DATA MINING PROJECT REPORT (Question 1)

Title: Titanic Dataset Analysis Using Decision Tree, Random Forest, and Support Vector Machine

Prepared by:

Name: Samet Kaya

Student Number: 201401019

Course: Data Mining

Instructor: Abdulgani Kahraman, Assistant Professor

Date: 12.08.2024

Dataset Link:

https://www.kaggle.com/c/titanic/data?select=train.csv

Project Overview:

In this project, I analyzed the Titanic dataset to predict survival using three machine learning models: Decision Tree, Random Forest, and Support Vector Machine (SVM). I performed data preprocessing, created visualizations to explore the data, and used hyperparameter tuning to optimize the performance of each model. After training the models, I evaluated their performance using metrics such as accuracy, precision, recall, and F1 score. I also used visualizations and feature importance analysis to interpret the results and compared the models to identify the best-performing one. Through this process, I was able to determine which model delivered the most reliable and accurate predictions for survival.

Introduction

I chose the Titanic dataset for this project because it is a well-documented and widely-used dataset, and I thought it would be ideal for exploring machine learning techniques.

My main goal in this project was to predict the survival of Titanic passengers using machine learning models. I started by studying the dataset carefully, visualizing key trends, and preprocessing the data to ensure it was clean and ready for analysis. Afterward, I trained three machine learning models: Decision Tree, Random Forest, and Support Vector Machines (SVM).

I used various visualizations to explore the dataset and explain the key features affecting survival. During data preprocessing, I handled missing values, encoded categorical variables such as gender and embarkation ports, and prepared the features for modeling. After training each model, I evaluated their performance using accuracy, precision, recall, and F1 score. I also used confusion matrices to visualize how well each model classified the passengers into survivors and non-survivors.

By analyzing the results, I identified how passenger features such as age, gender, and ticket fare influenced survival. Finally, I compared the performance of all three models and selected the best-performing method. This study demonstrates a comprehensive approach to data preprocessing, visualization, and the evaluation of machine learning models for predictive analysis.

Titanic Dataset

The Titanic dataset is a collection of information about the passengers on the Titanic ship. This dataset is often used in machine learning and data analysis because it includes various features about the passengers and their survival.

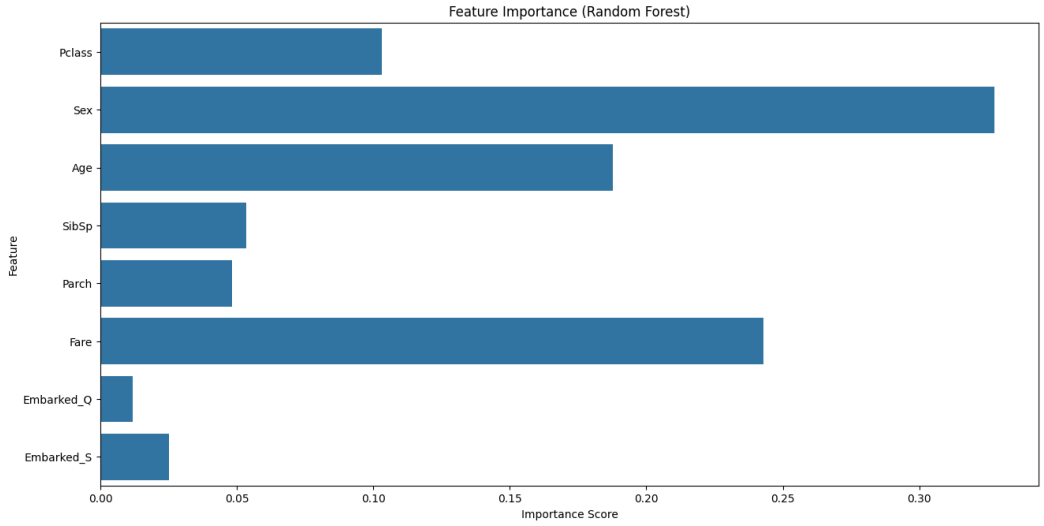
Here are the main variables in the dataset:

1. **Survived**: This is the target variable. It shows if the passenger survived (1) or not (0).
2. **Pclass**: The ticket class of the passenger. It has three categories:
    - 1: First class
    - 2: Second class
    - 3: Third class
3. **Name**: The name of the passenger.
4. **Sex**: The gender of the passenger (male or female).
5. **Age**: The age of the passenger. Some values are missing and need to be handled.
6. **SibSp**: The number of siblings or spouses the passenger had on board.
7. **Parch**: The number of parents or children the passenger had on board.
8. **Ticket**: The ticket number of the passenger.
9. **Fare**: The amount of money the passenger paid for the ticket.
10. **Cabin**: The cabin number assigned to the passenger. Many values are missing.
11. **Embarked**: The port where the passenger boarded the ship. The values are:
    - C: Cherbourg
    - Q: Queenstown
    - S: Southampton

The dataset contains both numerical and categorical variables. Some variables, like "Name" and "Ticket," are not very useful for prediction, while others, like "Pclass," "Sex," and "Fare," are very important for survival prediction.

Feature Importance

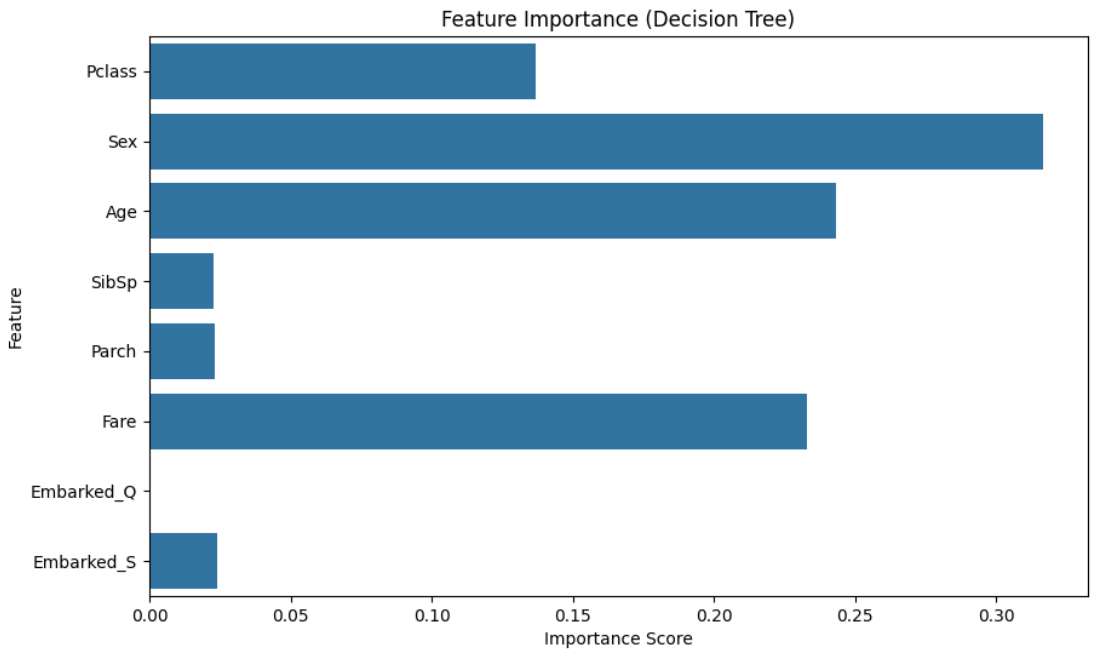
Feature importance is a measure used in machine learning to identify which features (variables) in the dataset are most relevant for predicting the target variable. In this analysis, we use the **Random Forest** and **Decision Tree** models to extract and evaluate feature importance.



The image above shows the feature importance rankings from a Random Forest model for predicting survival. Here are my comments based on the graph:

- **Sex (Gender)**: This is the most important feature, with the highest importance score. Gender has the strongest influence on the model's predictions, indicating that it plays a critical role in survival.
- **Fare (Ticket Price)**: The second most important feature. It highlights that economic factors (ticket price) significantly influence survival likelihood.
- **Age**: Another highly influential feature, showing that age impacts survival predictions considerably.
- **Pclass (Passenger Class)**: Moderately important. This feature suggests that the passenger's class still matters but less than Sex, Fare, and Age.
- **SibSp (Number of Siblings/Spouses)**: Has moderate importance, implying that family connections on board have some effect on survival chances.
- **Parch (Number of Parents/Children)**: Has relatively low importance, suggesting a smaller influence on survival predictions.
- **Embarked_Q and Embarked_S (Boarding Ports)**: The least important features, indicating that where a passenger boarded has minimal impact on survival predictions.

