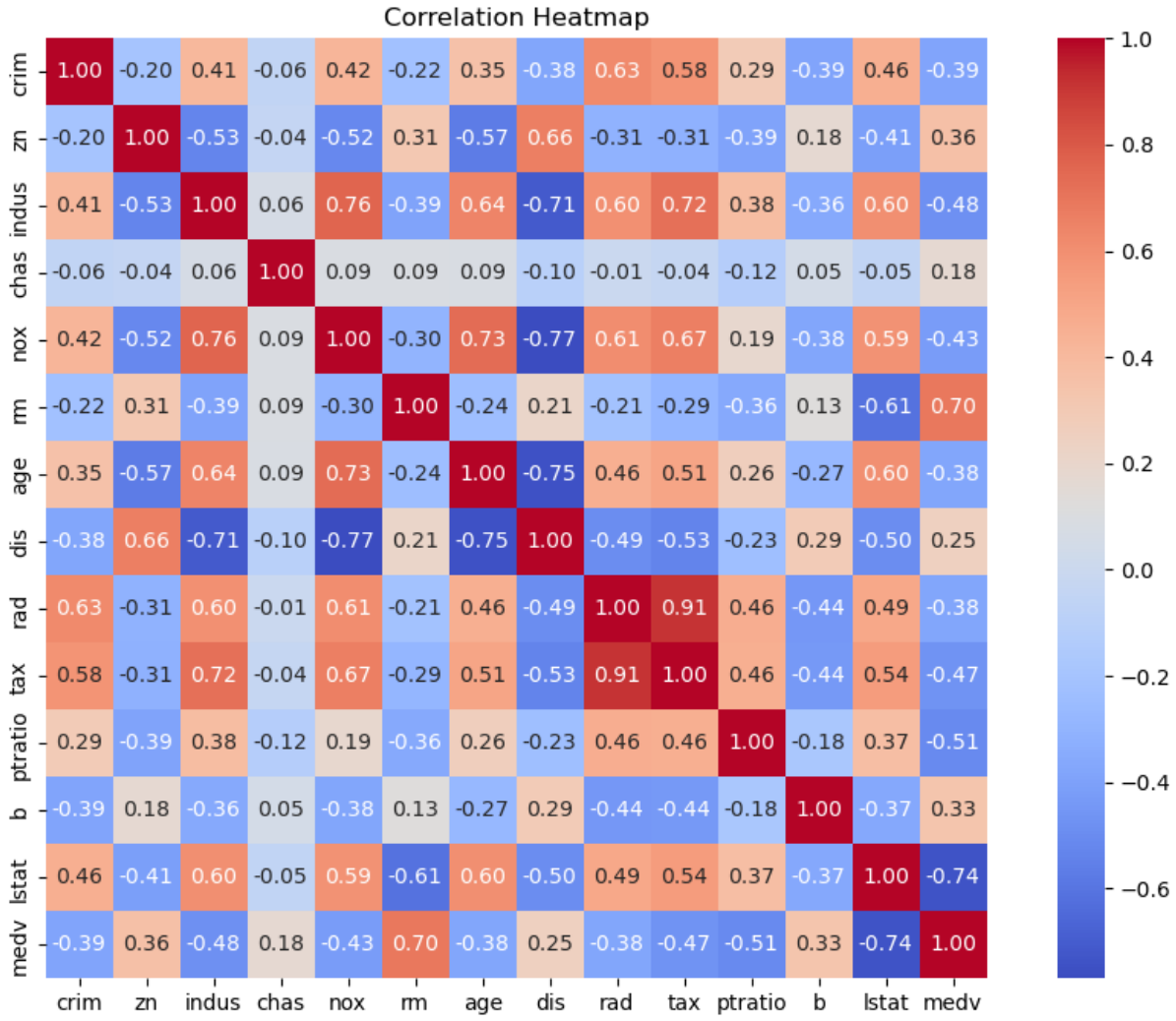
4.8 Correlation Analysis

A heatmap shows correlations between all features in the dataset.

Purpose: Identify relationships between features and the target variable, helping with feature selection.

Code:

```
1. # Correlation heatmap
2. plt.figure(figsize=(12, 8))
3. sns.heatmap(data.corr(), annot=True, fmt=".2f", cmap="coolwarm", square=True)
4. plt.title("Correlation Heatmap")
5. plt.show()
```



Strong positive correlation:

- rm (0.70): rm is strongly positively correlated with medv. Homes with more rooms tend to have higher median values.

