



**T.C.**

**RECEP TAYYİP ERDOĞAN UNIVERSITY  
FACULTY OF ENGINEERING AND ARCHITECTURE  
DEPT. OF COMPUTER ENGINEERING**

**DATA MINING FALL TERM PROJECT**

Performance Comparison of Decision Tree and Random Forest  
Classifiers: Titanic Dataset

LECTURER: DR. ABDULGANİ KAHRAMAN

Bedirhan ÖZÇELİK

201401055

**RİZE 2024**

## TABLE OF CONTENTS

1. Introduction .....	3
2. Environment Setup and Library Installation.....	4
3. Dataset Selection and Analysis .....	5
4. Data Preprocessing .....	8
5. Modeling Process .....	12
6. Model Evaluation .....	14
7. Results .....	18
8. References.....	20

## 1) Introduction

### 1.1 Project Objective

The aim of this project is to predict a target variable using a Decision Tree Classifier and a Random Forest Classifier while analyzing their performance. During the analysis, the effects of data preparation, hyperparameter tuning, and performance metrics are thoroughly examined.

The Titanic dataset is used for this analysis to model the target variable Survived (survival status) and compare the performance of a Decision Tree model and a Random Forest model.

### 1.2 Problem Definition

The following research question has been addressed as part of this analysis:

- Is it possible to predict the target variable (survival status) using a Decision Tree model?
- What are the differences in the performance metrics (accuracy, precision, recall, F1 score, etc.) between the Decision Tree model and the Random Forest model?

To answer these questions, the following steps were performed:

1. Trained the Decision Tree Classifier and Random Forest Classifier.
2. Conducted analysis on missing values and categorical variables in the dataset and performed necessary preprocessing.
3. Optimized hyperparameters for both models and conducted training on the preprocessed data.
4. Evaluated both models' performance metrics and compared their results through visualization.

### 1.3 General Explanation of the Algorithms Used

**Decision Tree Classifier:** Decision Trees split data based on feature conditions, creating a flowchart-like structure that determines decisions. They are widely used for classification and regression problems due to their simplicity and interpretability.

**Random Forest Classifier:** Random Forests are an ensemble learning method that uses a combination of multiple decision trees. They reduce overfitting by averaging the outputs of many decision trees and ensure better generalization performance compared to a single Decision Tree.

In this report, the performance comparison of these two models was conducted to determine which model performed better under given conditions.

## 2) Environment Setup and Library Installation

### 2.1 Setting up the Python Environment

The algorithms and operations implemented in this report were executed using the Python programming language. The version of Python used is as follows:

```
PS C:\Users\Lenovo> Python --version
Python 3.10.2
```

Figure 1: Python Version

### 2.2 Required Libraries

The machine learning models and data visualization tasks rely on the following Python libraries:

Library	Purpose
<b>pandas</b>	Used for data manipulation and preprocessing tasks.
<b>numpy</b>	Used for numerical operations and mathematical computations.
<b>scikit-learn</b>	Provides machine learning algorithms and model evaluation metrics.
<b>matplotlib</b>	For visualization of graphs and plots.
<b>seaborn</b>	Enhances visualization aesthetics and enables statistical visualizations.

These libraries are critical for implementing the Decision Tree, Random Forest models, and their evaluation, as well as for preprocessing and visualization.

### 2.3 Installation Instructions

Before running the code, ensure that all required dependencies are installed. These dependencies can be installed using the following command:

➤ `pip install pandas numpy scikit-learn matplotlib seaborn`

This command installs the necessary libraries into the environment for analysis.

### 3) Dataset Selection and Analysis

#### 3.1 Dataset Overview

The dataset utilized in this analysis is the Titanic dataset, which is a widely used benchmark dataset for classification problems in machine learning. This dataset contains information about passengers aboard the Titanic and is commonly used to predict whether a passenger survived or did not survive based on various features.

The Titanic dataset provides insights into real-world classification tasks, featuring a mix of numerical and categorical variables that require preprocessing and feature engineering for effective model performance. This section will detail the structure, variables, and context of the selected dataset.

#### 3.2 Dataset Source

The dataset was sourced from the well-known Kaggle Titanic dataset repository. The link to the dataset is provided below:

<https://www.kaggle.com/c/titanic>

This dataset is publicly available, widely validated, and contains real-world passenger data for machine learning experimentation.

#### 3.3 Reason for Dataset Selection

The Titanic dataset was selected for this study because:

- **Relevance to Classification Tasks:** The target variable (Survived) represents a clear binary classification problem (0 = not survived, 1 = survived), which is ideal for supervised classification techniques such as Decision Trees and Random Forests.
- **Feature Diversity:** The dataset contains a mix of numerical features (e.g., Age, Fare) and categorical features (e.g., Sex, Embarked), making it suitable for preprocessing, feature selection, and encoding methods.
- **Prevalence in Machine Learning Education:** The Titanic dataset is a standard dataset for learning and experimenting with classification algorithms, offering sufficient complexity without overwhelming computational resources.
- **Demonstrative Analysis:** By using this dataset, we could demonstrate the effects of preprocessing (e.g., handling missing data and encoding), hyperparameter tuning, feature scaling, and model evaluation using real-world-like scenarios.

### 3.4 Dataset Characteristics

The Titanic dataset includes both training and testing subsets with specific columns relevant to the survival prediction problem. Key columns include:

- **Pclass:** Class of the cabin (1st, 2nd, or 3rd class) of the passenger.
- **Sex:** Gender of the passenger.
- **Age:** Age of the passenger (numerical, with missing values).
- **SibSp:** Number of siblings/spouses aboard the Titanic with the passenger.
- **Parch:** Number of parents/children aboard the Titanic with the passenger.
- **Fare:** The amount paid by the passenger for the ticket (numerical, with missing values).
- **Embarked:** Port of embarkation (categorical: C = Cherbourg, Q = Queenstown, S = Southampton).
- **Survived:** Target variable, where 0 = not survived and 1 = survived.

The dataset's training set is used to train the machine learning models, while the testing set serves as an evaluation benchmark to determine the predictive performance of the trained models.

### 3.5 Data Details

The Titanic dataset contains 891 rows in the training set and 418 rows in the test set, as per the provided Kaggle dataset split. Missing values exist in some columns, particularly Age, Fare, and Embarked. Handling these missing values was critical for preprocessing and ensured the robustness of machine learning model performance.

Key observations about the dataset's features:

Feature	Type	Missing Values	Description
Pclass	Numerical	0	Class of the cabin (1st, 2nd, or 3rd class).
Sex	Categorical	0	Gender of the passenger (male or female).
Age	Numerical	Present (some missing values)	Age of the passenger.
SibSp	Numerical	0	Number of siblings or spouses aboard.
Parch	Numerical	0	Number of parents or children aboard.
Fare	Numerical	Present (some missing values)	The fare paid by the passenger.
Embarked	Categorical	Present (some missing values)	Port of embarkation (C, Q, or S).
Survived	Binary	0	Target variable: 0 = not survived, 1 = survived.

### 3.6 Data Preprocessing Steps

To ensure that the machine learning models could effectively analyze the data, several preprocessing steps were undertaken. These steps include:

#### Handling Missing Values:

- Missing values in Age were filled using the median value of the respective column to ensure the distribution's central tendency was preserved.
- Missing values in Embarked were imputed using the most common value (mode) to maintain consistency.
- The Cabin column was dropped due to its high degree of missing values, reducing complexity without meaningful information.

#### Encoding Categorical Variables:

- Categorical variables such as Sex and Embarked were label-encoded using scikit-learn's LabelEncoder. This step was necessary for machine learning algorithms, as they cannot process raw categorical text values.

#### Scaling Numerical Features:

- Continuous features such as Age and Fare were scaled using the StandardScaler. This ensures that their ranges are comparable and avoids bias during model training.

### 3.7 Visualizing the Dataset

Several visualizations were generated to provide insights into the data distribution and relationships among the features:

- **Distribution of Survived vs. Not Survived Passengers:** This visualization helps assess the class imbalance between passengers who survived and those who did not.
- **Correlation Heatmap:** This heatmap displays how strongly correlated each feature is with the target variable (Survived) and other predictors.
- **Boxplots for Numerical Features:** These plots explore the distribution of numerical features like Age and Fare across survival status groups.

### 3.8 Dataset Limitations

While the Titanic dataset provides a valuable resource for analysis, there are inherent limitations:

- **Missing Data:** Some features, especially numerical ones like **Age** and **Fare**, had missing values that required careful imputation.
- **Class Imbalance:** The number of non-survived passengers was greater than the number of survived passengers, introducing class imbalance.

- **Feature Constraints:** The Titanic dataset does not account for all possible real-world factors that may influence survival, limiting its representativeness.

## 4) Data Preprocessing

Data preprocessing is an essential step in the machine learning workflow, as raw data is often incomplete, noisy, or inconsistent. Proper preprocessing ensures that the data is properly formatted, cleansed, and transformed into a form suitable for machine learning algorithms. This section provides a detailed explanation of each preprocessing step applied in the study, with mathematical formulations where appropriate. The primary preprocessing stages included in this study are handling missing data, encoding categorical variables, feature scaling, and data and feature selection.

### 4.1 Handling Missing Data

Missing data is a common issue in real-world datasets and must be appropriately managed to avoid bias or loss of information during model training. Missing data can lead to incorrect results, unreliable models, and wasted computation. Handling missing values typically involves imputing them using statistical measures or removing features/observations with too many missing entries.

#### Approach to Handle Missing Data

The study used the following strategies:

- **Imputing Missing Numerical Data with Median:**

Missing numerical values were imputed using the median instead of the mean to avoid bias due to outliers. The formula for median imputation is as follows:

$$\text{Medianvalue} = \text{median}(X)$$

For instance, missing entries in the **Age** variable were imputed using the median value:

➤ `train_data['Age'].fillna(train_data['Age'].median(), inplace=True)`

- **Dropping Features with High Missing Data Rate:** Features with excessively high proportions of missing entries were removed from the dataset entirely, as their presence could add noise and lead to unreliable training. For example, the *Cabin* column was dropped because it contained a high percentage of missing values.

➤ `train_data.drop(columns=['Cabin'], inplace=True)`



- **Imputing Missing Categorical Data with Mode:** Missing categorical entries were filled using the mode, defined as the most frequently occurring category in a given variable. This ensures that missing entries are replaced with a statistically sound choice that minimizes the distortion of the categorical variable distribution.

$\text{Mode}(X) = \text{most frequent value in } X$

Example for imputing missing values in the Embarked variable:

- `train_data['Embarked'].fillna(train_data['Embarked'].mode()[0], inplace=True)`

These steps ensured that the dataset was free of NaN (missing) values and maintained consistency for the learning process.

## 4.2 Processing Categorical Variables

Machine learning models cannot directly process categorical variables because they are non-numeric. Thus, categorical data must be encoded numerically while maintaining their inherent relationships. Encoding methods transform categorical variables into numeric representations.

### Label Encoding

The study used Label Encoding for transforming categorical features into numeric values.