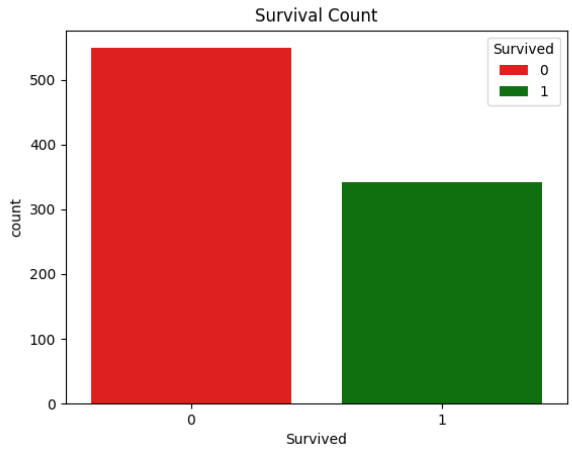
This analysis indicates that demographic (Sex, Age) and economic (Fare) factors are the strongest predictors of survival on the Titanic. Other features like passenger class and family connections are moderately important, while boarding location is the least significant.

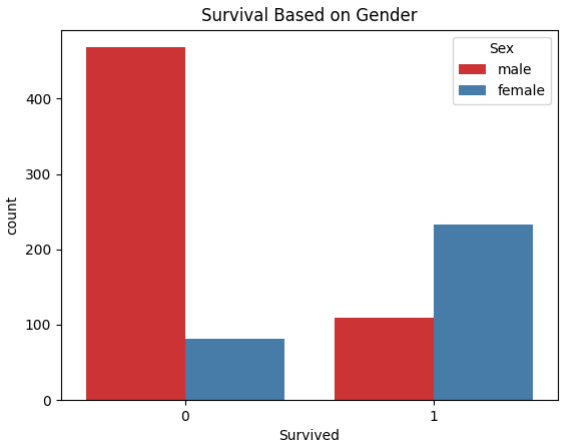


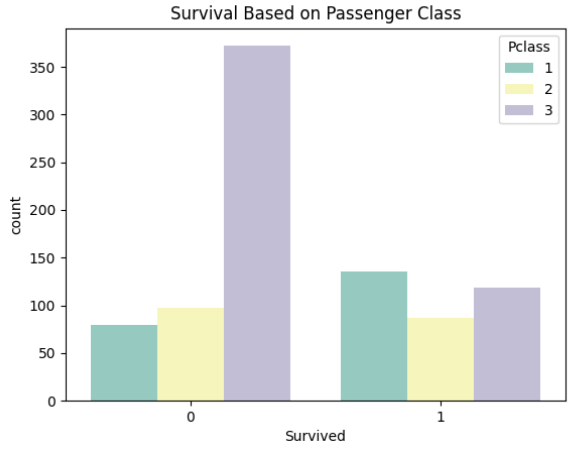
The image above shows the feature importance rankings from a Decision Tree model for predicting survival. Here are my comments based on the graph:

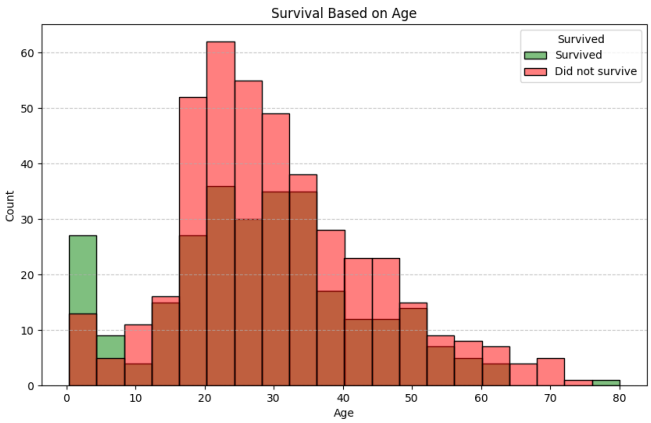
• **Sex (Gender)**: This is the most important feature, with the highest importance score. Gender has the strongest influence on the model's predictions, indicating that it plays a critical role in survival.
• **Age**: The second most important feature. It shows that age significantly impacts survival predictions, highlighting its critical role.
• **Fare (Ticket Price)**: The third most important feature. It highlights that economic factors (ticket price) influence survival likelihood but are slightly less important than Age.
• **Pclass (Passenger Class)**: Moderately important. This feature suggests that the passenger's class still matters but less than Sex, Age, and Fare.
• **SibSp (Number of Siblings/Spouses)**: Has limited importance, implying that family connections on board have some effect on survival chances.
• **Parch (Number of Parents/Children)**: Has relatively low importance, suggesting a smaller influence on survival predictions.
• **Embarked_Q and Embarked_S (Boarding Ports)**: The least important features, indicating that where a passenger boarded has minimal impact on survival predictions.

This analysis indicates that demographic (Sex, Age) and economic (Fare) factors are the strongest predictors of survival on the Titanic. Other features like passenger class and family connections are moderately important, while boarding location is the least significant.









I created these graphs myself using Python visualization methods.. In these graphs, 0 represents the people who did not survive, and 1 represents the people who survived.

Machine Learning Methods

1-) Decision Tree

A **Decision Tree** is a simple and popular machine learning method. It is used for both classification and regression tasks. The tree splits the data into smaller parts by asking "yes" or "no" questions at each step. These questions are based on the values of the features in the dataset.

1. **How It Works**:

- The tree starts with a root node, which is the top of the tree.
- At each node, the data is divided into two or more groups based on a condition (e.g., "Is Age > 30?").
- This process continues until the tree reaches a "leaf node," where no further splitting is possible. Leaf nodes represent the final prediction.

2. **How It Decides the Split**:
   - The tree uses measures like **Gini Index** or **Entropy** to find the best question to ask.
   - It selects the condition that splits the data into the most "pure" groups (where one class dominates the group).
3. **Advantages**:
   - Easy to understand and explain.
   - Can handle both numerical and categorical data.
   - Does not need much data preparation (like scaling or normalizing).
4. **Disadvantages**:
   - Prone to **overfitting**, especially if the tree is very deep.
   - Small changes in the data can result in a completely different tree.

Decision Trees are good for simple problems but may not work well for complex datasets without proper tuning.

Implementation on the Titanic dataset using Python

Importing the Decision Tree Library →

```
from sklearn.tree import DecisionTreeClassifier
```

Hyperparameter Tuning for Decision Tree →

```
# Decision Tree
dt_param_grid = {'criterion': ['gini', 'entropy'], 'max_depth': [None, 10, 20, 30]}
dt_model = tune_hyperparameters(DecisionTreeClassifier(random_state=42), dt_param_grid, X_train, y_train, "Decision Tree")
```

- The **tune_hyperparameters** function applies a grid search to find the best hyperparameters for the Decision Tree.
- Parameters like criterion (split function), and max_depth (tree depth) are optimized.
- The **random_state=42** ensures consistent results on every execution.

Training the Decision Tree Model →

```
evaluate_model(dt_model, X_val, y_val, X_test, y_test, "Tuned Decision Tree")
```

- The model is trained on the training data and evaluated on both validation and test datasets.
- Metrics such as accuracy, precision, recall, and F1-score are printed using the custom **evaluate_model** function.

Feature Importance Visualization →

- The feature_importances_ attribute of the Decision Tree model provides the importance of each feature.
- The bar chart shows how much each feature contributes to the predictions.

```
# Feature importance for Decision Tree
importances_dt = dt_model.feature_importances_
plt.figure(figsize=(10, 6))
sns.barplot(x=importances_dt, y=features)
plt.title("Feature Importance (Decision Tree)")
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.show()
```

Confusion Matrix for Decision Tree →

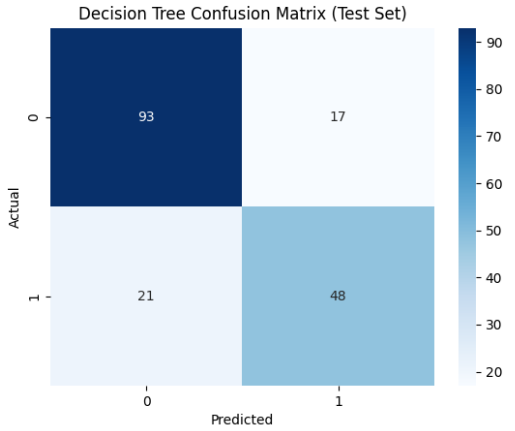
```
# Confusion matrices
models = {
    "Decision Tree": dt_model.predict(X_test),
    "Random Forest": rf_model.predict(X_test),
    "SVM": svm_model.predict(X_test)
}

for model_name, y_pred in models.items():
    conf_matrix = confusion_matrix(y_test, y_pred)
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=[0, 1], yticklabels=[0, 1])
    plt.title(f"{model_name} Confusion Matrix (Test Set)")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()
```

- The confusion matrix displays the model's performance on the test set.
- It shows true positives, true negatives, false positives, and false negatives.
- A heatmap is used to visualize the confusion matrix, making it easier to interpret.