Strong negative correlation:

- lstat (-0.74): lstat has a strong negative impact on medv. Areas with higher poverty rates tend to have lower home values.
- ptratio (-0.51): ptratio is moderately negatively correlated with medv, suggesting that higher ratios are associated with lower property values.
- dis (-0.50): dis is negatively correlated, indicating homes closer to such centers tend to have higher values.

Variables with Weak Correlation to the Target (medv)

Variables with weak correlations to the target variable may contribute little to the model's predictive power. These variables could potentially be excluded:

- **chas (0.18)**:
    - Whether the property is near the Charles River (chas) shows a weak positive correlation. If it doesn't add significant value, it can be removed.
- **zn (0.36)**:
    - The proportion of residential land zoned for lots over 25,000 sq. ft. (zn) has a weak positive correlation with medv. Its contribution to the model might be minimal.
- **age (-0.38)**:
    - The proportion of older homes (age) has a weak negative correlation. It might not provide meaningful insights for predicting medv.

This analysis suggests a simplified feature set by removing redundant or weakly correlated variables. For example, I might remove rad, tax, chas, and zn but it's not necessary at all.

4.9 Splitting Features and Target Variable

The dataset is split into features (X) and the target variable (y), where medv (median value of owner-occupied homes) is the target.

Code:

```
1. # Split features and target variable
2. X = data.drop(columns=["medv"])  # Features
3. y = data["medv"]                 # Target
```

4.10 Train-Test Split

The data is split into training and testing sets:

- Training set (80%): Used to train the models.
- Testing set (20%): Used to evaluate the models.
- random_state=42 ensures reproducibility.

Code:

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

4.11 Feature Scaling

Standardization is applied to the features for linear models:

- The mean is subtracted, and the values are divided by the standard deviation.
- Ensures features are on the same scale, improving model performance for linear models.

Code:

```
1. # Feature scaling for linear models
2. scaler = StandardScaler()
3. X_train_scaled = scaler.fit_transform(X_train)
4. X_test_scaled = scaler.transform(X_test)
```

---

5 MODEL OPERATIONS

5.1 Initializing Models

Four regression models are initialized:

- Linear Regression: A baseline model.
- Ridge and Lasso Regression: Regularized versions of linear regression.
- Random Forest Regression: An ensemble model using decision trees.

Code:

```
1. # Initialize models
2. models = {
3.     "Linear Regression": LinearRegression(),
4.     "Ridge Regression": Ridge(),
5.     "Lasso Regression": Lasso(),
6.     "Random Forest Regression": RandomForestRegressor(random_state=42)
7. }
```

5.2 Training and Evaluating Models

Each model is trained on the training data and tested on the test data.

Scaling is applied for linear models but not for Random Forest (tree-based models don't require scaling).

Metrics Calculated:

- **MAE (Mean Absolute Error):** Measures average prediction error.
- **MSE (Mean Squared Error):** Penalizes large errors more than MAE.
- **R^2 (R-squared):** Indicates the proportion of variance explained by the model.

Code:

```
1. # Train and evaluate models
2. results = []
3. for name, model in models.items():
4.     if "Regression" in name and name != "Random Forest Regression":
5.         model.fit(X_train_scaled, y_train)
6.         y_pred = model.predict(X_test_scaled)
7.     else:
```

```
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    results.append({"Model": name, "MAE": mae, "MSE": mse, "R²": r2})
```