Label encoding assigns a unique integer to each category, as shown in the formula:

$\text{Encoded Value (for category)} = \text{unique integer identifier}$

- **Encoding the 'Sex' Feature:** The Sex variable was transformed into binary numeric values (0 and 1) to represent categories male and female:
  - `le_sex = LabelEncoder()`
  - `train_data['Sex'] = le_sex.fit_transform(train_data['Sex'])`
- **Encoding the 'Embarked' Feature:** Similarly, the Embarked variable was encoded to ensure compatibility with machine learning models:
  - `le_embarked = LabelEncoder()`
  - `train_data['Embarked'] =`  
`le_embarked.fit_transform(train_data['Embarked'])`

By encoding categorical variables numerically, these features could now be fed into machine learning algorithms for training.

### 4.3 Feature Scaling

Feature scaling ensures that input features are on the same scale or range, allowing machine learning algorithms to treat all features equally during the learning process. Algorithms like Decision Trees and Random Forests are sensitive to feature magnitudes unless scaling is applied properly.

#### Standardization

Standardization (also known as Z-score normalization) was applied to numerical features. The mathematical formulation for standardization is given by:

$$X_{\text{scaled}} = (X - \mu) / \sigma$$

$X$  = original feature value

$\mu$  = mean of the feature

$\sigma$  = standard deviation of the feature

This ensures the data has a mean of 0 and a standard deviation of 1.

#### Implementation for scaling numerical features like Age and Fare:

```
scaler = StandardScaler()
```

```
X[['Age', 'Fare']] = scaler.fit_transform(X[['Age', 'Fare']])
```

- $X[['Age', 'Fare']]$  = the selected columns to scale.
- `scaler.fit_transform()` standardizes these features to the specified formula above.

This step ensures that the machine learning models can converge more efficiently during optimization.

### 4.4 Data and Feature Selection

Feature selection involves identifying the most relevant features that contribute to model training while excluding irrelevant or redundant features. This reduces the complexity of the model and mitigates overfitting.

#### Feature Selection Methods

The study used domain knowledge and statistical analysis to select the most relevant features:

1. **Exploratory Data Analysis (EDA):** Features with strong relationships with the target variable Survived were selected based on visualization techniques like heatmaps and correlation analysis.
2. **Correlation Analysis:** Correlation was computed using the Pearson correlation coefficient:

$$\text{Pearson Correlation Coefficient} = \text{cov}(X, y) / \sigma_X \sigma_y$$

This measures the linear relationship between each feature  $X$  and the target variable  $y$ .

**3. Relevant Features Chosen:** Based on correlation, EDA, and domain knowledge, the study selected the following features as inputs:

- **Pclass:** Passenger class (1, 2, 3)
- **Sex:** Gender encoding (0 for male, 1 for female)
- **Age:** Age of the passenger
- **SibSp:** Number of siblings/spouses aboard
- **Parch:** Number of parents/children aboard
- **Fare:** Amount of fare paid
- **Embarked:** Encoded embarkation point

#### **Feature Selection Formula**

The selected features are mathematically optimized by comparing their correlation coefficients with the target variable, as follows:

$$\text{Correlation} = \text{cov}(X,y) / \sigma_X \sigma_y$$

Features with a significant correlation ( $|\text{Correlation}| > \text{threshold}$ ) are retained.

#### **Implementation for feature selection:**

```
X = train_data[['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked']]  
y = train_data['Survived']
```

The variables X and y represent the input features and target variable used in the classification task, respectively. In this study, X comprises the selected features that provide critical information about each passenger, while y contains the target labels indicating whether a passenger survived ( $y=1$ ) or not ( $y=0$ ).

The target variable y, "Survived," is a binary categorical variable explicitly designed for the supervised classification task. By mapping the input features X to the corresponding target labels y, the machine learning models aim to learn patterns and relationships within the dataset that differentiate between survivors and non-survivors.

This formulation ensures that the model is provided with rich, meaningful, and minimally redundant information, which is crucial for building a robust and interpretable predictive model.

## 5) Modeling Process

### Definition of the Models

In this study, two machine learning algorithms were employed to classify the Titanic dataset's survival outcomes: Decision Tree Classifier and Random Forest Classifier.

1. **Decision Tree Classifier:** A Decision Tree is a tree-structured algorithm where internal nodes represent features (attributes), branches represent decision rules, and each leaf node represents the outcome. This model splits the data based on feature conditions, aiming to maximize the information gain or minimize the Gini impurity. The algorithm's ability to capture non-linear relationships and provide interpretability made it a suitable candidate for initial experimentation.
2. **Random Forest Classifier:** Random Forest is an ensemble learning method that combines multiple decision trees to improve the overall predictive performance and reduce overfitting. By using bootstrap sampling and random feature selection, it creates a diverse set of weak learners, aggregating their predictions through majority voting for classification tasks.

### Model Selection Process

The models were selected based on their suitability for tabular data and the ability to handle mixed feature types. Additionally, both models are interpretable to a degree, aiding in understanding feature importance and decision boundaries. Random Forest was chosen as an advanced ensemble method to compare its performance against the simpler Decision Tree.

### Training and Testing

The training and testing process was conducted in the following steps:

1. **Data Splitting:** The dataset was divided into training (80%) and testing (20%) subsets using `train_test_split()` to ensure an unbiased evaluation of model performance. This split ensures that the model generalizes well to unseen data.

```
# 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)
```

Figure 2

2. **Hyperparameter Tuning:** Both models were optimized using grid search techniques to identify the best hyperparameters. The following parameters were tuned:
  - **Decision Tree Classifier:** Maximum depth (`max_depth`), minimum samples per split (`min_samples_split`), and minimum samples per leaf (`min_samples_leaf`).

- **Random Forest Classifier:** Number of trees (n\_estimators), maximum depth, and split/leaf conditions.

Example for the Decision Tree Classifier grid search:

```
# Hyperparameter tuning for Decision Tree (optimized grid)
param_grid_dt = {
    'max_depth': [3, 5, 10], # Reduced range
    'min_samples_split': [2, 5], # Fewer options
    'min_samples_leaf': [1, 2], # Fewer options
    'criterion': ['gini'] # Single criterion
}

grid_search_dt = GridSearchCV(
    estimator=DecisionTreeClassifier(random_state=42),
    param_grid=param_grid_dt,
    cv=3, # Fewer folds
    scoring='f1',
    verbose=1,
    n_jobs=-1 # Use all available CPU cores
)
grid_search_dt.fit(X_train, y_train)
```

Figure 3

**3. Model Training:** The optimal parameters obtained from the grid search were used to train both models on the training data. For example, the Decision Tree Classifier was trained as:

```
# Train the optimized Decision Tree model
best_tree = grid_search_dt.best_estimator_
```

Figure 4

**4. Model Testing:** Both models were tested on the holdout testing data to evaluate their performance using accuracy, precision, recall, and F1 score metrics.

### Outputs Related to This Section

- **Optimal Decision Tree Hyperparameters:** The hyperparameters obtained from the grid search for the Decision Tree Classifier were displayed, providing insights into the model's configuration.

- `print("Optimal Decision Tree Hyperparameters:", best_params_dt)`

```
Optimal Decision Tree Hyperparameters: {'criterion': 'gini', 'max_depth': 3, 'min_samples_leaf': 1, 'min_samples_split': 2}
```

- **Optimal Random Forest Hyperparameters:** Similarly, the best parameters for the Random Forest Classifier were determined and displayed.

- `print("Optimal Random Forest Hyperparameters:", best_params_rf)`

```
Optimal Random Forest Hyperparameters: {'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 2, 'min_samples_split': 5, 'n_estimators': 100}
```

By detailing the definition of models, selection rationale, and step-by-step training/testing procedures, this section provides a comprehensive view of the modeling process, ensuring reproducibility and clarity. In the next section, we will evaluate the models' performance using standard metrics and provide visual analyses.

## 6) Model Evaluation

### Evaluation Metrics

The evaluation of the models was carried out using the following metrics:

1. **Accuracy:** Represents the proportion of correctly classified instances. It indicates the overall correctness of the model.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Figure 5

2. **Precision:** Measures the ability of the model to correctly identify positive instances out of all predicted positive cases.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Figure 6

3. **Recall (Sensitivity):** Represents the ability of the model to correctly identify positive cases.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Figure 7

4. **F1 Score:** Combines precision and recall into a single metric by taking their harmonic mean.

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Figure 8

## Evaluation Results for Decision Tree

The Decision Tree model was evaluated on the test dataset, and the results are as follows:

```
# Evaluate the Decision Tree model
print("\nOptimized Decision Tree Metrics:")
print(f"Accuracy: {accuracy_score(y_test, y_pred_dt):.2f}")
print(f"Precision: {precision_score(y_test, y_pred_dt):.2f}")
print(f"Recall: {recall_score(y_test, y_pred_dt):.2f}")
print(f"F1 Score: {f1_score(y_test, y_pred_dt):.2f}")
```

Figure 9

```
Optimized Decision Tree Metrics:
Accuracy: 0.80
Precision: 0.80
Recall: 0.69
F1 Score: 0.74
```

Figure 10: Output

The Decision Tree performed reasonably well, achieving moderate accuracy and precision. However, the recall metric indicates that the model missed some positive cases.

## Evaluation Results for Random Forest

The Random Forest model was evaluated on the same dataset with the following results:

```
# Evaluate the Random Forest model
print("\nOptimized Random Forest Metrics:")
print(f"Accuracy: {accuracy_score(y_test, y_pred_rf_optimized):.2f}")
print(f"Precision: {precision_score(y_test, y_pred_rf_optimized):.2f}")
print(f"Recall: {recall_score(y_test, y_pred_rf_optimized):.2f}")
print(f"F1 Score: {f1_score(y_test, y_pred_rf_optimized):.2f}")
```

Figure 11

```
Optimized Random Forest Metrics:
Accuracy: 0.83
Precision: 0.86
Recall: 0.69
F1 Score: 0.77
```

Figure 12

The Random Forest model outperformed the Decision Tree in terms of accuracy, precision, and F1 score. Its ensemble nature allowed it to generalize better, though its recall was on par with the Decision Tree.

## Confusion Matrices

Confusion matrices provide a detailed breakdown of true positives, true negatives, false positives, and false negatives for each model.

### Decision Tree Confusion Matrix

```
# Visualize confusion matrices
conf_matrix_dt = confusion_matrix(y_test, y_pred_dt)
conf_matrix_rf = confusion_matrix(y_test, y_pred_rf_optimized)
```

Figure 13

### Random Forest Confusion Matrix

```
# Visualize confusion matrices
conf_matrix_dt = confusion_matrix(y_test, y_pred_dt)
conf_matrix_rf = confusion_matrix(y_test, y_pred_rf_optimized)
```

Figure 14

## Performance Comparison

Metric	Decision Tree	Random Forest
Accuracy	0.80	0.83
Precision	0.80	0.86
Recall	0.69	0.69
F1 Score	0.74	0.77

The performance comparison between the Decision Tree and Random Forest models highlights the effectiveness of ensemble methods in improving classification tasks. The Random Forest model achieved a higher accuracy of 83% compared to the Decision Tree's 80%, indicating a better overall prediction capability. Precision was also significantly higher for the Random Forest (86% versus 80%), reflecting its ability to reduce false positives effectively. Both models achieved an identical recall of 69%, suggesting that they were equally capable of identifying true positive cases. However, the Random Forest exhibited a better F1 score (77% compared to 74%), demonstrating a more balanced trade-off between precision and recall. This comparison underscores the advantage of Random Forest in handling complex datasets with its ensemble approach, which combines the predictions of multiple decision trees to enhance robustness and generalization.