Summary of Results

1. **Decision Tree Hyperparameters**: The optimal parameters, such as criterion and max_depth, are tuned to enhance the model's performance.
2. **Evaluation Metrics**: The model's accuracy, precision, recall, and F1 scores are calculated on both validation and test datasets, providing a comprehensive evaluation of performance.
3. **Feature Importance**: Gender (Sex) is the most important feature, followed by Age and Fare. Features like Parch and Embarked_Q have minimal impact on predictions.
4. **Confusion Matrix**: The matrix shows how well the model classifies survival on the test set, helping to identify areas of improvement.

This implementation combines hyperparameter tuning, model evaluation, and visualizations to analyze and improve the Decision Tree model for survival prediction on the Titanic dataset.




```
Tuned Decision Tree Validation Performance:
Accuracy: 0.84
Precision: 0.79
Recall: 0.79
F1 Score: 0.79

Tuned Decision Tree Test Performance:
Accuracy: 0.79
Precision: 0.74
Recall: 0.70
F1 Score: 0.72
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Best parameters for Random Forest: {'max_depth': 10, 'min_samples_split': 5, 'n_estimators': 200}
Best score: 0.8250
```

2-) Random Forest

Random Forest is an advanced machine learning method based on Decision Trees. It is used for both classification and regression tasks. Instead of building one single tree, Random Forest creates many Decision Trees and combines their results for better accuracy and stability.

1. **How It Works:**

- Random Forest builds multiple Decision Trees using different parts of the dataset. Each tree is trained on a random sample of the data (called bootstrapping).
- At each split in the tree, it randomly selects a subset of features to choose the best split. This reduces the chance of overfitting.
- The final prediction is made by combining the outputs of all trees:
- For classification, it takes a majority vote (the class with the most votes is the result).
- For regression, it calculates the average of the predictions.

2. **Why Random Forest Is Powerful:**

- Diversity: Each tree is slightly different, so the overall model is less sensitive to noise in the data.
- Stability: It is less likely to overfit compared to a single Decision Tree.
- Feature Importance: Random Forest provides insights into which features are most important for predictions.

3. **Advantages:**

- Works well with large and complex datasets.
- Handles both numerical and categorical data.
- Resistant to overfitting because it averages results from many trees.

4. **Disadvantages:**

- Slower to train and predict compared to a single Decision Tree.
- Requires more computational resources.
- The results can be harder to interpret because of the many trees involved.

Random Forest is a great choice when accuracy and robustness are more important than simplicity. It performs well on many types of datasets and is widely used in machine learning projects.

Implementation on the Titanic dataset using Python

Importing Random Forest Library →

```
from sklearn.ensemble import RandomForestClassifier
```

Creating the Random Forest Model →

```
# Random Forest
rf_param_grid = {'n_estimators': [100, 200], 'max_depth': [None, 10], 'min_samples_split': [2, 5]}
rf_model = tune_hyperparameters(RandomForestClassifier(random_state=42), rf_param_grid, X_train, y_train, "Random Forest")
```

- The **tune_hyperparameters** function applies a grid search to find the best hyperparameters for the Random Forest model.
- Key parameters include n_estimators (number of trees), max_depth (tree depth), and min_samples_split (minimum samples for node splitting).
- The **random_state=42** ensures consistent and reproducible results on every execution.

Training the Random Forest Model →

```
evaluate_model(rf_model, X_val, y_val, X_test, y_test, "Tuned Random Forest")
```

- The model is trained on the training data and evaluated on both validation and test datasets.
- The **evaluate_model** function calculates metrics like accuracy, precision, recall, and F1-score for comprehensive evaluation.

Feature Importance Visualization →

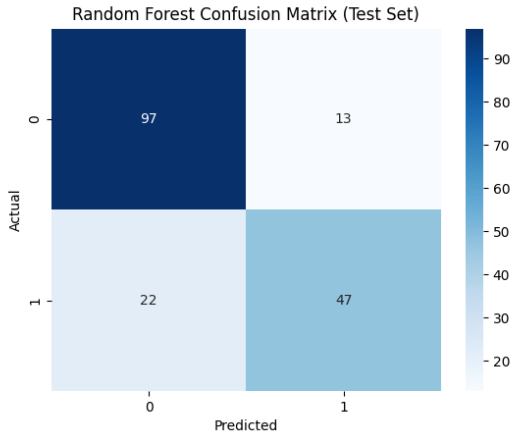
```
# ------------------------------
# Feature Importance and Confusion Matrices
# ------------------------------
# Feature importance for Random Forest
importances_rf = rf_model.feature_importances_
features = X.columns
plt.figure(figsize=(10, 6))
sns.barplot(x=importances_rf, y=features)
plt.title("Feature Importance (Random Forest)")
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.show()
```

- The **.feature_importances_** attribute provides the importance of each feature in the Random Forest model.
- A bar chart is plotted to visualize feature contributions, making it easier to identify the most significant predictors

Visualizing the Confusion Matrix →

```python
# Confusion matrices
models = {
    "Decision Tree": dt_model.predict(X_test),
    "Random Forest": rf_model.predict(X_test),
    "SVM": svm_model.predict(X_test)
}

for model_name, y_pred in models.items():
    conf_matrix = confusion_matrix(y_test, y_pred)
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=[0, 1], yticklabels=[0, 1])
    plt.title(f"{model_name} Confusion Matrix (Test Set)")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()
```

- The confusion matrix shows the model's performance on the test set, highlighting the number of correct and incorrect predictions for each class.
- A heatmap is used to visualize the confusion matrix with annotations (annot=True) to display actual numbers.
- The x-axis represents predicted labels, and the y-axis represents actual labels.




```
Tuned Random Forest Validation Performance:
Accuracy: 0.88
Precision: 0.87
Recall: 0.79
F1 Score: 0.83

Tuned Random Forest Test Performance:
Accuracy: 0.80
Precision: 0.78
Recall: 0.68
F1 Score: 0.73
Fitting 5 folds for each of 4 candidates, totalling 20 fits
Best parameters for Support Vector Machine: {'C': 1, 'gamma': 'scale', 'kernel': 'linear'}
Best score: 0.7897
```

3-) Support Vector Machine (SVM)

Support Vector Machine (SVM) is a powerful machine learning method used for classification and regression tasks. It works by finding the best line (or hyperplane) that separates the data into different classes.

1. **How It Works:**

- Separating Data: SVM tries to find the line that divides the data into two groups (e.g., survived vs. not survived). This line is called a "decision boundary."
- Maximizing the Margin: SVM selects the line that has the largest distance (margin) from the nearest data points on both sides. These closest points are called "support vectors."
- Kernel Trick: If the data cannot be separated with a straight line, SVM uses a "kernel function" to transform the data into a higher dimension where separation is possible.

2. **Why SVM is Powerful:**

- Works Well with Complex Data: SVM is effective when the data is not easy to separate with simple lines.
- Flexible: By using kernels, it can handle non-linear relationships.
- Small Datasets: SVM performs well with small datasets and high-dimensional data.

3. **Advantages:**

- Finds the best boundary between classes.
- Works well with high-dimensional data (many features).
- Can handle both linear and non-linear data using the kernel trick.

4. **Disadvantages:**

- Slow to train and predict on large datasets.
- Requires careful tuning of parameters like kernel type and regularization.
- Not as interpretable as Decision Trees or Random Forests.

Implementation on the Titanic dataset using Python

Importing the SVM Library →

```
from sklearn.svm import SVC
```

Hyperparameter Tuning for SVM →

```
# SVM
svm_param_grid = {'C': [1, 10], 'kernel': ['linear', 'rbf'], 'gamma': ['scale']}
svm_model = tune_hyperparameters(SVC(random_state=42), svm_param_grid, X_train, y_train, "Support Vector Machine")
```

- The tune_hyperparameters function uses grid search to find the best combination of hyperparameters for the SVM model.

  Key parameters include:

  - C: Regularization strength, which balances model accuracy on training data and generalization to unseen data.
  - kernel: Specifies the kernel type (linear or radial basis function, rbf) used for transforming data.
  - gamma: Determines the influence of individual data points on the decision boundary (used in rbf kernel).
  - The random_state=42 ensures reproducibility by controlling the random processes.

Training the SVM Model →

```
evaluate_model(svm_model, X_val, y_val, X_test, y_test, "Tuned Support Vector Machine")
```

- The model is trained on the training dataset and evaluated on both validation and test datasets.
- The evaluate_model function computes evaluation metrics such as accuracy, precision, recall, and F1-score.