6 MODEL PERFORMANCES

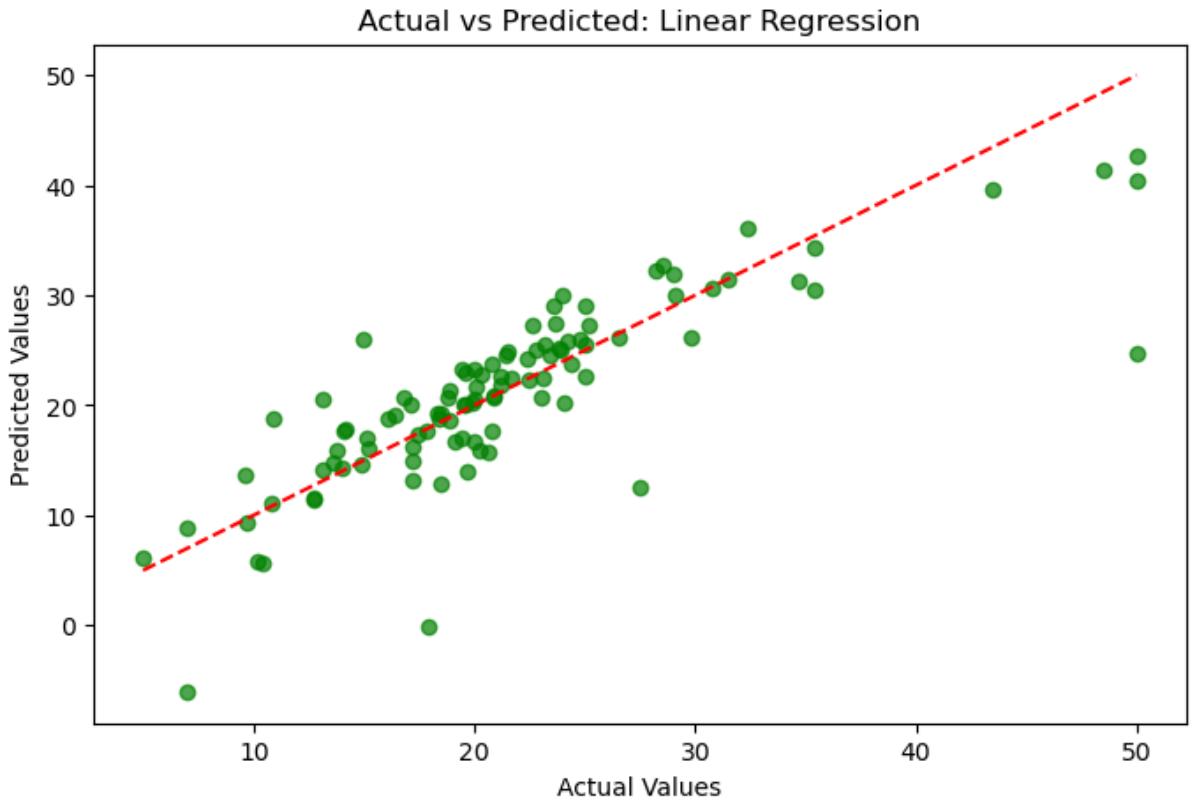
Code:

```
    # Scatter plot for actual vs predicted
    plt.figure(figsize=(8, 5))
    plt.scatter(y_test, y_pred, alpha=0.7, color="green")
    plt.title(f"Actual vs Predicted: {name}")
    plt.xlabel("Actual Values")
    plt.ylabel("Predicted Values")
    plt.plot([y.min(), y.max()], [y.min(), y.max()], color="red", linestyle="--")
    plt.show()
# Convert results to DataFrame
results_df = pd.DataFrame(results)