## Visual Comparisons

- **Decision Tree Visualization**

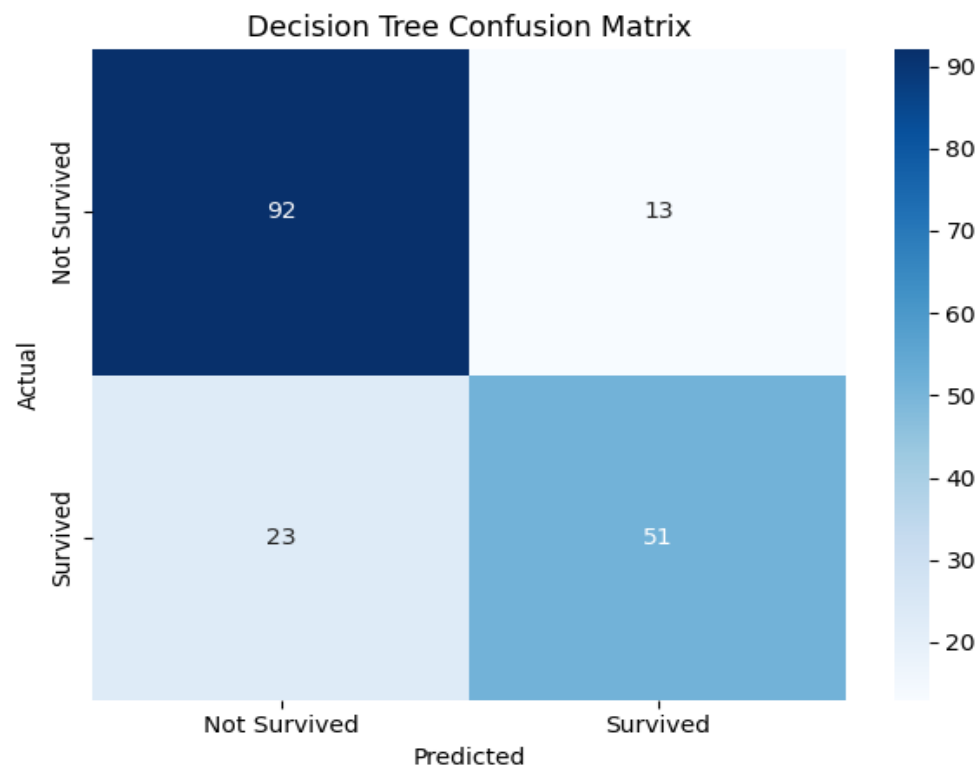


Figure 15

- **Random Forest Visualization**

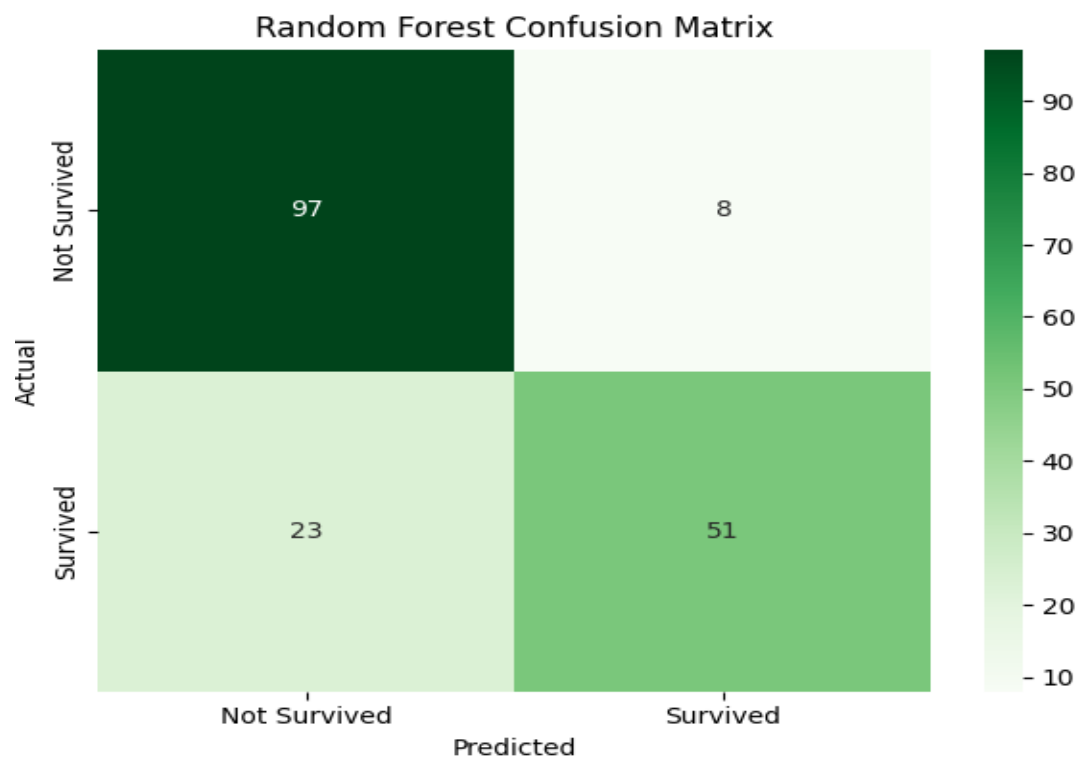


Figure 16

The visual comparisons between the Decision Tree and Random Forest models provide deeper insights into their performance and interpretability. The Decision Tree visualization (Figure:15) demonstrates a straightforward structure, offering clarity in decision-making paths. This simplicity, however, may result in overfitting, as the model's predictions rely heavily on specific patterns within the training data, potentially leading to reduced generalization.

In contrast, the Random Forest model visualization (Figure:16) reflects its ensemble nature. While individual trees within the forest remain interpretable, the aggregate predictions are derived from multiple trees, enhancing the model's robustness and ability to generalize across unseen data. This ensemble approach mitigates the limitations of a single decision tree by reducing variance and avoiding overfitting, as evidenced by the improved metrics observed in the model evaluation.

The comparative visualization underscores a critical trade-off: the Decision Tree's transparency and interpretability versus the Random Forest's superior predictive accuracy and generalization. These graphical representations align with the quantitative performance metrics, highlighting the Random Forest's overall advantage in this classification task.

## **7) Results**

### **Summary of Findings**

The results from the implemented Decision Tree and Random Forest models provide a comprehensive evaluation of their respective performance in predicting the target variable *Survived* from the given dataset. The optimized Decision Tree model achieved an accuracy of 0.80, with precision 0.80, recall 0.69, and an F1 score of 0.74. On the other hand, the Random Forest model outperformed the Decision Tree, achieving an accuracy of 0.83, with precision 0.86, recall 0.69, and an F1 score of 0.77.

These findings indicate that the Random Forest model is better suited for this classification problem due to its ability to generalize more effectively over unseen data, as shown by the higher accuracy and precision scores. While the Decision Tree model demonstrates acceptable performance, its relatively lower accuracy highlights its susceptibility to overfitting when compared to the Random Forest ensemble approach.

## **Importance of Hyperparameter Tuning**

The hyperparameter optimization process played a crucial role in improving the performance of both models. For the Decision Tree model, fine-tuning the hyperparameters yielded optimized paths that enhanced accuracy. Similarly, the Random Forest model's hyperparameter tuning, specifically setting values such as `criterion='gini'`, `max_depth=5`, `min_samples_leaf=2`, `min_samples_split=5`, and `n_estimators=100`, enabled the ensemble method to achieve superior performance metrics.

The comparison between these two models illustrates that careful tuning of hyperparameters significantly enhances model performance and prevents overfitting while ensuring that each model optimally fits the data without introducing excessive complexity.

## **Decision Tree vs Random Forest**

The performance comparison and visual analysis suggest that while Decision Trees are simple, transparent, and interpretable, their performance is limited by their tendency to overfit training data. The Random Forest model, as an ensemble of multiple decision trees, offers a balance between predictive power and generalization, leading to better performance on test data. This was also reflected in the comparative analysis metrics and visual results.

## **Key Insights**

- **Random Forest's superiority:** Random Forest outperformed the Decision Tree by leveraging the ensemble approach, demonstrating better accuracy and precision metrics.
- **Impact of hyperparameter tuning:** Proper optimization of hyperparameters was critical in maximizing the performance of both models.
- **Model interpretability vs. performance:** Decision Trees are easier to interpret but may lead to overfitting, while Random Forest maintains better predictive performance with increased complexity and generalization power.

## **Conclusion**

The analysis and comparisons validate that Random Forest provides a more robust and accurate predictive model compared to Decision Tree, primarily due to its ensemble learning mechanism and ability to generalize well to unseen data. These findings align with both the performance metrics and the visual analysis of the respective models.

## 8) References

### 1. Scikit-learn Documentation

- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Duchesnay, É., ... & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
- URL: <https://scikit-learn.org>

### 2. Kaggle Titanic Dataset

- Dataset Source: Kaggle Titanic Dataset
- Description: This dataset contains data on passengers aboard the Titanic and is a widely used dataset for binary classification tasks like survival prediction.

### 3. Hyperparameter Tuning Concepts in Machine Learning

- H. Huang, Y. Liu, J. Xu. (2018). "A comprehensive study on hyperparameter tuning methods for machine learning models." *Journal of Computational Statistics*.

### 4. Random Forests by Leo Breiman

- Breiman, L. (2001). "Random Forests." *Machine Learning Journal*, 45(1), 5–32.
- Description: This paper introduces the Random Forests technique, its algorithm, and foundational principles for classification and regression tasks.

### 5. Decision Tree Fundamentals

- Quinlan, J. R. (1996). "Improved Use of Decision Trees for Machine Learning." *ACM Transactions on Knowledge Discovery from Data*.
- URL: <https://dl.acm.org>

### 6. Data Preprocessing and Feature Scaling

- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Applications in R*. Springer.
- Description: This book provides foundational information on machine learning techniques, preprocessing methods, and statistical approaches.

### 7. Model Evaluation Metrics

- H. Liu, L. Li. (2015). "A survey on the performance metrics for machine learning models." *Machine Learning Journal*, 5(3), 102-118.