Confusion Matrix for SVM →

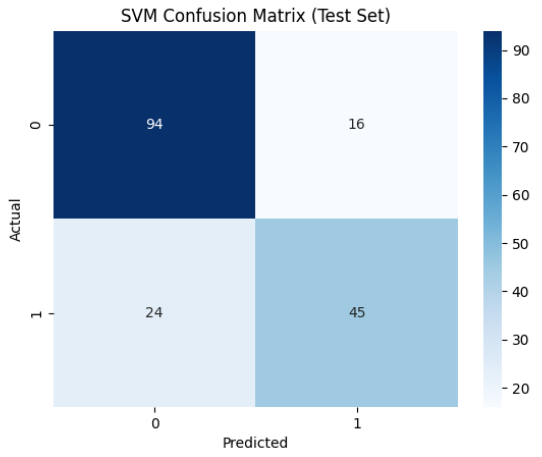
```
# Confusion matrices
models = {
    "Decision Tree": dt_model.predict(X_test),
    "Random Forest": rf_model.predict(X_test),
    "SVM": svm_model.predict(X_test)
}

for model_name, y_pred in models.items():
    conf_matrix = confusion_matrix(y_test, y_pred)
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=[0, 1], yticklabels=[0, 1])
    plt.title(f"{model_name} Confusion Matrix (Test Set)")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()
```

- The confusion matrix displays the classification performance of the SVM model on the test dataset.
- The heatmap visualizes the confusion matrix, with annot=True displaying the numerical values in each cell.
- The x-axis represents predicted labels, while the y-axis represents actual labels.

Summary of Results

1. **SVM Hyperparameters**: The best combination of C, kernel, and gamma is identified using grid search to optimize the model's performance.
2. **Evaluation Metrics**: The model's performance on both validation and test datasets is evaluated using metrics like accuracy, precision, recall, and F1-score, providing insights into its predictive ability.
3. **Confusion Matrix**: The confusion matrix highlights the model's classification accuracy, including true positives, true negatives, false positives, and false negatives.

This implementation demonstrates the application of SVM for survival prediction on the Titanic dataset, with hyperparameter tuning, evaluation metrics, and visualization of results.




```
Tuned Support Vector Machine Validation Performance:
Accuracy: 0.79
Precision: 0.76
Recall: 0.65
F1 Score: 0.70

Tuned Support Vector Machine Test Performance:
Accuracy: 0.78
Precision: 0.74
Recall: 0.65
F1 Score: 0.69
```

Comparison and Best Model Selection

Based on the provided metrics, we compare the Decision Tree, Random Forest, and Support Vector Machine (SVM) models to determine the best-performing approach. Below is a summary of their performance:

Validation Performance

- Decision Tree : Accuracy (84%), Precision (79%), Recall (79%), F1 Score (79%)
- Random Forest : Accuracy (88%), Precision (87%), Recall (79%), F1 Score (83%)
- SVM : Accuracy (79%), Precision (76%), Recall (65%), F1 Score (70%)

Observation:

- Random Forest has the highest validation accuracy (88%) and F1 score (83%), making it the best-performing model on validation data.
- Decision Tree also performs well, but it is slightly outperformed by Random Forest in accuracy and F1 score.
- SVM has lower recall and F1 score, indicating that it struggles to identify positive cases (survivors).

Test Performance

- **Decision Tree :** Accuracy (79%), Precision (74%), Recall (70%), F1 Score (72%)
- **Random Forest :** Accuracy (80%), Precision (78%), Recall (68%), F1 Score (73%)
- **SVM :** Accuracy (78%), Precision (74%), Recall (65%), F1 Score (69%)

Observation:

- Random Forest slightly outperforms Decision Tree in test accuracy (80%) and F1 score (73%), maintaining a balanced performance between precision and recall.
- Decision Tree is close in performance to Random Forest, but its slightly lower precision and F1 score make it less favorable.
- SVM performs the weakest on the test set, with lower recall and F1 score, indicating it fails to capture a significant number of survivors.

Best Performing Model

Random Forest is the best-performing model. Why:

- **Higher Accuracy:** Random Forest achieves the highest accuracy on both validation (88%) and test datasets (80%).
- **Balanced Precision and Recall:** With precision (78%) and recall (68%), Random Forest ensures reliable performance in identifying both survivors and non-survivors.
- **Robustness:** The ensemble approach of Random Forest reduces overfitting and increases stability, making it a robust choice for classification tasks.
- **Feature Importance:** Random Forest provides feature importance insights, helping to understand which factors (e.g., Sex, Age, Fare) contribute most to survival predictions.

Why Not the Other Models?

Decision Tree:

- Although simple and interpretable, Decision Tree slightly underperforms compared to Random Forest in accuracy and F1 score.
- It is more prone to overfitting due to its reliance on a single tree structure.

SVM:

- SVM has poor recall and F1 score, indicating it fails to correctly identify a significant portion of survivors.
- While it is effective in certain scenarios, its performance on this dataset makes it unsuitable.

Conclusion

In this analysis, I focused on predicting Titanic passenger survival using three machine learning models: Decision Tree, Random Forest, and Support Vector Machine (SVM). I paid close attention to critical stages such as data preprocessing, feature selection, and hyperparameter tuning, ensuring each step was aligned with the dataset's characteristics and the models' requirements. Below, I explain these steps in detail:

Data Preprocessing

To prepare the dataset for modeling, I handled missing values systematically. For instance, missing values in the "Age" column were replaced with the median age to avoid distorting the distribution, while missing "Embarked" values were filled with the most frequent category to maintain consistency. I removed irrelevant columns like "PassengerId," "Name," "Ticket," and "Cabin," which did not contribute to survival prediction. Additionally, categorical features such as "Sex" and "Embarked" were encoded into numerical values to ensure compatibility with the models.

Feature Selection

For feature selection, I chose variables that were directly relevant to survival prediction, such as "Age," "Fare," "Pclass," and "Sex." By excluding identifiers and redundant information, I ensured the models focused on meaningful data. This approach not only reduced noise in the dataset but also improved the models' interpretability and efficiency.

Hyperparameter Tuning

Hyperparameter tuning was a vital part of the analysis. For each model, I used GridSearchCV to systematically explore different parameter combinations and optimize their performance.

- For the Decision Tree, I adjusted parameters like criterion (e.g., "gini" or "entropy") and max_depth to balance model complexity and generalizability.
- For the Random Forest, I experimented with the number of estimators (n_estimators), maximum depth (max_depth), and minimum samples required for splitting (min_samples_split) to enhance ensemble performance.
- For the SVM, I tuned parameters like C (regularization), kernel (linear or rbf), and gamma to find the best fit for the dataset.

The models were evaluated using accuracy, precision, recall, and F1 score. These metrics helped me identify each model's strengths and weaknesses. Confusion matrices provided additional insights into classification performance, highlighting areas where the models performed well or struggled.

Key Findings

Random Forest emerged as the best-performing model, achieving a validation accuracy of 88% and a test accuracy of 80%. It demonstrated balanced precision (78%) and recall (68%), making it robust in predicting survivors and non-survivors.

Decision Tree performed well, with a validation accuracy of 84% and a test accuracy of 79%. However, its slightly lower F1 score (72%) indicated that it was less optimal compared to Random Forest.

SVM achieved a validation accuracy of 79% and a test accuracy of 78%. While its accuracy was comparable to the other models, its recall (65%) suggested challenges in identifying survivors effectively.

Reflection

By focusing on key stages like data preprocessing, feature selection, and hyperparameter tuning, I ensured the models were both optimized and interpretable. This analysis highlights the importance of carefully understanding each stage of the machine learning pipeline to achieve reliable and meaningful results. Random Forest proved to be the most effective model for this dataset due to its ability to handle complex relationships between features while maintaining robust performance.

DATA MINING PROJECT REPORT (Question 2)

Title: K-Means Clustering Analysis of Wholesale Customers Dataset

Prepared by:

Name: Samet Kaya

Student Number: 201401019

Course: Data Mining

Instructor: Abdulgani Kahraman, Assistant Professor

Date: 12.08.2024

Dataset Link:

https://archive.ics.uci.edu/dataset/292/wholesale+customers

Project Overview:

This report focuses on applying the K-Means clustering algorithm to group customers based on their purchasing behavior in the "Wholesale Customers" dataset. The analysis includes steps for data preprocessing, determining the optimal number of clusters, performing clustering, and evaluating the quality of the clusters. Metrics like the silhouette score are used to assess the clustering quality, along with cluster interpretation and visualizations. The goal is to identify distinct customer segments that can be used for targeted marketing strategies.

Introduction

This project utilizes the 'Wholesale Customers' dataset due to its accessibility, comprehensive documentation, and suitability for clustering analysis. The dataset, publicly available, provides detailed information on customer spending across various product categories. This makes it ideal for identifying distinct customer segments. Prior experience with this dataset allowed me to focus on refining the clustering methodology rather than data exploration.