print("\nModel Performance:")
print(results_df)
```

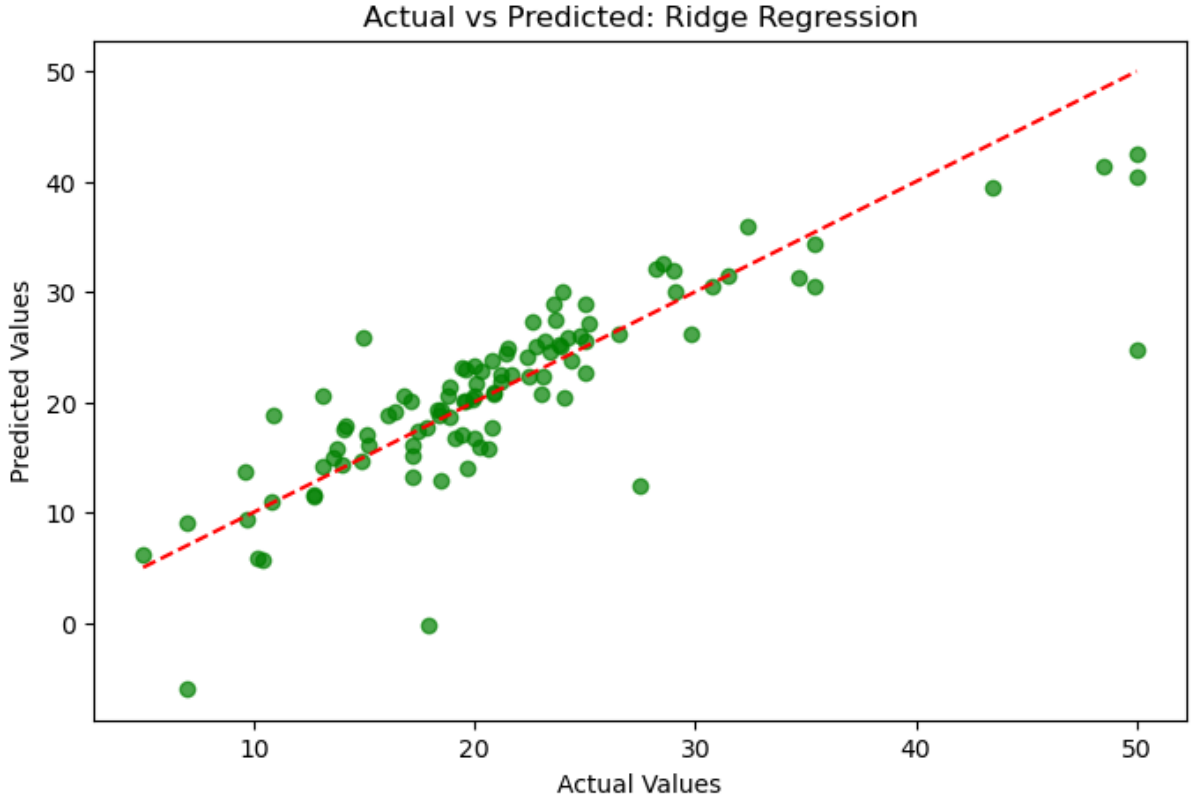
Console Output:

```
Model Performance:
                        Model       MAE        MSE        R²
0         Linear Regression  3.189092  24.291119  0.668759
1          Ridge Regression  3.185724  24.312904  0.668462
2          Lasso Regression  3.473770  27.577692  0.623943
3  Random Forest Regression  2.039539   7.901514  0.892253
```