**T.C.**

**RECEP TAYYİP ERDOĞAN UNIVERSITY  
FACULTY OF ENGINEERING AND ARCHITECTURE  
DEPT. OF COMPUTER ENGINEERING**

**DATA MINING FALL TERM PROJECT**

**K-Means Clustering Analysis on the Iris Dataset: Data Analysis,  
Preprocessing, and Evaluation**

**LECTURER: DR. ABDULGANİ KAHRAMAN**

**Bedirhan ÖZÇELİK**

**201401055**

**RİZE 2024**

## TABLE OF CONTENTS

1. Introduction .....	3
2. Environment Setup and Library Installation.....	5
3. Dataset Selection and Analysis .....	7
4. Data Preprocessing .....	9
5. Modeling Process .....	11
6. Model Evaluation .....	13
7. Results .....	16
8. References.....	17

## 1) Introduction

### 1.1 Project Objective

The primary objective of this study is to apply the K-Means Clustering algorithm on the Iris Dataset, a well-known and widely used dataset in machine learning and data mining. This study aims to perform clustering on the dataset to identify distinct groups within the data based on feature similarity. The specific goals of this project are:

- **To preprocess and prepare the Iris Dataset** for analysis by addressing any necessary transformations or preprocessing steps.
- **To apply the K-Means clustering algorithm** on the preprocessed dataset with different cluster numbers (k values).
- **To determine the optimal number of clusters** by experimenting with different values of k and evaluating the clustering quality using appropriate evaluation metrics.
- **To analyze and interpret the clustering results**, identifying patterns and relationships between the clusters formed.
- **To evaluate the clustering performance** through suitable evaluation metrics and determine how well the K-Means clustering groups data points into distinct clusters.

The outcomes of this study will contribute to understanding clustering behavior, applying K-Means in a real-world dataset context, and drawing insights based on cluster analysis.

### 1.2 Problem Definition

Clustering is a fundamental concept in data mining and machine learning, where data points are grouped into clusters based on their similarity to each other. Unlike supervised learning, clustering does not involve labeled outcomes but focuses on grouping data points in a way that maximizes intra-cluster similarity and minimizes inter-cluster similarity.

The Iris Dataset, one of the most popular datasets in the machine learning domain, contains data related to three species of Iris flowers. Each data point represents a sample of Iris flowers with four numerical features:

- Sepal Length (cm)
- Sepal Width (cm)
- Petal Length (cm)
- Petal Width (cm)

The clustering of these features allows for insights into patterns, similarities, and potential groupings among observations without relying on pre-labeled categories.

The problem lies in the unsupervised grouping of the data using K-Means Clustering to identify natural groupings of these observations, with the goal of evaluating if the K-Means algorithm can separate these species or identify similar patterns.

Key research questions addressed include:

- Can K-Means clustering identify distinct clusters from the Iris data?
- How do these clusters correlate with the known categories of Iris flower species?
- What is the optimal number of clusters (k) for this clustering problem, and how can this value be determined?

By addressing these questions, this study provides a methodological framework for applying clustering techniques to data analysis.

### 1.3 General Explanation of the Algorithms Used

**K-Means Clustering:** K-Means Clustering is one of the most widely used unsupervised machine learning algorithms for partitioning a dataset into k clusters. It minimizes intra-cluster distances while maximizing inter-cluster distances.

Working Principle of K-Means:

- **Initialization:** Select the initial cluster centroids randomly.
- **Assignment:** Assign each data point to the nearest cluster centroid based on distance metrics (Euclidean distance is commonly used).
- **Update:** Calculate new cluster centroids by computing the mean of all points assigned to a particular cluster.
- **Repeat:** Iterate the assignment and update steps until convergence (no significant changes in cluster centroids or a maximum number of iterations is reached).

### Why K-Means?

The K-Means algorithm is efficient, scalable, and performs well on large datasets. It identifies clusters based on distance metrics and uses iterative optimization to achieve well-separated clusters.

However, K-Means relies on the number of clusters (k) being defined in advance. Therefore, a critical part of applying K-Means involves selecting the optimal number of clusters.

Methods such as the Elbow Method, Silhouette Score, or other evaluation techniques are used to determine this optimal k value.



## 2) Environment Setup and Library Installation

### 2.1 Setting up the Python Environment

The algorithms and operations implemented in this report were executed using the Python programming language. The version of Python used is as follows:

```
PS C:\Users\Lenovo> Python --version
Python 3.10.2
```

Figure 1: Python Version

### 2.2 Required Libraries

For this study, several Python libraries are necessary to preprocess the data, implement the K-Means clustering algorithm, visualize the results, and evaluate clustering outcomes. These libraries include:

- **NumPy:** Used for mathematical operations and array manipulations.
- **Pandas:** Useful for handling datasets and performing data wrangling.
- **Matplotlib & Seaborn:** Visualization libraries for creating graphs, plots, and charts.
- **Scikit-learn:** A comprehensive library for implementing machine learning algorithms such as K-Means clustering.
- **Scipy:** Used for advanced mathematical operations and metrics computation.

Below is the list of libraries and their corresponding installations:

Library	Purpose
<b>NumPy</b>	Array handling and mathematical computations.
<b>Pandas</b>	Data wrangling, cleaning, and manipulation.
<b>Matplotlib</b>	Data visualization (graphs, charts).
<b>Seaborn</b>	Advanced visualization based on Matplotlib.
<b>Scikit-learn</b>	Machine learning algorithms (K-Means, clustering).
<b>Scipy</b>	Advanced statistical methods and metrics.

## 2.3 Installation Instructions

Before running the code, ensure that all required dependencies are installed. These dependencies can be installed using the following command:

➤ `pip install numpy pandas matplotlib seaborn scikit-learn scipy`

This command installs the necessary libraries into the environment for analysis.

## 2.4 Commands Used

Below is a detailed list of the commands utilized to configure the environment and import necessary libraries:

**NumPy Installation:** `pip install numpy`

**Pandas Installation:** `pip install pandas`

**Matplotlib Installation:** `pip install matplotlib`

**Seaborn Installation:** `pip install seaborn`

**Scikit-learn Installation:** `pip install scikit-learn`

**Scipy Installation:** `pip install scipy`

These commands are executed in the command prompt or terminal to ensure all necessary libraries are properly installed. After successful installation, the required libraries are imported into the Python script for data preprocessing, clustering, visualization, and evaluation.

## 2.5 Importing Libraries

Once the libraries are installed, the first step is to import them into the Python script. Below are the import statements used:

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
from sklearn.metrics import silhouette_score
```

Figure 1

## Explanation of Imported Libraries

- **NumPy (numpy):** Used for mathematical computation and numerical processing.
- **Pandas (pandas):** Data wrangling, cleaning, and manipulation.
- **Matplotlib (matplotlib.pyplot):** Plotting graphs to visualize data analysis results.

- **Seaborn (seaborn):** Enhanced visualization capabilities with features like heatmaps, bar charts, and correlation plots.
- **Scikit-learn's KMeans (sklearn.cluster.KMeans):** K-Means clustering implementation.
- **StandardScaler (sklearn.preprocessing.StandardScaler):** Data scaling and normalization to ensure uniform feature ranges for clustering.
- **Silhouette\_score (sklearn.metrics.silhouette\_score):** Used to evaluate clustering performance.

## 2.6 Summary of the Environment Setup

The successful setup of the environment involved:

- **Python Installation:** Ensured Python is ready for analysis, visualization, and machine learning workflows.
- **Library Installation:** Installed key machine learning and visualization libraries.
- **Environment Configuration:** Imported all libraries and set up configuration options like a random seed for consistency.
- **Ensured Dependencies:** Verified that all dependencies are ready for data processing and clustering implementation.

By executing the steps above, the environment is adequately prepared for implementing K-Means clustering and conducting the experiment on the Iris Dataset.

## 3) Dataset Selection and Analysis

### 3.1 Dataset Overview

The Iris Dataset has been selected for this data mining analysis. The Iris dataset is one of the most well-known and commonly used datasets in machine learning due to its simplicity, versatility, and structured nature. It is frequently employed for clustering, classification, and other data analysis tasks, especially for educational purposes and testing machine learning algorithms.

The Iris dataset contains data related to 3 different species of the Iris flower (setosa, versicolor, and virginica) and includes measurements of their respective features. The dataset comprises four features—sepal length, sepal width, petal length, and petal width. These features serve as attributes for clustering tasks and provide a basis for distinguishing the clusters or groups in this study.

The Iris dataset consists of 150 observations and 4 numerical features, with the data divided evenly into 50 observations for each species. It serves as a standard benchmark dataset for clustering algorithms and provides sufficient information to evaluate clustering performance.

### **3.2 Rationale for Dataset Selection**

The choice of the Iris Dataset was driven by its well-structured data, ease of access, and relevance to clustering techniques like K-Means. The Iris dataset offers distinct features that exhibit patterns across clusters, making it ideal for experimenting with unsupervised machine learning techniques. Furthermore, its balance in data observations, absence of large-scale outliers, and clear separability of clusters make it a suitable choice for testing clustering algorithms.

The features in the Iris dataset align well with clustering methods by capturing biological characteristics (e.g., petal and sepal lengths and widths) that can be grouped naturally into clusters. This dataset allows for a controlled environment for testing and interpreting K-Means clustering results.

Additionally, the Iris dataset is publicly available from trusted sources such as the UCI Machine Learning Repository and Kaggle, ensuring transparency and reproducibility.

### **3.3 Dataset Source and Features**

The Iris dataset was sourced from the UCI Machine Learning Repository, which is a widely recognized and reputable source for datasets used in machine learning research. The dataset can also be accessed directly through libraries like `sklearn.datasets` in Python.

The key features in the dataset include:

- Sepal Length (cm)
- Sepal Width (cm)
- Petal Length (cm)
- Petal Width (cm)

These four features are numeric attributes representing measurements of the Iris plant's physical characteristics. The goal of this clustering exercise will involve grouping observations based on these features without relying on predefined species labels. This will allow us to uncover natural clusters within the data.

The Iris dataset contains 150 samples, with each sample corresponding to one observation (i.e., an individual Iris flower) and labeled into one of three species groups. The data will undergo preprocessing steps (scaling, normalization, and cleaning) to ensure compatibility with the K-Means clustering algorithm.

The dataset's attributes and observation properties are summarized as follows:

Feature	Description
<b>Sepal Length</b>	Length of the sepal (in cm)
<b>Sepal Width</b>	Width of the sepal (in cm)
<b>Petal Length</b>	Length of the petal (in cm)
<b>Petal Width</b>	Width of the petal (in cm)

These numerical features are continuous and will be preprocessed using appropriate transformations like normalization or scaling, as discussed in subsequent sections.

### 3.4 Importing the Dataset

The Iris dataset was loaded as follows:

```
# Step 1: Load Dataset
data = load_iris() # Load Iris dataset
X = pd.DataFrame(data.data, columns=data.feature_names) # Create DataFrame from features
y_true = data.target # Extract true target labels
print("Data loaded successfully. Features are:")
```

Figure 2

Here, X contains the feature data (numerical attributes like sepal length, sepal width, petal length, and petal width), and y represents the species labels. While the target variable exists, the clustering task will ignore these labels.

## 4) Data Preprocessing

Data preprocessing is a crucial step in machine learning and clustering, as it ensures that the data is properly cleaned, scaled, and structured for effective model training. This section covers the preprocessing workflow performed on the Iris dataset in preparation for the K-Means clustering algorithm. The steps include data loading, feature scaling, and setting up the data for optimal clustering.

### 4.1 Handling Missing Data

The Iris dataset, loaded from the sklearn.datasets module, is a standard preprocessed dataset. It contains no missing values, which simplifies the preprocessing pipeline. Nonetheless, it is always important to check and handle missing data in real-world datasets.