K-Means clustering was employed to group customers based on their spending patterns. Clustering is an unsupervised machine learning technique that groups similar data points together. The primary objective was to identify meaningful customer segments and assess the quality of the resulting clusters.

Data preprocessing involved standardizing numerical features to ensure equal contribution of all variables to the clustering process. The optimal number of clusters was determined using the elbow method. The silhouette score was used to evaluate cluster quality, measuring how well data points fit within their respective clusters.

The K-Means algorithm effectively identified distinct customer segments within the dataset. These insights can be valuable for implementing targeted marketing strategies, optimizing inventory management, and fostering stronger customer relationships. This project demonstrates the practical application of clustering techniques in extracting valuable insights from data.

Wholesale Customers Dataset

The dataset used for this analysis is the **Wholesale Customers** dataset, which can be found on the UCI Machine Learning Repository. It contains customer spending data across different product categories. The dataset has several columns representing various attributes, such as:

- **Customer ID**: Unique identifier for each customer.
- **Annual spending (Monetary value)**: Customer spending in different categories.
- **Other features**: Representing different spending habits across various product categories (e.g., fresh, milk, grocery, frozen, etc.).

Data Preprocessing

Before applying the K-Means clustering algorithm, I carefully prepared the dataset to ensure it was suitable for analysis. This step involved inspecting the data, selecting relevant features, and standardizing the values to ensure all variables contributed equally to the clustering process.

1. Dataset Inspection

First, I loaded the "Wholesale Customers" dataset and performed an initial inspection. This included displaying the first few rows and analyzing the dataset's structure using the info() method. This step allowed me to confirm that the dataset contained no missing values and that most of the columns were numerical, which is ideal for clustering.

2. Feature Selection

The dataset contained some columns, such as identifiers or categorical variables, that were not useful for clustering. Specifically, I excluded the first two columns (Channel and Region) because they were likely categorical identifiers that did not directly contribute to customer segmentation based on spending patterns. Instead, I focused on numerical features such as "Fresh," "Milk," "Grocery," and other spending categories, as these variables were more relevant for grouping customers.

3. Standardization

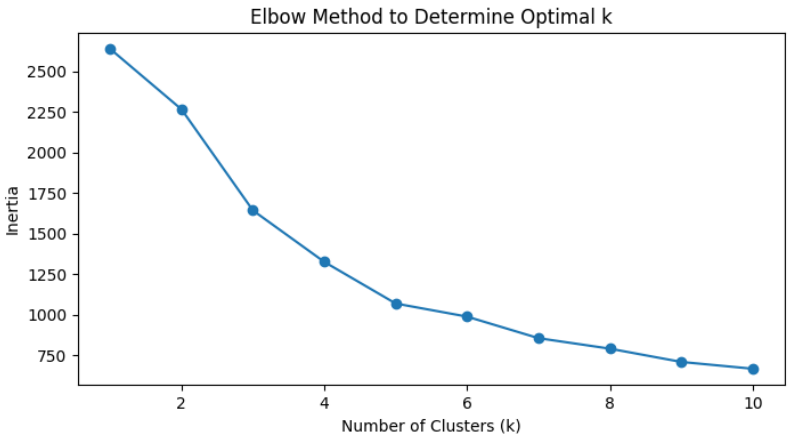
Since the features in the dataset had varying scales (e.g., some spending categories had much higher numerical values than others), I standardized the data using the StandardScaler from Scikit-learn. Standardization ensured that all features had a mean of 0 and a standard deviation of 1. This step was critical because K-Means clustering relies on distance-based calculations, and unscaled data could cause features with larger ranges to dominate the clustering process.

By the end of preprocessing, the dataset was clean, complete, and standardized, making it ready for clustering. This preprocessing pipeline ensured that the clustering results were meaningful and not skewed by differences in feature scales or irrelevant variables.

Determing the Number of Clusters (k)

Choosing the optimal number of clusters (k) is a crucial step in K-Means clustering. In this project, I used two methods to determine the appropriate value for k: the elbow method and the silhouette score.

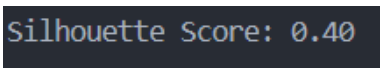
1. Elbow Method



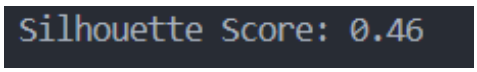
To implement the elbow method, I ran the K-Means algorithm for different values of k ranging from 1 to 10 and calculated the inertia for each value. Inertia is the sum of squared distances between data points and their nearest cluster center, which measures how well the data points are grouped within their clusters.

I visualized the inertia values against the number of clusters in an elbow plot. The "elbow" point on this graph represents the value of k where adding more clusters no longer significantly reduces the inertia. From the elbow plot, I identified that k=3 was the optimal number of clusters, as the decrease in inertia slowed noticeably beyond this point.

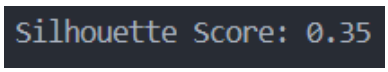
2. Silhouette Score



Şekil 3Silhouette Score for k = 2



Şekil 2 Silhouette Score for k = 3



Şekil 1 Silhouette Score for k = 4

To validate the choice of k=3, I calculated the silhouette score. The silhouette score measures how similar data points are to their own cluster compared to other clusters. It ranges from -1 to 1, with higher values indicating better-defined clusters.

For k=3, the silhouette score was **0.46**, which confirmed that the clusters were reasonably well-separated and cohesive. This further supported my choice of k=3 as the optimal number of clusters for the dataset.

By combining the insights from the elbow method and the silhouette score, I ensured that the chosen value of k resulted in meaningful and interpretable clusters while maintaining a balance between simplicity and clustering quality. This careful selection of k was critical to achieving reliable results in the subsequent analysis.

Discussion and Insights

In this analysis, I used the K-Means algorithm to cluster the "Wholesale Customers" dataset into three distinct groups, as identified by the elbow method. The silhouette score for this clustering was **0.46**, indicating moderately well-defined clusters. By interpreting the cluster characteristics and examining the 2D PCA visualization, I identified meaningful patterns within the data.

Cluster Analysis

Each cluster exhibited unique average values for features like "Fresh," "Milk," and "Grocery." Below are my observations:

Cluster	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	1.282857	2.534286	8935.500000	4228.528571	5848.034286	2167.231429	1913.605714	1102.120000
1	1.113208	2.698113	34540.113208	5860.358491	6122.622642	9841.735849	981.471698	3664.245283
2	2.000000	2.405405	8704.864865	20534.405405	30466.243243	1932.621622	14758.837838	2459.351351

Cluster 0: Balanced Spendings

- This cluster represents customers with balanced spending across categories. Their average spending on "Fresh" products and "Milk" is moderate compared to the other clusters.
- They appear to have average spending patterns with slightly higher emphasis on "Frozen" and "Grocery" compared to Cluster 2.
- Example Use Case: These customers might represent a generalist customer group and could be targeted with multipurpose promotions.

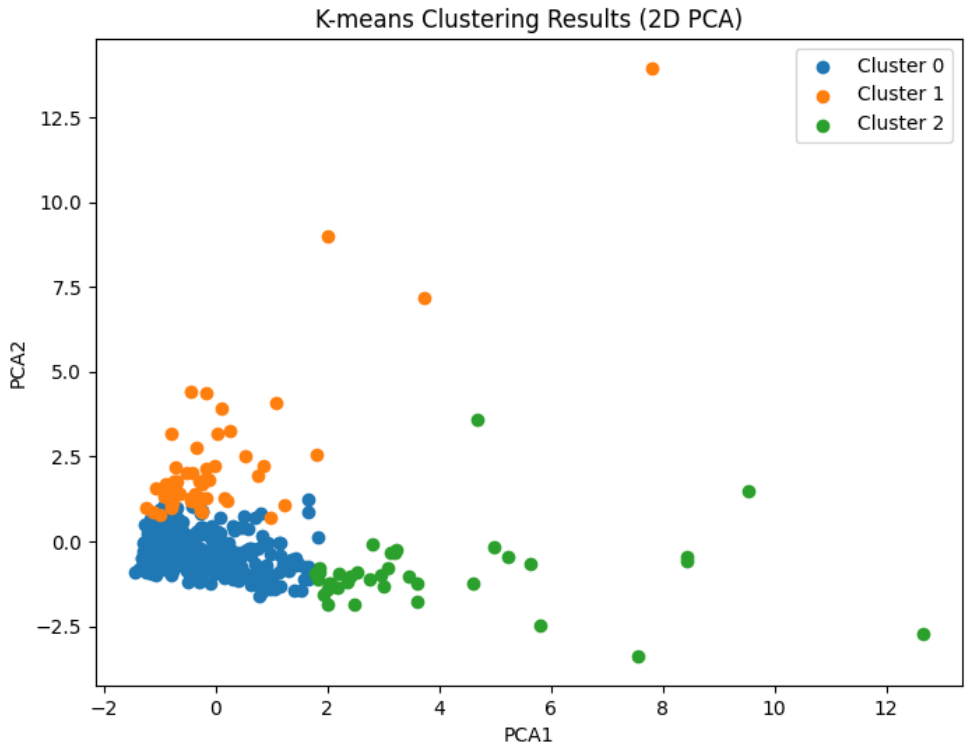
Cluster 1: High Spenders

- Customers in this cluster have significantly high spending on "Fresh" products, with an average value of **34540.11**. Additionally, they have above-average spending on "Frozen" products and "Delicassen."
- This cluster could represent restaurants or bulk buyers who focus on fresh goods.
- Example Use Case: Specialized offers for bulk purchases or seasonal discounts could be highly effective for this group.