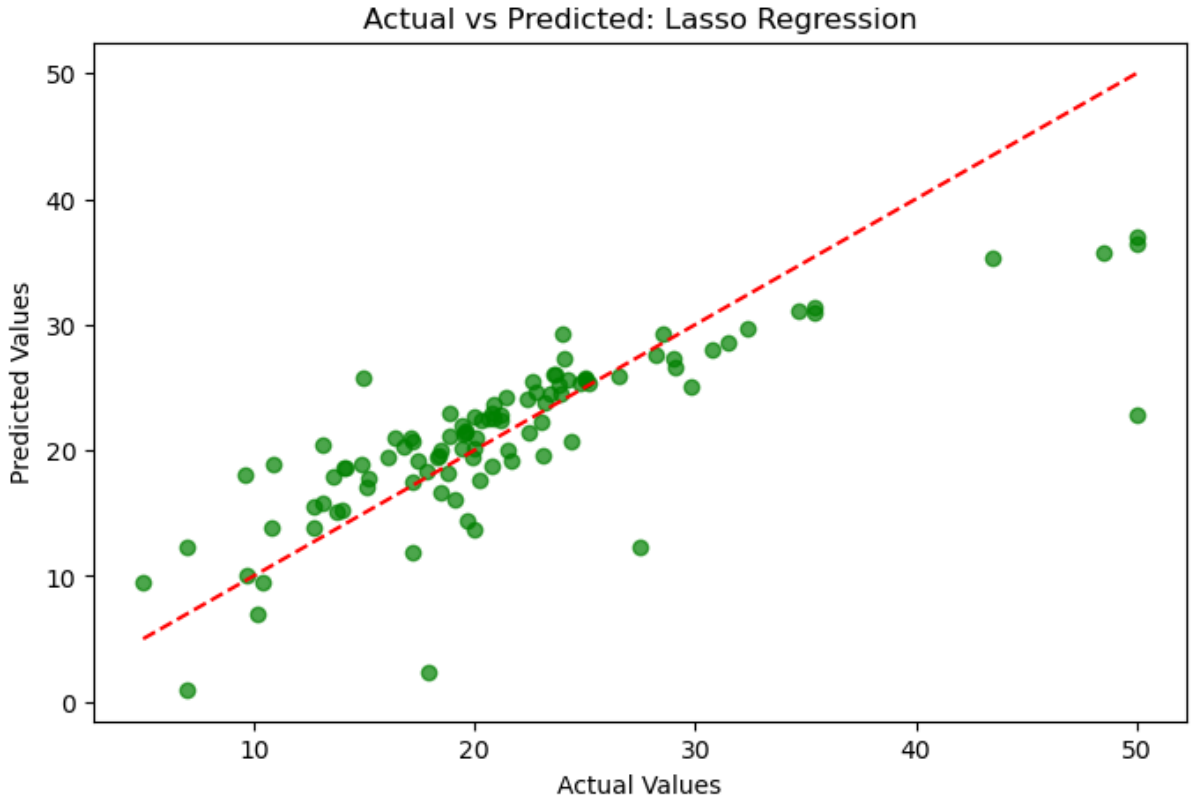
Linear Regression



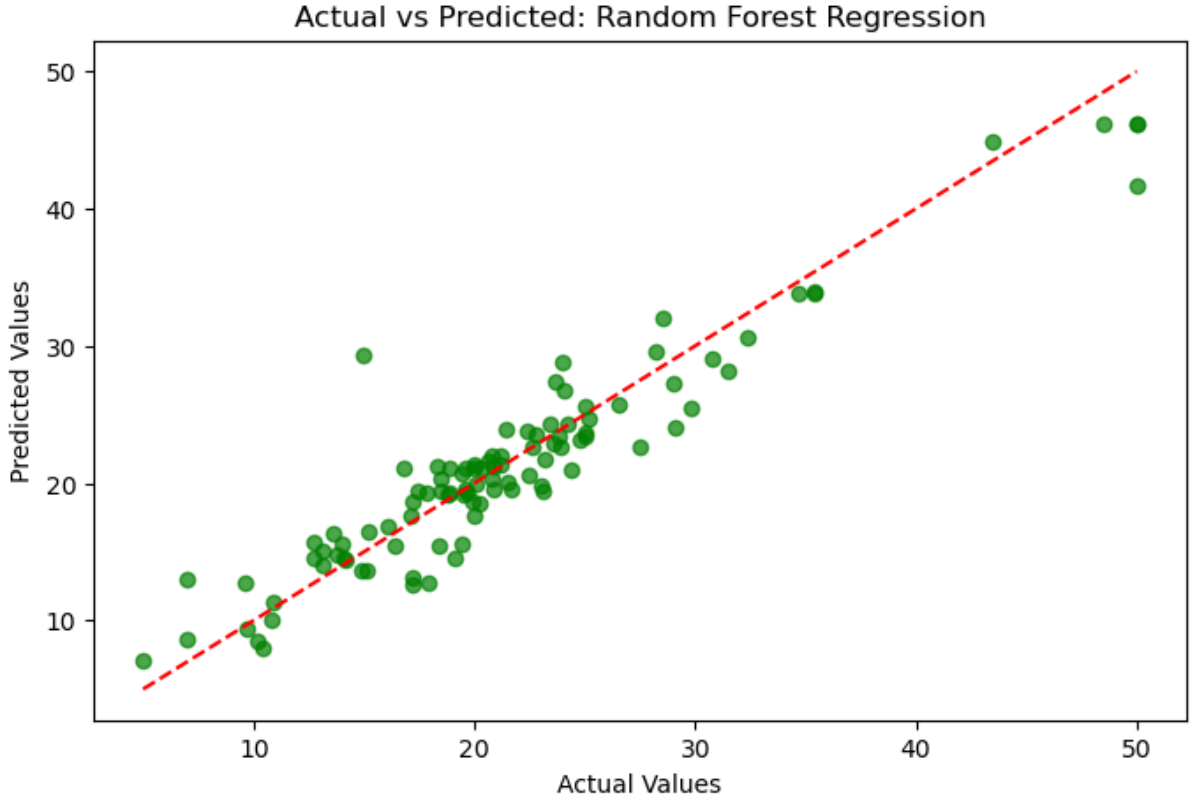
Ridge Regression



Lasso Regression



Random Forest Regression



Mean Absolute Error (MAE):

- **Definition**: MAE measures the average magnitude of errors in predictions, without considering their direction. Lower MAE values indicate better predictive accuracy.
- **Observation**:
    - Random Forest Regression has the lowest MAE (2.04), meaning its predictions are closest to the true values on average.
    - Lasso Regression has the highest MAE (3.47), indicating it performs the worst among the models in terms of error magnitude.