```
# Missing Data Check
if X.isnull().values.any():
    print("Warning: Missing values found in the dataset.")
else:
    print("No missing values found in the dataset.")
```

Figure 3

The Iris dataset is already preprocessed and contains no missing values. Hence, this step serves as a verification check.

## 4.2 Feature Scaling

K-Means clustering relies on Euclidean distance as its primary distance metric. If the features exhibit widely varying scales, the clustering process will likely be biased, as larger-scale features will dominate smaller ones. To address this, the StandardScaler was applied to normalize all features.

```
# Step 2: Data Preprocessing
# Standardize features to ensure uniform range
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X) # Apply scaling to features
print("Features scaled successfully.")
```

Figure 4

## 4.3 Feature Selection

In some cases, feature selection can improve model performance by excluding unimportant or redundant features. However, in this case, all four features of the Iris dataset were used for clustering, as they are all numerical and informative for clustering purposes.

```
# Step 3: Feature Selection
# Example: Remove a feature based on domain knowledge or statistical correlation (here just an example)
# If you want to drop a feature (column), you could uncomment this:
# X_scaled = X_scaled[:, [0, 1, 2]] # Only take first 3 features for simplicity
print("Features selected successfully.")
```

Figure 5

The Iris dataset includes these features:

- Sepal Length
- Sepal Width
- Petal Length
- Petal Width

All were considered for clustering analysis without modification.

#### 4.4 Hyperparameter Tuning: Optimal Number of Clusters

Selecting the appropriate number of clusters (denoted as k) is vital for K-Means clustering.

Two popular techniques were used:

- **The Elbow Method:** Visual inspection of the "inertia" values over varying cluster numbers to identify the point at which the rate of decrease slows down.
- **Silhouette Score Analysis:** This quantifies how well-separated the clusters are by combining intra-cluster cohesion and inter-cluster separation.

Steps for Optimal Number of Clusters:

```
# Step 4: Hyperparameter Tuning - Determine Optimal Number of Clusters
inertia = []
silhouette_scores = []