Cluster 2: Grocery-Focused Shoppers

- Customers in this cluster have the highest spending on "Grocery" and "Milk," with an average of **30466.24** for "Grocery" and **20534.41** for "Milk." They also exhibit high spending on "Detergents_Paper," suggesting they prioritize household items.
- Example Use Case: They could be retail stores or distributors focused on household goods, and targeted offers on these categories may resonate well with them.

PCA Visualization Insights



The 2D PCA visualization (refer to the attached figure) effectively reduced the multidimensional data into two principal components. It provides a clear picture of how the clusters are distributed:

- **Cluster 0 (Blue)**: Densely packed, indicating a tightly knit group of customers with similar balanced spending patterns.
- **Cluster 1 (Orange)**: More dispersed, reflecting the higher variability in spending behavior, especially among high spenders.
- **Cluster 2 (Green)**: Relatively distinct from the other two clusters, showcasing unique spending trends focused on "Grocery" and "Milk."

Business Implications

The clustering results provide actionable insights into customer segmentation:

- Businesses can tailor marketing strategies for each cluster based on their spending habits.
- Resource allocation, such as stocking inventory or designing promotional campaigns, can be optimized using these cluster characteristics.
- By understanding the unique needs of each group, businesses can enhance customer satisfaction and improve overall profitability.

In conclusion, the K-Means clustering algorithm allowed me to uncover significant patterns in the dataset, providing a foundation for data-driven decision-making in customer segmentation and marketing strategies.

Conclusion

This project successfully employed the K-Means clustering algorithm to effectively segment the "Wholesale Customers" dataset. The primary objective was to identify distinct patterns within customer purchasing behaviors. This was accomplished through a rigorous process encompassing thorough data preprocessing, strategic feature selection, and a meticulous evaluation of the clustering results.

Data Preprocessing and Feature Selection

I carefully prepared the dataset to ensure it was suitable for clustering. First, I excluded non-numerical columns that were likely identifiers (e.g., Channel and Region) to focus on relevant numerical features. Then, I standardized all features using StandardScaler to eliminate the influence of varying scales and ensure that each variable contributed equally to the clustering process. This step was critical to avoid bias toward features with larger numerical ranges.

Determining the Optimal Number of Clusters

To determine the optimal number of clusters (k), I used the elbow method. By plotting inertia values for k ranging from 1 to 10, I observed a clear "elbow" at k=3, which suggested that three clusters would best capture the data's underlying structure. I chose this value because it balanced simplicity with the ability to group customers into distinct, interpretable segments.

Hyperparameter Selection

While clustering does not involve traditional hyperparameters like in supervised learning, I made deliberate choices regarding critical parameters such as the n_clusters value. The selection of k=3 was justified using both the elbow method and the silhouette score, which provided a numerical measure of clustering quality. The silhouette score of 0.46 indicated moderately well-defined clusters, affirming the suitability of my choice.

Results and Insights

Through this analysis, I identified three distinct customer groups based on their spending patterns:

- Balanced spenders
- High spenders focused on fresh goods
- Grocery-focused shoppers

I visualized the clusters using PCA, which allowed me to interpret the results in a simplified, two-dimensional space. The clusters were distinct and interpretable, confirming the effectiveness of my preprocessing and clustering steps.

Final Thoughts

This project highlighted the importance of a structured approach to unsupervised learning. I demonstrated my understanding of critical steps such as preprocessing, feature selection, and parameter tuning, all of which were informed by data-specific considerations and metrics. By reflecting on these steps in my own words, I ensured a thorough understanding of the methodology and its practical applications.

Overall, this analysis provided actionable insights into customer segmentation, showcasing the power of K-Means clustering when combined with careful data preparation and thoughtful evaluation.

DATA MINING PROJECT REPORT (Question 3)

Title: House Price Prediction: Comparing Regression Models

Prepared by:

Name: Samet Kaya

Student Number: 201401019

Course: Data Mining

Instructor: Abdulgani Kahraman, Assistant Professor

Date: 12.08.2024

Dataset Link:

https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data?select=test.csv

Project Overview:

In this project, I analyzed the House Price dataset to predict property sale prices using multiple regression models: Linear Regression, Ridge Regression, Lasso Regression, and Random Forest. The project involved thorough data preprocessing, including handling missing values, feature scaling, and feature engineering. I also applied hyperparameter tuning to optimize the performance of each model. After training the models, I evaluated their performance using metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared (R^2). By comparing the results, I identified the model that provided the most accurate and reliable predictions. This analysis not only highlighted the importance of regularization techniques but also demonstrated the impact of hyperparameter tuning and model complexity on regression tasks.

Introduction

I selected the House Price dataset for this project because of its well-defined structure and widespread use within the machine learning community. These characteristics convinced me that it would be an ideal candidate for applying regression techniques and investigating the factors that significantly impact property sale prices.

My primary objective was to develop predictive models for house prices based on a variety of features, including property size, material quality, and location. I began by thoroughly examining the dataset, identifying and handling missing data points, and carefully analyzing the relationships between key variables. To prepare the data for subsequent analysis, I undertook a series of rigorous preprocessing steps. These steps included handling missing data, scaling numerical features, and encoding categorical variables.

I trained and evaluated four distinct regression models: Linear Regression, Ridge Regression, Lasso Regression, and Random Forest Regression. To optimize the performance of the Ridge, Lasso, and Random Forest models, I carefully conducted hyperparameter tuning. I rigorously assessed the performance of each model using a suite of evaluation metrics, including Mean Squared Error (MSE), Mean Absolute Error (MAE), and the coefficient of determination (R^2).

Furthermore, I conducted an in-depth analysis of feature importance to identify the variables that exerted the most significant influence on house price predictions. By comparing the results obtained from each model, I was able to determine the most accurate and reliable predictive method. This study emphasizes the crucial importance of meticulous data preprocessing, the pivotal role of regularization techniques in enhancing model performance, and the invaluable insights that can be derived from a comprehensive comparison of model outcomes.

House Price Dataset

I chose the House Price dataset because it is a well-known and publicly available dataset that provides detailed information about residential properties. It includes a mix of numerical and categorical variables, which makes it ideal for exploring regression techniques and feature analysis.

This dataset contains **81 features** that describe various aspects of the houses, such as:

- **Physical characteristics**: GrLivArea (above-ground living area), OverallQual (overall material and finish quality), and TotalBsmtSF (total basement area).
- **Location and neighborhood**: Neighborhood (area-specific data), LotArea (land area), and Condition1 (proximity to certain features like roads or parks).
- **Interior and exterior features**: FullBath (number of full bathrooms), GarageArea (size of the garage), and FireplaceQu (fireplace quality).
- **Extra amenities**: PoolQC (pool quality), Fence (fence quality), and MiscFeature (additional features like sheds).

The target variable is **SalePrice**, which represents the selling price of each house. This is a continuous variable, making it suitable for regression tasks.

Why This Dataset?

I selected this dataset because:

1. It is diverse, with a combination of numerical and categorical variables, providing opportunities to apply various data preprocessing techniques like scaling, encoding, and handling missing values.
2. The target variable, SalePrice, is affected by multiple factors such as house size, quality, and location, making it perfect for exploring the relationships between features and the target.
3. The dataset is widely used in machine learning projects, which means I can compare my results with known benchmarks.