Mean Squared Error (MSE):

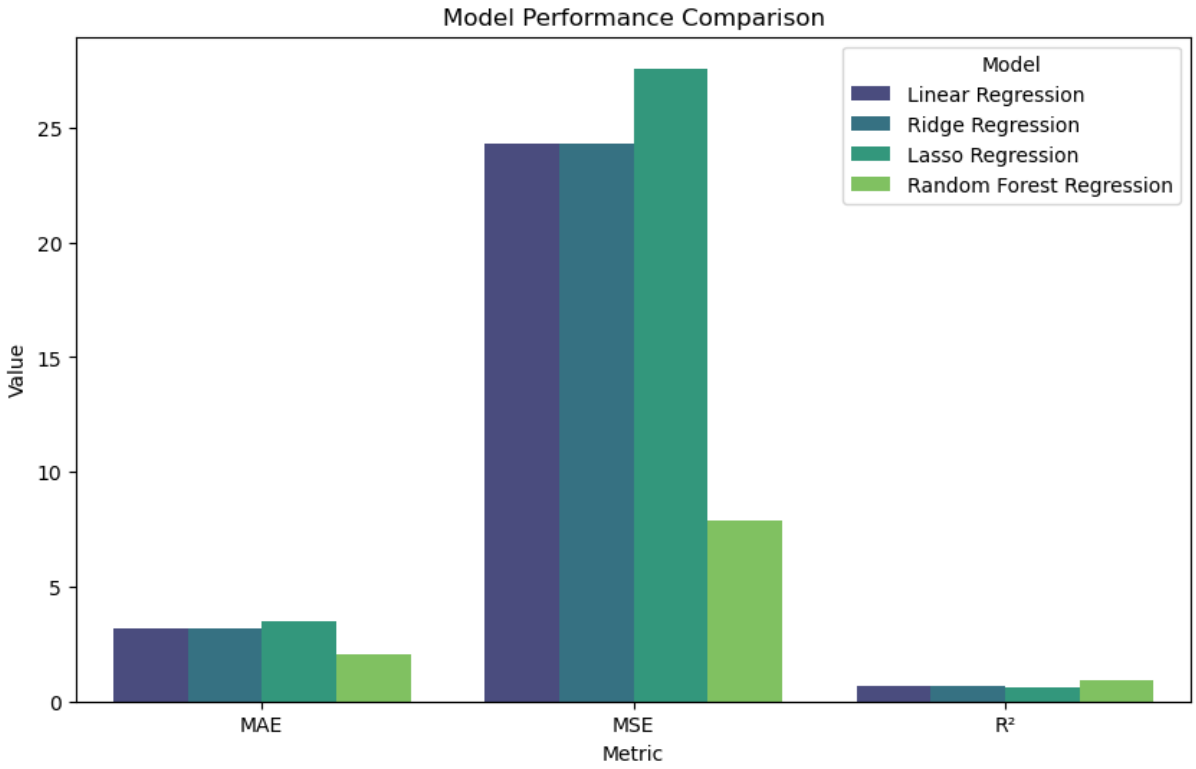
- **Definition**: MSE measures the average squared difference between actual and predicted values, heavily penalizing large errors. Lower MSE values suggest better performance.
- **Observation**:
    - Random Forest Regression significantly outperforms other models with an MSE of 7.90, compared to 24.29–27.58 for the others.
    - Linear and Ridge Regression are similar in performance (MSE around 24.3), while Lasso Regression again performs the worst.

R-squared (R^2):

- **Definition**: R^2 represents the proportion of variance in the target variable explained by the model. Values closer to 1 indicate better fit.
- **Observation**:
    - Random Forest Regression has the highest R^2 (0.892), explaining nearly 89% of the variance in the target variable, indicating a strong predictive ability.
    - Linear and Ridge Regression have similar R^2 values (around 0.669), performing moderately well.
    - Lasso Regression lags with the lowest R^2 (0.624), indicating it explains only 62% of the variance.

Code:

```
1. # Visualize model comparison
2. plt.figure(figsize=(10, 6))
3. results_df_melted = results_df.melt(id_vars="Model", var_name="Metric", value_name="Value")
4. sns.barplot(x="Metric", y="Value", hue="Model", data=results_df_melted, palette="viridis")
5. plt.title("Model Performance Comparison")
6. plt.show()
```



7 CONCLUSION

Random Forest Regression emerges as the best-performing model across all metrics, with the lowest MAE and MSE and the highest R^2. This suggests that its ensemble-based approach effectively captures complex relationships in the data, leading to superior predictive performance. Linear and Ridge Regression perform moderately and are nearly identical in performance, likely because Ridge includes regularization, which minimally impacts this dataset. Lasso Regression shows the weakest performance, likely due to its tendency to shrink coefficients of less important features to zero, which may have oversimplified the model for this dataset. Thus, for predicting housing prices, Random Forest Regression is clearly the most suitable choice.