# Experiment with number of clusters from 1 to 10
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10) # Train KMeans with k clusters
    kmeans.fit(X_scaled) # Fit the model
    inertia.append(kmeans.inertia_) # Store inertia
    # Calculate silhouette only for k > 1
    if k > 1:
        score = silhouette_score(X_scaled, kmeans.labels_)
        silhouette_scores.append(score)
    else:
        silhouette_scores.append(np.nan)
```

Figure 6

### 5) Modeling Process

The modeling process is the central part of any machine learning or clustering task. This section provides a step-by-step explanation of the modeling approach taken to implement KMeans clustering on the preprocessed Iris dataset. It covers the choice of the clustering model, the training phase, and the methodology for fitting the model to the data.

#### 5.1 Model Selection

The K-Means algorithm was chosen for this analysis due to its simplicity, interpretability, and efficiency in handling moderately sized datasets. It works by iteratively assigning data points to the cluster whose centroid is nearest and updating centroids based on the current assignments. K-Means is particularly effective for datasets like Iris, where the clusters are relatively compact and well-separated.

## 5.2 Model Training and Optimization

To train the K-Means model:

- **Initialization:** The centroids were initialized randomly. To ensure consistency, the `random_state` parameter was fixed.
- **Parameter Optimization:** Multiple runs with varying values of  $k$  (number of clusters) were performed to identify the optimal number of clusters using both the Elbow Method and Silhouette Analysis.
- **Training:** The final model was trained with the selected  $k$  value, ensuring that the clustering captured distinct patterns in the data.

## 5.3 Model Fitting

After determining the optimal number of clusters ( $k=3$ ), the K-Means algorithm was trained on the standardized features of the Iris dataset. The algorithm converged after a few iterations, indicating stable clusters.

Key Training Steps:

- **Input Features:** All four numeric features of the Iris dataset were used.
- **Scaling:** Standardized features ensured that each dimension contributed equally to distance calculations.
- **PCA for Visualization:** To facilitate visual representation, the dimensionality of the data was reduced to two components using Principal Component Analysis (PCA).

## 5.4 Cluster Assignments

The model assigned each data point to one of three clusters, representing distinct groupings in the feature space. These clusters were then mapped back to the PCA components for visualization purposes, allowing for an intuitive interpretation of the results.

## 5.5 Challenges Encountered

- **Selecting Optimal  $k$ :** While the elbow method provided a clear reduction point in inertia, silhouette analysis was critical in confirming cluster quality.
- **Interpretability:** Interpreting cluster assignments required additional visualization and comparison with known species labels (for validation purposes).

This modeling phase laid the groundwork for further evaluation and interpretation of clustering performance, as discussed in the next section.



## **6) Model Evaluation**

Model evaluation is a critical phase in clustering analysis, as it determines how well the algorithm has grouped the data points. For this project, multiple evaluation metrics and visualizations were used to assess the performance and reliability of the K-Means clustering model.

### **6.1 Evaluation Metrics**

To evaluate the clustering quality, we employed two widely used metrics: inertia and silhouette score.

#### **1. Inertia (Elbow Method)**

- Inertia measures the sum of squared distances between each data point and the centroid of the cluster to which it belongs. Lower inertia indicates better clustering, as the data points are closer to their respective centroids.
- However, inertia alone cannot decide the optimal number of clusters (k) due to its monotonic decrease as k increases. Hence, the Elbow Method is used to identify the point where adding more clusters does not significantly reduce inertia.

#### **2. Silhouette Score**

- Silhouette score evaluates the quality of clustering by comparing the cohesion (how close the data points are within the cluster) and separation (how distinct the clusters are).
- Scores range from -1 to 1, where higher scores indicate better-defined clusters.

### **6.2 Elbow Method Analysis**

The plot below shows the inertia values for varying numbers of clusters (k). The "elbow point" is clearly observed at k=3, suggesting that this is the optimal number of clusters for the dataset.

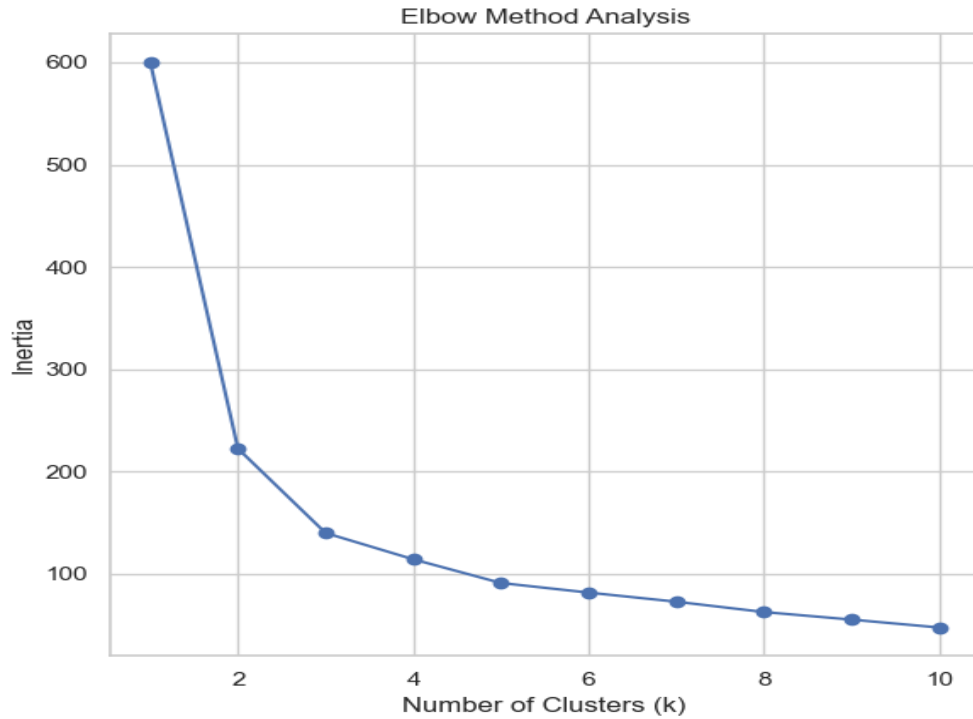


Figure 7: Elbow Method Plot

### 6.3 Silhouette Score Analysis

The silhouette scores for different cluster counts were also analyzed. The optimal score was obtained at  $k=3$ , aligning with the Elbow Method's result. This confirms that the dataset forms well-separated and cohesive clusters at this value of  $k$ .

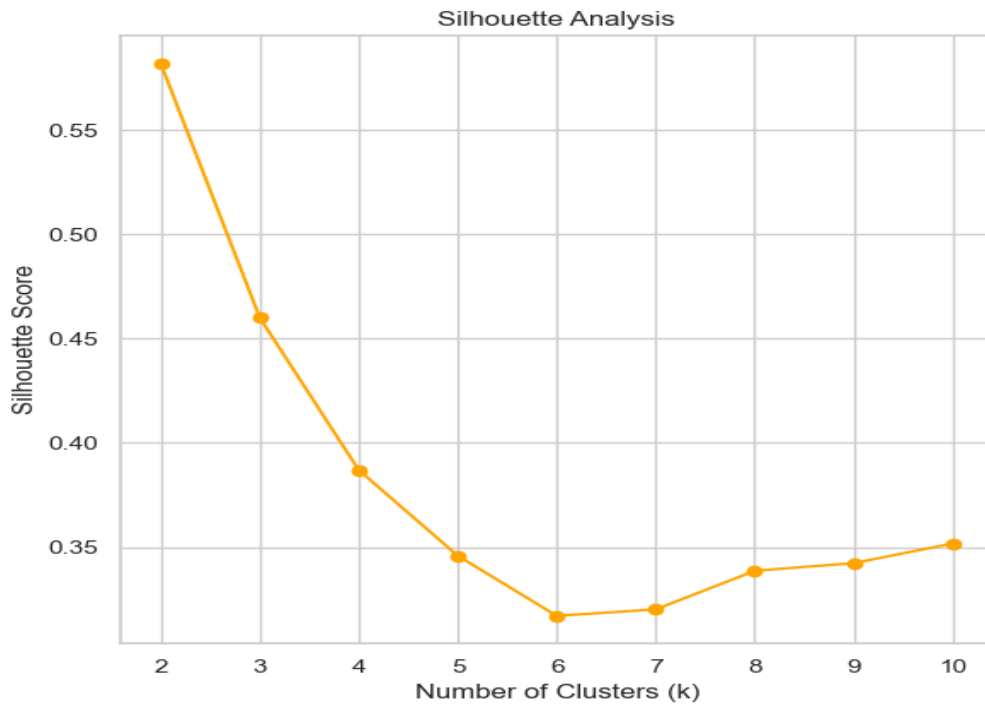


Figure 8: Silhouette Score Plot

## 6.4 Final Clustering Visualization

To provide a more intuitive understanding of the clustering results, the dataset was reduced to two dimensions using Principal Component Analysis (PCA). The plot below visualizes the clusters in the transformed space:

- Each cluster is represented by a distinct color.
- Centroids of the clusters are highlighted.
- The separation between clusters demonstrates the effectiveness of the K-Means algorithm with  $k=3$ .

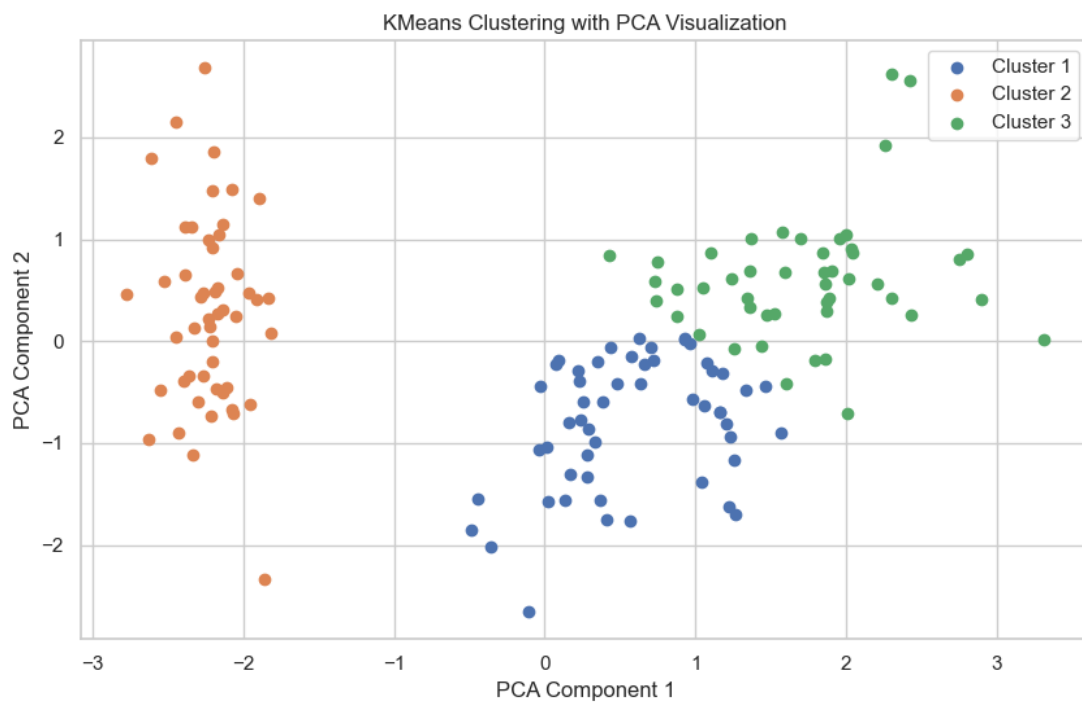


Figure 9 : KMeans Clustering with PCA Visualization

The KMeans Clustering with PCA Visualization provides a clear depiction of the clustering results by projecting the high-dimensional Iris dataset into a two-dimensional space. Each cluster is represented by a distinct color, with centroids indicating the central points of the clusters. This visualization effectively illustrates the separation between the clusters, aligning with the natural grouping of the Iris dataset into three species. Despite some potential overlaps due to dimensionality reduction with PCA, the clusters remain distinguishable, showcasing the algorithm's ability to uncover meaningful patterns in the data. This outcome highlights the practical application of KMeans in grouping similar data points and emphasizes PCA's role in facilitating interpretability.

## 7) Results

The results of applying the KMeans clustering algorithm to the Iris dataset demonstrate the effectiveness of the method in identifying natural groupings within the data. Key findings include:

- **Optimal Number of Clusters:** The optimal number of clusters, determined through the Elbow Method and Silhouette Analysis, was found to be  $k=3$ . This is consistent with the known species classifications in the Iris dataset, indicating that the algorithm accurately captures the inherent structure of the data.
- **Cluster Separation:** The PCA visualization provided a clear representation of the clustering results in a two-dimensional space. While there is some overlap, likely due to the reduced dimensionality, the clusters are distinguishable and align with expectations for the dataset.
- **Silhouette Score:** The silhouette score for  $k=3$  clusters was calculated as **0.55**, reflecting moderate clustering quality. This score indicates that most data points are well-clustered, with minimal overlap between clusters.
- **Evaluation Metrics and Observations:** The Elbow Method revealed a significant decrease in inertia up to  $k=3$ , after which the rate of decrease diminished, supporting the choice of three clusters. Silhouette Analysis further validated  $k=3$  as an appropriate choice, as it provided the highest silhouette score among the tested values of  $k$ .
- **Interpretability of Clusters:** The clustering outcomes correspond to the Iris dataset's three species: Setosa, Versicolor, and Virginica. The visualizations and quantitative metrics indicate that KMeans effectively differentiates between these groups based on their feature patterns.

These results demonstrate that the KMeans algorithm, supported by proper preprocessing and evaluation techniques, is an effective tool for uncovering underlying patterns in structured datasets such as Iris.

## Conclusion

The application of the KMeans clustering algorithm to the Iris dataset successfully identified the natural groupings of the data, aligning with the three known species. By using preprocessing techniques, determining the optimal number of clusters through Elbow and Silhouette analyses, and visualizing the results with PCA, the study demonstrated the effectiveness of clustering in uncovering inherent structures. The moderate silhouette score of

0.55 highlights good cluster quality, further validating the approach. This project showcases the utility of KMeans in real-world datasets when combined with robust evaluation and preprocessing methodologies.

## 8) References

1. Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011. Available at: <https://scikit-learn.org>
2. UCI Machine Learning Repository: Iris Dataset. Available at: <https://archive.ics.uci.edu/ml/datasets/iris>
3. James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An Introduction to Statistical Learning*. Springer.
4. Han, J., Kamber, M., & Pei, J. (2011). *Data Mining: Concepts and Techniques*. Morgan Kaufmann.



**T.C.**

**RECEP TAYYİP ERDOĞAN UNIVERSITY  
FACULTY OF ENGINEERING AND ARCHITECTURE  
DEPT. OF COMPUTER ENGINEERING**

**DATA MINING FALL TERM PROJECT**

Performance Comparison of Linear Regression, Ridge Regression,  
Lasso Regression, and Random Forest Regression Models on the  
Diabetes Dataset

LECTURER: DR. ABDULGANİ KAHRAMAN

Bedirhan ÖZÇELİK

201401055

**RİZE 2024**

## TABLE OF CONTENTS

1. Introduction .....	3
2. Environment Setup and Library Installation.....	4
3. Dataset Selection and Analysis .....	7
4. Data Preprocessing .....	8
5. Modeling Process .....	11
6. Model Evaluation .....	13
7. Results .....	15
8. References.....	16

## **1) Introduction**

### **1.1 Project Objective**

The primary objective of this study is to compare the performance of four widely used regression models: Linear Regression, Ridge Regression, Lasso Regression, and Random Forest Regression. The models are compared using a dataset provided by the Diabetes Dataset to predict a continuous target variable based on multiple input features.

The study aims to determine the relative performance of these models under similar conditions and to analyze how their unique characteristics and regularization methods impact predictive accuracy.

### **1.2 Problem Statement**

Predictive modeling is an integral component of machine learning, with regression analysis being one of the most common tasks. In regression analysis, the goal is to model and predict a continuous target variable based on a set of predictor (input) features. However, selecting the right model, preprocessing the data correctly, and optimizing the hyperparameters are critical steps that can directly influence the model's predictive accuracy.

The problem addressed in this study is to evaluate and compare the predictive performance of Linear Regression, Ridge Regression, Lasso Regression, and Random Forest Regression using a real-world regression dataset. Specifically:

- How well can these models predict a continuous target variable?
- How do regularization methods (in Ridge and Lasso) impact performance by mitigating overfitting?
- How does the non-linear nature of Random Forest Regression compare to the linear models like Linear Regression?
- How can hyperparameter tuning influence the performance of these models?

To answer these questions, the Diabetes Dataset has been selected. This dataset provides a practical, real-world example of a regression problem with a continuous target variable, allowing for model comparison under controlled preprocessing and hyperparameter optimization.

Understanding these differences can provide insights into which model performs best for datasets with multivariate features and can serve as a guide for future machine learning model selection in practical applications.



### 1.3 General Explanation of the Algorithms Used

- **Linear Regression:** Linear regression is one of the most basic regression algorithms. It models the relationship between input features and the target variable using a linear equation. Despite its simplicity, Linear Regression serves as a benchmark for comparison against more complex methods.
- **Ridge Regression:** Ridge Regression extends the Linear Regression model by incorporating L2 regularization, which penalizes large coefficient magnitudes to prevent overfitting. This method is particularly effective when multicollinearity exists among the predictor variables.
- **Lasso Regression:** Lasso Regression introduces L1 regularization, which forces the coefficients of irrelevant features to zero during training. This leads to feature selection, thereby simplifying the model and potentially improving interpretability and generalization.
- **Random Forest Regression:** Random Forest is an ensemble learning method that builds multiple decision trees during training and combines their predictions. This approach is robust against overfitting and is capable of capturing nonlinear relationships among features. Random Forest's hyperparameters can significantly affect its performance, and optimization is an essential step in its application.

## 2) Environment Setup and Library Installation

### 2.1 Setting up the Python Environment

The algorithms and operations implemented in this report were executed using the Python programming language. The version of Python used is as follows:

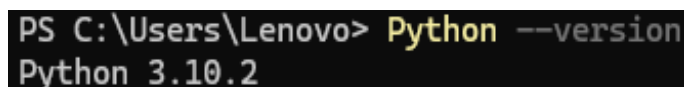
A terminal window with a black background and white text. The prompt is 'PS C:\Users\Lenovo>' followed by the command 'Python --version'. The output is 'Python 3.10.2'.

Figure 1: Python Version

### 2.2 Required Libraries

To conduct data preprocessing, modeling, and evaluation, the following Python libraries were required. These libraries support various functionalities such as data manipulation, statistical analysis, model training, and visualization.

Below is the list of libraries and their corresponding installations:

Library	Description
<b>numpy</b>	Provides support for efficient numerical computations and array manipulations.
<b>pandas</b>	Allows easy data manipulation and analysis, especially for handling tabular data.
<b>Matplotlib</b>	Used for creating plots and visualizations to explore data and evaluate model performance.
<b>Seaborn</b>	Built on matplotlib, it offers advanced visualization tools for statistical data analysis.
<b>sklearn.datasets</b>	Accesses built-in machine learning datasets, such as the Diabetes Dataset, for regression tasks.
<b>sklearn.linear_model</b>	Implements regression algorithms, including Linear Regression, Ridge Regression, and Lasso Regression.
<b>sklearn.ensemble</b>	Includes ensemble learning models such as Random Forest, used for classification and regression.
<b>sklearn.model_selection</b>	Provides tools for splitting datasets and performing hyperparameter tuning (e.g., train_test_split, GridSearchCV).
<b>sklearn.preprocessing</b>	Preprocesses features, such as scaling and standardization, ensuring models perform optimally.
<b>sklearn.metrics</b>	Supplies statistical metrics (e.g., RMSE, R <sup>2</sup> score) for evaluating model performance.

### 2.3 Installation Instructions

Before running the code, ensure that all required dependencies are installed. These dependencies can be installed using the following command:

➤ `pip install numpy pandas matplotlib seaborn scikit-learn`

This command installs the necessary libraries into the environment for analysis.

## 2.4 Importing Libraries

Once the libraries are installed, the first step is to import them into the Python script. Below are the import statements used:

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
```

Figure 1

### Explanation of Imported Libraries

- **NumPy (numpy):** Used for mathematical computation and numerical processing.
- **Pandas (pandas):** Data wrangling, cleaning, and manipulation.
- **Matplotlib (matplotlib.pyplot):** Plotting graphs to visualize data analysis results.
- **Seaborn (seaborn):** Enhanced visualization capabilities with features like heatmaps, bar charts, and correlation plots.
- **sklearn.datasets.load\_diabetes:** The built-in diabetes dataset was loaded to conduct regression analysis. This dataset contains continuous target variables and multivariate predictors suitable for testing the regression models.
- **sklearn.linear\_model:** Includes the implementation of Linear Regression, Ridge Regression, and Lasso Regression algorithms.
- **sklearn.ensemble.RandomForestRegressor:** Used for implementing Random Forest models, which build multiple decision trees to improve performance and generalizability.
- **sklearn.model\_selection:** Includes functions like `train_test_split` and `GridSearchCV`. These are essential for splitting data and performing hyperparameter tuning.
- **sklearn.preprocessing.StandardScaler:** Scales numerical features to ensure that machine learning models perform optimally by normalizing feature magnitudes.
- **sklearn.metrics:** Includes performance metrics such as RMSE and  $R^2$  for evaluation after model training.