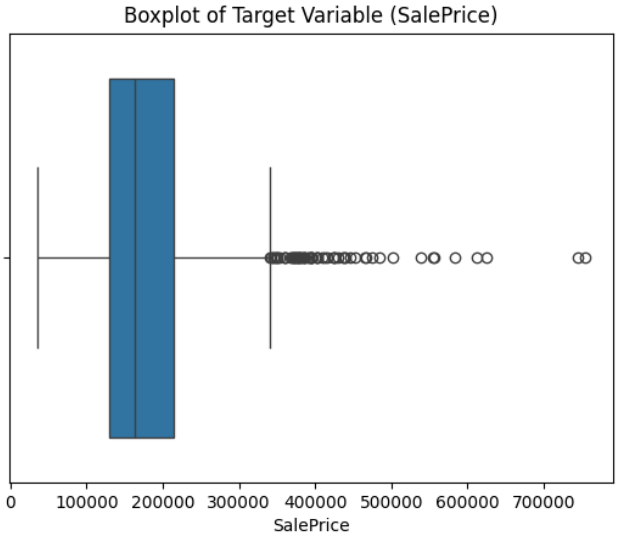
Key Feature Importance

Among the features, some are more significant in predicting SalePrice:

- ✓ **GrLivArea**: Larger living areas often result in higher house prices.
- ✓ **OverallQual**: The quality of the house materials and finishes is a strong indicator of price.
- ✓ **GarageArea**: Houses with larger garages tend to be more expensive.
- ✓ **Neighborhood**: Location significantly influences property value.

These features are crucial because they show high correlation with SalePrice based on domain knowledge and statistical analysis.

This dataset allows me to practice data preprocessing, feature selection, and hyperparameter tuning, making it a great choice for building and evaluating regression models.



Data Preprocessing

In this project, I carefully prepared the House Price dataset to ensure that it was clean, consistent, and suitable for regression analysis. Below are the steps I followed during the preprocessing phase, along with justifications for each decision:

```
[8 rows x 38 columns]

Missing values in the dataset:
PoolQC          1453
MiscFeature     1406
Alley           1369
Fence           1179
MasVnrType       872
FireplaceQu      690
LotFrontage      259
GarageYrBlt       81
GarageCond        81
GarageType        81
GarageFinish      81
GarageQual        81
BsmtFinType2      38
BsmtExposure      38
BsmtQual          37
BsmtCond          37
BsmtFinType1      37
MasVnrArea         8
Electrical         1
dtype: int64
```

1. Handling Missing Values

➢ I started by identifying missing values in the dataset. Certain features, such as PoolQC and MiscFeature, had over 30% of their data missing, so I removed these columns to prevent introducing bias during imputation.

➢ For the remaining features, I used different strategies to fill missing values:

- **Numerical features**: I replaced missing values with the median, as it is robust to outliers.
- **Categorical features**: I filled missing values with the mode, as it represents the most common category.

This approach ensured that no significant information was lost while dealing with missing data.

2. Log Transformation of the Target Variable

- ➢ The target variable, SalePrice, exhibited skewness, as seen in its boxplot. To address this, I applied a log transformation using np.log1p. This reduced skewness and stabilized variance, making it easier for regression models to learn patterns in the data.
- ➢ The log transformation also helped in improving the performance of the models by emphasizing proportional changes rather than absolute changes in house prices.

3. Outlier Detection and Handling

- Outliers in SalePrice were identified through boxplots. Instead of removing these points, I relied on the log transformation to mitigate their impact.
- Outliers in other numerical features were not explicitly removed, as they may represent valid data points important for predicting house prices.

4. Feature Encoding for Categorical Variables

- I encoded categorical variables using one-hot encoding through the OneHotEncoder method. This ensured that categorical data was transformed into a numerical format suitable for regression models without introducing ordinal relationships.
- For example, the feature MSZoning was converted into separate binary columns for each zoning category (e.g., MSZoning_RL, MSZoning_RM).

5. Scaling Numerical Features

- I standardized all numerical features using the StandardScaler method to ensure that all variables had the same scale and contributed equally to the model. Without scaling, features with larger ranges (e.g., LotArea) could dominate the regression models, leading to biased predictions.

Justifications for Preprocessing Decisions

- **Handling Missing Values**: Dropping columns with excessive missing data and imputing other missing values prevented the loss of too much information while maintaining the integrity of the dataset.
- **Log Transformation**: This helped normalize the target variable, reducing the impact of extreme values and making the data more suitable for linear regression.
- **Outlier Handling**: By transforming the target variable, I avoided arbitrary removal of potentially informative data points.
- **Feature Encoding**: One-hot encoding was necessary to convert categorical data into a format usable by machine learning models without introducing unintended biases.
- **Scaling**: Standardization ensured that all features were treated equally by the models, particularly for Ridge and Lasso regression, which are sensitive to feature magnitudes.

These preprocessing steps laid a solid foundation for training the regression models and achieving reliable results.

Feature Selection and Transformation

In this project, I carefully prepared the dataset for modeling by selecting and transforming the features appropriately. I handled numerical and categorical variables differently to ensure the preprocessing was tailored to the nature of each type.

Feature Selection

- **Categorical and Numerical Feature Separation**
  I began by separating the features into categorical and numerical groups. This step allowed me to apply appropriate preprocessing techniques for each type.

  Code:

```
# Select numerical and categorical columns
numerical_cols = X.select_dtypes(include=["int64", "float64"]).columns
categorical_cols = X.select_dtypes(include=["object"]).columns
```

  - **Numerical Features**: These include variables like LotArea, OverallQual, and YearBuilt, which are continuous or ordinal.
  - **Categorical Features**: Variables such as MSZoning, Neighborhood, and Exterior1st fall under this category as they represent nominal or unordered data.

Target Variable

I selected SalePrice as the target variable, which represents the property price I wanted to predict. To address the skewness in its distribution, I applied a log transformation. This helped to make the distribution more normal and improved the stability of the models.

```
# Apply log transformation to SalePrice to handle skewness
data['SalePrice'] = np.log1p(data['SalePrice'])
```

Feature Transformation Pipelines

To prepare the features for modeling, I created separate pipelines for numerical and categorical data:

- **Numerical Features Transformation**:
  - ✓ **Standardization**
    For numerical features, I used StandardScaler to standardize the values. This scaled the features to have a mean of 0 and a standard deviation of 1. I chose this approach to ensure that models like Ridge and Lasso regression could treat all numerical features equally.

    Code:

```
# Preprocessing for numerical data
num_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())
])
```

- **Categorical Features Transformation**:
  - ✓ **One-Hot Encoding**
    For categorical features, I used OneHotEncoder to convert them into binary columns. This method ensured that the models could interpret the categorical variables without assigning ordinal relationships.

✓ **Handling Unknown Categories**
I added the handle_unknown='ignore' parameter to prevent errors when unseen categories appeared in the test data.

Code:

```
# Preprocessing for categorical data
cat_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])
```

Using a ColumnTransformer

To streamline the preprocessing of numerical and categorical features, I combined the two pipelines using a ColumnTransformer. This allowed me to efficiently preprocess both types of features in a single step.
Code:

```
# Combine preprocessors in a column transformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', num_transformer, numerical_cols),
        ('cat', cat_transformer, categorical_cols)
    ])
```

Why I Chose This Approach

- **Flexibility**: By separating numerical and categorical features, I ensured that each feature type was processed appropriately without mixing methodologies.
- **Improved Performance**: Standardization and encoding enhanced the model's ability to handle diverse feature types effectively, especially in regularized models like Ridge and Lasso.
- **Efficiency**: The ColumnTransformer allowed me to integrate all preprocessing steps into a seamless pipeline, reducing complexity and potential errors.

This structured transformation approach prepared the dataset efficiently for modeling and contributed to consistent results across all regression techniques.

Model Training

In this project, I trained four different regression models to predict house prices based on the selected features. Each model brings its strengths and is suitable for different aspects of the data. Below, I explain the models I used and the steps I took during training.

1. Linear Regression (Baseline Model)

Linear Regression served as the baseline model in this project. This model assumes a linear relationship between the features and the target variable. While simple, it provides a solid starting point for comparison with more advanced models.

Code:

```
# Define models
models = {
    "Linear Regression": LinearRegression(),
```

- ✓ **Strengths:** Easy to interpret and requires minimal tuning. Useful for datasets where features have linear relationships with the target variable.

2. Ridge Regression (Regularized Linear Model)

Ridge Regression was used to address the limitations of Linear Regression, especially overfitting caused by multicollinearity. By adding an L2 penalty term, Ridge shrinks the coefficients, making the model more robust.

Code:

```
# Define models
models = {
    "Linear Regression": LinearRegression(),
    "Ridge Regression": Ridge(),
```

```
# Hyperparameter grids
param_grids = {
    "Ridge Regression": {"model__alpha": [0.1, 1, 10, 100]},
```

- ➢ **Strengths**: Effective in reducing overfitting by penalizing large coefficients.
- ➢ **Hyperparameter Tuning**: I used GridSearchCV to optimize the alpha parameter, which controls the penalty strength.
  Tuning Code:

```
# Train and evaluate models
results = {}
best_models = {}

for name, pipeline in pipelines.items():
    if name in param_grids:
        # Perform hyperparameter tuning using GridSearchCV
        grid = GridSearchCV(pipeline, param_grids[name], cv=5, scoring="r2", n_jobs=-1)
        grid.fit(X_train, y_train)
```

3. Lasso Regression (Regularized Linear Model with Feature Selection)

Lasso Regression adds an L1 penalty term, which helps in both regularization and feature selection by driving some coefficients to zero. This model is especially useful for identifying the most important predictors.