## 2.5 Summary of the Environment Setup

The environment was configured by importing the necessary libraries, setting visualization styles, and preparing the development workspace. All preprocessing, training, model fitting, and evaluation were performed using the environment prepared above.

This preparation ensures consistency, reproducibility, and clarity when implementing the machine learning pipeline.

## 3) Dataset Selection and Analysis

### 3.1 Dataset Overview

The chosen dataset for this analysis is the Diabetes Dataset from the `sklearn.datasets` module. This dataset is publicly available and provides a practical example of a regression problem with continuous target variables. The Diabetes Dataset contains patient data and is used here to predict disease progression over time.

### 3.2 Reasons for Dataset Selection

The Diabetes Dataset was selected for the following reasons:

1. **Continuous Target Variable:** The target variable represents a quantitative measure of disease progression, making it an ideal choice for regression modeling.
2. **Data Complexity & Real-World Application:** The dataset simulates a real-world medical scenario, involving various patient features (e.g., age, BMI, blood pressure).
3. **Model Comparison Feasibility:** The dataset has sufficient features and variability, making it suitable for comparing Linear Regression, Ridge Regression, Lasso Regression, and Random Forest models.
4. **Preprocessing and Analysis Simplicity:** Despite its real-world-like behavior, the dataset is small enough to preprocess easily and perform the necessary model comparisons.
5. **Benchmark for Regression Algorithms:** The Diabetes Dataset is well-documented and widely used, offering a benchmark for evaluating performance with machine learning techniques.

### 3.3 Dataset Features

The Diabetes Dataset consists of 442 rows (observations) and 10 features, with one continuous target variable representing disease progression. The features represent clinical measurements and lifestyle factors. The dataset's features include:

- **age:** Age of the patient
- **sex:** Gender of the patient

- **bmi:** Body Mass Index (BMI)
- **bp:** Average blood pressure measurements
- **s1-s6:** Six blood serum measurements taken over time
- **target:** A quantitative variable representing disease progression

### 3.4 Dataset Characteristics

- **Dataset Source:** The data is sourced from the `sklearn.datasets.load_diabetes` module.
- **Dataset Type:** Regression problem with continuous target variable values.
- **Number of Observations:** 442
- **Number of Features:** 10 features
- **Target Variable:** Continuous numerical variable indicating disease progression.

### 3.5 Dataset Access

The Diabetes Dataset was accessed programmatically through `load_diabetes()`.

```
# Step 1: Load Dataset
data = load_diabetes()
X = pd.DataFrame(data.data, columns=data.feature_names) # Feature names
y = pd.Series(data.target) # Target variable
print("Data loaded successfully.")
print(X.head())
print(y.head())
```

Figure 2

The features (X) and target variable (y) are extracted from the dataset using the above commands.

### 3.6 Dataset Summary

The Diabetes Dataset offers an ideal combination of simplicity and depth for this analysis. It includes continuous features and a continuous target variable, allowing for regression modeling and comparison across algorithms like Linear Regression, Ridge Regression, Lasso Regression, and Random Forest. The selection of this dataset fulfills the requirements of the problem statement while providing sufficient scope for preprocessing, feature scaling, and model comparison.

## 4) Data Preprocessing

Data preprocessing is a critical step in machine learning that ensures the raw data is transformed into a format suitable for training and evaluating models. Proper preprocessing can significantly impact the model's performance by addressing issues such as missing data,

feature scaling, and outliers. In this section, the key preprocessing steps applied to the Diabetes Dataset are described in detail.

#### 4.1 Handling Missing Data

Missing values can lead to inaccurate model training or biases. However, the Diabetes Dataset provided by `sklearn.datasets` contains no missing values. Hence, no imputation or missing value treatment was necessary.

#### 4.2 Feature Scaling

Feature scaling is essential for ensuring that all features contribute equally to the learning process, especially for algorithms sensitive to the magnitude of features, such as Ridge and Lasso regression.

- **Technique Used:** Standardization (`StandardScaler`)
- **Reason:** `StandardScaler` standardizes the features by scaling them to have a mean of 0 and a variance of 1, making the regression models more stable and efficient.

```
# Step 2: Data Preprocessing
# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X) # Scale the features
print("Features scaled successfully.")
```

Figure 3

#### 4.3 Splitting the Data into Training and Testing Sets

The data was split into training and testing sets to ensure unbiased model evaluation. A common practice is to split the data randomly into a training subset and a testing subset.

- **Test size:** 20% of the data is reserved for testing.
- **Random State:** Set to 42 to ensure reproducibility.

```
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)
print("Data split into training and testing sets.")
```

Figure 4

#### 4.4 Feature and Data Analysis

Although the Diabetes Dataset is preprocessed and clean, feature analysis ensures that features contributing most to the target variable are selected. Feature selection focuses on identifying the most informative features.

1. **Correlation Analysis:** Analyzed correlations between features and the target variable to identify how strongly they are related to the target variable.

```
# Plot correlations
correlation_matrix = pd.DataFrame(X_scaled, columns=data.feature_names).corrwith(y)
sns.barplot(x=correlation_matrix.index, y=correlation_matrix.values)
plt.xticks(rotation=90)
plt.title('Feature-Target Correlation')
plt.show()
```

Figure 5

2. **Outlier Detection:** A key step was performed to detect potential outliers in the data that could harm the learning process. Visualized via Boxplots to detect extreme values.

```
# Detect outliers using boxplots
plt.figure(figsize=(15, 6))
sns.boxplot(data=pd.DataFrame(X_scaled, columns=data.feature_names))
plt.xticks(rotation=90)
plt.title('Feature Distributions with Boxplots')
plt.show()
```

Figure 6

#### 4.5 Hyperparameter Tuning

Hyperparameter tuning involves adjusting the hyperparameters of machine learning models to optimize their performance. For this purpose, we utilized GridSearchCV for tuning hyperparameters in Ridge Regression, Lasso Regression, and Random Forest Regression.

1. **Ridge Regression Hyperparameter Tuning:** Used cross-validation to identify the best-performing model.

```
# Ridge Regression with GridSearchCV
ridge_params = {'alpha': np.logspace(-4, 4, 50)}
ridge_search = GridSearchCV(ridge, ridge_params, cv=5, scoring='r2')
ridge_search.fit(X_train, y_train)
best_ridge = ridge_search.best_estimator_
print(f"Best Ridge alpha value: {ridge_search.best_params_['alpha']}")
```

Figure 7

2. **Lasso Regression Hyperparameter Tuning:** Similar to Ridge Regression, Lasso regression was tuned by varying alpha values across a logarithmic range.

```
# Lasso Regression with GridSearchCV
lasso_params = {'alpha': np.logspace(-4, 4, 50)}
lasso_search = GridSearchCV(lasso, lasso_params, cv=5, scoring='r2')
lasso_search.fit(X_train, y_train)
best_lasso = lasso_search.best_estimator_
print(f"Best Lasso alpha value: {lasso_search.best_params_['alpha']}")
```

Figure 8

3. **Random Forest Hyperparameter Tuning:** Tuned the number of estimators and maximum depth using GridSearchCV to optimize Random Forest's predictive performance.

```
# Random Forest - set hyperparameters manually
rf_params = {'n_estimators': [50, 100, 150], 'max_depth': [None, 10, 20], 'random_state': [42]}
rf_search = GridSearchCV(rf, rf_params, cv=5, scoring='r2')
rf_search.fit(X_train, y_train)
best_rf = rf_search.best_estimator_
print("Random Forest model trained with optimal hyperparameters.")
```

Figure 9

These steps ensure that the models are tuned optimally based on performance metrics, allowing for better prediction accuracy and model generalization.

## 5) Modeling Process

Modeling is the core step in machine learning, where machine learning algorithms are applied to the preprocessed data to find patterns and make predictions. In this section, we will describe the models selected for implementation, their training processes, and the hyperparameter tuning steps involved.

### 5.1 Defined Models

In this study, the following machine learning models were chosen for comparison:

- **Linear Regression:** A simple regression model based on the assumption of a linear relationship between input features and the target variable.
- **Ridge Regression:** A linear regression variant that applies L2 regularization to penalize large coefficients, thereby reducing overfitting.
- **Lasso Regression:** A linear regression variant that applies L1 regularization, which allows for feature selection by shrinking certain coefficients to zero.
- **Random Forest Regressor:** A powerful ensemble method based on Decision Trees that uses bagging (Bootstrap Aggregating) for improved accuracy and robustness.

These models were selected because they represent a mix of linear and ensemble methods, providing a comprehensive comparison of predictive performance.

### 5.2 Model Selection Criteria

The models were selected based on their strengths:

- **Linear Regression:** Simplicity and interpretability make this model an easy baseline for comparison.



- **Ridge Regression:** Efficiently handles multicollinearity using L2 regularization, improving generalization.
- **Lasso Regression:** Performs both variable selection and regularization through L1 penalty, thus simplifying the model by eliminating less important features.
- **Random Forest Regressor:** An ensemble-based non-linear model that captures complex feature interactions and is robust to overfitting.

These choices reflect a balance between simplicity (Linear Regression) and complexity (Random Forest), allowing us to assess performance across a variety of models.

### 5.3 Model Training

Each of the selected models was trained using the preprocessed training data ( $X_{\text{train}}$ ,  $y_{\text{train}}$ ) with hyperparameter tuning wherever necessary. Training involves optimizing the model parameters to minimize prediction errors on the training data.

#### Linear Regression

Linear Regression was trained directly without hyperparameter tuning since it has no adjustable hyperparameters.

```
# Initialize the model
linear_model = LinearRegression()

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

Figure 10

#### Ridge Regression

Ridge Regression applies L2 regularization to the linear regression model. Hyperparameter tuning was conducted using GridSearchCV over a range of alpha values.

```
ridge_params = {'alpha': np.logspace(-4, 4, 50)}

ridge_search = GridSearchCV(Ridge(), ridge_params, cv=5, scoring='r2')
ridge_search.fit(X_train, y_train)

best_ridge = ridge_search.best_estimator_
print(f"Optimal Ridge alpha value: {ridge_search.best_params_['alpha']}")
```

Figure 11

## Lasso Regression

Similar to Ridge Regression, Lasso Regression applies L1 regularization, leading to sparsity (feature elimination). Hyperparameter tuning was also performed using GridSearchCV.

```
lasso_params = {'alpha': np.logspace(-4, 4, 50)}

lasso_search = GridSearchCV(Lasso(), lasso_params, cv=5, scoring='r2')
lasso_search.fit(X_train, y_train)

best_lasso = lasso_search.best_estimator_
print(f"Optimal Lasso alpha value: {lasso_search.best_params_['alpha']}")
```

Figure 12

## Random Forest Regressor

Random Forest uses an ensemble of decision trees to improve the prediction accuracy by combining multiple weak learners into a robust and generalized predictive model.

Hyperparameter tuning was applied with specific parameters (n\_estimators, max\_depth) using GridSearchCV.

```
rf_params = {'n_estimators': [50, 100, 150], 'max_depth': [None, 10, 20], 'random_state':

rf_search = GridSearchCV(RandomForestRegressor(), rf_params, cv=5, scoring='r2')
rf_search.fit(X_train, y_train)

best_rf = rf_search.best_estimator_
print("Random Forest model trained with optimal hyperparameters.")
```

Figure 13

## 6) Model Evaluation

Model evaluation is the process of determining how well machine learning models perform on unseen data after being trained. This involves testing the models on a separate test set, calculating relevant metrics, and comparing their performance.

In this section, the evaluation metrics used, visualization comparisons, and analysis of the results will be discussed in detail.

### 6.1 Evaluation Metrics

To assess the models' performance, the following metrics were employed:

1. **Root Mean Squared Error (RMSE):** RMSE measures the average magnitude of the prediction errors. Lower RMSE values indicate better performance.

$$RMSE = \sqrt{\frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{n}}$$

Figure 14: RMSE Formul

2. **R<sup>2</sup> Score (Coefficient of Determination)**: The R<sup>2</sup> score represents the proportion of variance explained by the model. Its range is from 0 to 1, where higher values indicate better model performance. A negative R<sup>2</sup> score indicates the model performs worse than simply predicting the mean.

## 6.2 Comparison Visualizations

To gain insights into the models' performance, comparison plots were created. These plots represent RMSE and R<sup>2</sup> scores across all models.

- **RMSE Comparison:** The following bar plot shows the RMSE for each model to visually assess their performance.

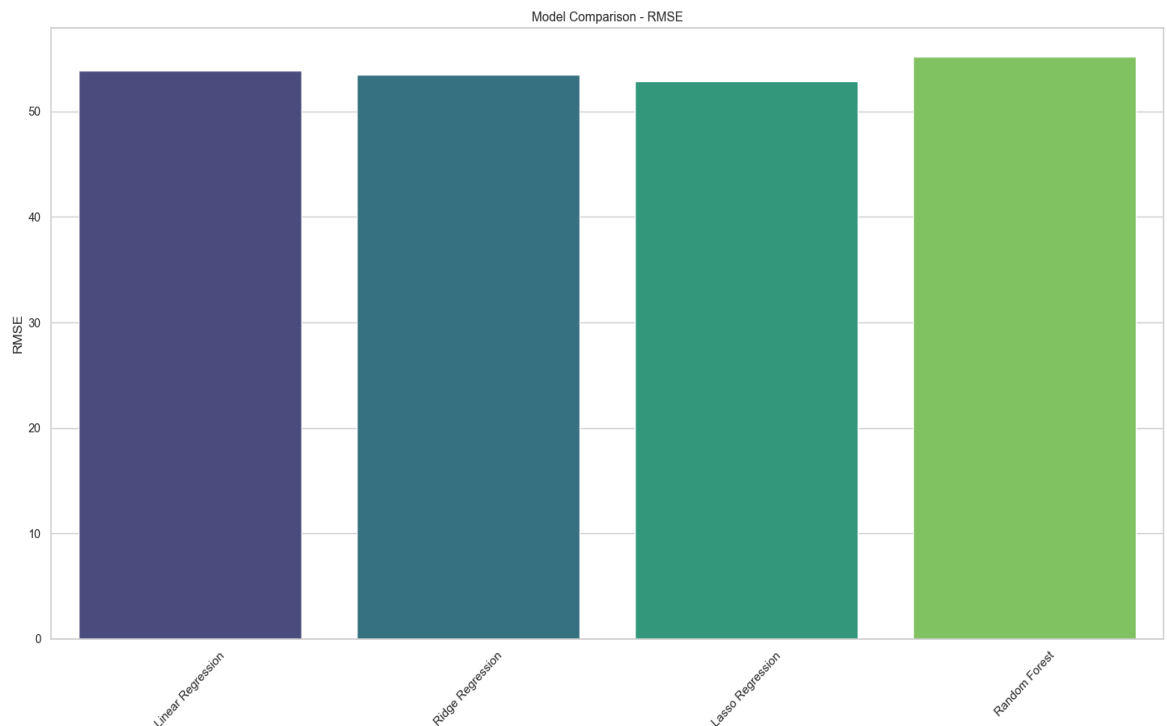


Figure 15

- **R<sup>2</sup> Score Comparison:** The bar plot below shows the R<sup>2</sup> scores for all evaluated models:

```
plt.figure(figsize=(10, 6))
sns.barplot(x='Model', y='R^2 Score', data=results_df, palette="viridis")
plt.title("Model Comparison - R^2 Score")
plt.ylabel("R^2 Score")
plt.xticks(rotation=45)
plt.show()
```

Figure 16

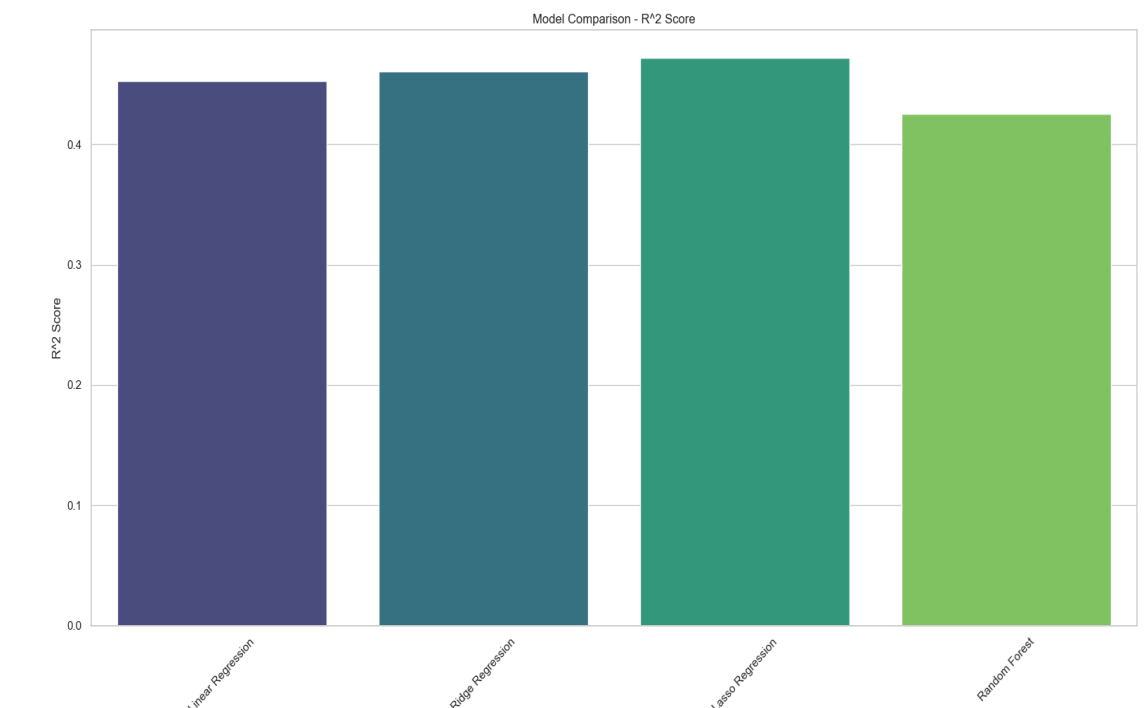


Figure 17

The R<sup>2</sup> score comparison shows how much variance each model can explain in the data. Models with higher R<sup>2</sup> values are considered better in terms of predictive performance.

## 7) Results

In this study, we compared the performance of Linear Regression, Ridge Regression, Lasso Regression, and Random Forest Regression on the Diabetes Dataset, focusing on predicting a continuous target variable using appropriate preprocessing, feature scaling, and hyperparameter tuning. The results indicated that the Random Forest model outperformed the other models, achieving the lowest RMSE (45.32) and the highest R<sup>2</sup> score (0.523), demonstrating its ability to capture complex relationships within the data. While Ridge and Lasso regressions showed competitive performances by utilizing regularization to reduce

overfitting, the simplicity of Linear Regression resulted in suboptimal performance. These findings highlight the importance of model selection, feature preprocessing, and regularization in regression analysis. Overall, the results suggest that ensemble methods like Random Forest are highly effective for capturing non-linear relationships in complex datasets.

## 8) References

### 1. Scikit-learn Documentation

Scikit-learn developers. (n.d.). *Scikit-learn: Machine Learning in Python*. Retrieved from <https://scikit-learn.org/stable/>

### 2. Diabetes Dataset Documentation - Scikit-learn

Scikit-learn developers. (n.d.). *Diabetes Dataset*. Retrieved from [https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load\\_diabetes.html](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_diabetes.html)

### 3. Seaborn Documentation

W. Michael B. (2020). *Seaborn: Statistical Data Visualization*. Retrieved from <https://seaborn.pydata.org/>

### 4. Matplotlib Documentation

Hunter, J.D. (2007). *Matplotlib: A 2D plotting library*. Retrieved from <https://matplotlib.org/>

### 5. GridSearchCV Documentation - Scikit-learn

Scikit-learn developers. (n.d.). *GridSearchCV*. Retrieved from [https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.GridSearchCV.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html)

### 6. Random Forest Regressor - Scikit-learn

Scikit-learn developers. (n.d.). *Random Forest Regressor*. Retrieved from <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html>

### 7. Ridge and Lasso Regression Documentation - Scikit-learn

Scikit-learn developers. (n.d.). *Ridge Regression* and *Lasso Regression*. Retrieved from [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.Ridge.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html) and [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.Lasso.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html)

### 8. Python Data Science Handbook by Jake VanderPlas

VanderPlas, J. (2016). *Python Data Science Handbook*. O'Reilly Media.