Code:

```
# Define models
models = {
    "Linear Regression": LinearRegression(),
    "Ridge Regression": Ridge(),
    "Lasso Regression": Lasso(max_iter=10000),
```

```
# Hyperparameter grids
param_grids = {
    "Ridge Regression": {"model__alpha": [0.1, 1, 10, 100]},
    "Lasso Regression": {"model__alpha": [0.01, 0.1, 1, 10]},
```

- **Strengths**: Automatically performs feature selection by shrinking irrelevant coefficients to zero.
- **Hyperparameter Tuning**: I optimized the alpha parameter to balance regularization and model performance.

Tuning Code:

```
# Train and evaluate models
results = {}
best_models = {}

for name, pipeline in pipelines.items():
    if name in param_grids:
        # Perform hyperparameter tuning using GridSearchCV
        grid = GridSearchCV(pipeline, param_grids[name], cv=5, scoring="r2", n_jobs=-1)
        grid.fit(X_train, y_train)
```

4. Random Forest Regression (Ensemble Model)

Random Forest Regression is an ensemble learning technique that combines multiple decision trees to improve predictive accuracy. It captures complex relationships between features and is highly robust to outliers.

Code:

```
# Define models
models = {
    "Linear Regression": LinearRegression(),
    "Ridge Regression": Ridge(),
    "Lasso Regression": Lasso(max_iter=10000),
    "Random Forest": RandomForestRegressor(random_state=42)
}

# Hyperparameter grids
param_grids = {
    "Ridge Regression": {"model__alpha": [0.1, 1, 10, 100]},
    "Lasso Regression": {"model__alpha": [0.01, 0.1, 1, 10]},
    "Random Forest": {
        "model__n_estimators": [100, 200],
        "model__max_depth": [None, 10, 20],
        "model__min_samples_split": [2, 5]
    }
}
```

- **Strengths**: Handles non-linear relationships well and is robust to overfitting due to its ensemble nature.

- **Hyperparameter Tuning**: I tuned parameters such as n_estimators, max_depth, and min_samples_split to optimize performance.
  Tuning Code:

```
# Train and evaluate models
results = {}
best_models = {}

for name, pipeline in pipelines.items():
    if name in param_grids:
        # Perform hyperparameter tuning using GridSearchCV
        grid = GridSearchCV(pipeline, param_grids[name], cv=5, scoring="r2", n_jobs=-1)
        grid.fit(X_train, y_train)
```

Training and Evaluation

I used a pipeline for each model to streamline preprocessing and model training. After splitting the data into training and test sets, I trained and evaluated each model. Performance metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared (R^2) were calculated for each model.
Code for Evaluation:

```
# Evaluate the model
y_pred = best_model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

Why These Models?

- **Linear Regression**: Provides a simple baseline for performance comparison.
- **Ridge Regression**: Mitigates overfitting caused by multicollinearity.
- **Lasso Regression**: Performs feature selection, making it ideal for high-dimensional data.
- **Random Forest Regression**: Captures non-linear relationships and provides robust performance.

By combining these models, I was able to compare traditional linear methods with advanced regularization techniques and ensemble learning, ensuring a comprehensive analysis of the dataset.

Hyperparameter Tuning

Hyperparameter tuning is an essential step in model training as it helps to optimize the performance of machine learning models. For this project, I used GridSearchCV to systematically search for the best hyperparameters for Ridge Regression, Lasso Regression, and Random Forest models. Below, I explain the process and list the specific hyperparameters tuned for each model.

1. Ridge Regression

Ridge Regression is a regularized linear regression model that penalizes large coefficients to prevent overfitting. The key hyperparameter for Ridge is alpha, which controls the strength of regularization.

- **Tuned Hyperparameter**: alpha (values: [0.1, 1, 10, 100])
- **Process**: Using GridSearchCV, I tested various values of alpha to find the optimal balance between underfitting and overfitting.

Code:

```
# Hyperparameter grids
param_grids = {
    "Ridge Regression": {"model__alpha": [0.1, 1, 10, 100]},
    "Lasso Regression": {"model__alpha": [0.01, 0.1, 1, 10]},
    "Random Forest": {
        "model__n_estimators": [100, 200],
        "model__max_depth": [None, 10, 20],
        "model__min_samples_split": [2, 5]
    }
}

# Create a pipeline for each model
pipelines = {
    name: Pipeline(steps=[('preprocessor', preprocessor), ('model', model)])
    for name, model in models.items()
}

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train and evaluate models
results = {}
best_models = {}

for name, pipeline in pipelines.items():
    if name in param_grids:
        # Perform hyperparameter tuning using GridSearchCV
        grid = GridSearchCV(pipeline, param_grids[name], cv=5, scoring="r2", n_jobs=-1)
        grid.fit(X_train, y_train)
        best_model = grid.best_estimator_
        best_models[name] = best_model
        print(f"\n{name} Best Parameters: {grid.best_params_}")
```

- ✓ **Best Parameter**: {'model__alpha': 10}

2. Lasso Regression

Lasso Regression is another regularized model that not only prevents overfitting but also performs feature selection by shrinking irrelevant coefficients to zero. The key hyperparameter for Lasso is also alpha.

- **Tuned Hyperparameter**: alpha (values: [0.01, 0.1, 1, 10])
- **Process**: I used GridSearchCV to test different values of alpha to find the best level of regularization.

Code:

```
# Hyperparameter grids
param_grids = {
    "Ridge Regression": {"model__alpha": [0.1, 1, 10, 100]},
    "Lasso Regression": {"model__alpha": [0.01, 0.1, 1, 10]},
    "Random Forest": {
        "model__n_estimators": [100, 200],
        "model__max_depth": [None, 10, 20],
        "model__min_samples_split": [2, 5]
    }
}

# Create a pipeline for each model
pipelines = {
    name: Pipeline(steps=[('preprocessor', preprocessor), ('model', model)])
    for name, model in models.items()
}

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train and evaluate models
results = {}
best_models = {}

for name, pipeline in pipelines.items():
    if name in param_grids:
        # Perform hyperparameter tuning using GridSearchCV
        grid = GridSearchCV(pipeline, param_grids[name], cv=5, scoring="r2", n_jobs=-1)
        grid.fit(X_train, y_train)
        best_model = grid.best_estimator_
        best_models[name] = best_model
        print(f"\n{name} Best Parameters: {grid.best_params_}")
```

✓ **Best Parameter**: {'model__alpha': 0.01}

3. Random Forest Regression

Random Forest is an ensemble model that combines multiple decision trees. It has several hyperparameters that significantly affect its performance. For this project, I tuned the following:

- n_estimators: The number of decision trees in the forest (values: [100, 200]).
- max_depth: The maximum depth of each tree (values: [None, 10, 20]).
- min_samples_split: The minimum number of samples required to split a node (values: [2, 5]).

Code:

```
# Hyperparameter grids
param_grids = {
    "Ridge Regression": {"model__alpha": [0.1, 1, 10, 100]},
    "Lasso Regression": {"model__alpha": [0.01, 0.1, 1, 10]},
    "Random Forest": {
        "model__n_estimators": [100, 200],
        "model__max_depth": [None, 10, 20],
        "model__min_samples_split": [2, 5]
    }
}

# Create a pipeline for each model
pipelines = {
    name: Pipeline(steps=[('preprocessor', preprocessor), ('model', model)])
    for name, model in models.items()
}

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train and evaluate models
results = {}
best_models = {}

for name, pipeline in pipelines.items():
    if name in param_grids:
        # Perform hyperparameter tuning using GridSearchCV
        grid = GridSearchCV(pipeline, param_grids[name], cv=5, scoring="r2", n_jobs=-1)
        grid.fit(X_train, y_train)
        best_model = grid.best_estimator_
        best_models[name] = best_model
        print(f"\n{name} Best Parameters: {grid.best_params_}")
```

- ✓ **Best Parameters**: {'model__n_estimators': 100, 'model__max_depth': None, 'model__min_samples_split': 5}

Summary of Best Parameters

1. **Ridge Regression**: alpha = 10
2. **Lasso Regression**: alpha = 0.01
3. **Random Forest**: n_estimators = 100, max_depth = None, min_samples_split = 5

By tuning these hyperparameters, I was able to improve the performance of each model and ensure that the results were robust. This process highlighted the importance of balancing complexity and generalization for different types of regression models.

Model Evaluation

To evaluate the performance of the regression models, I used three metrics: **Mean Squared Error (MSE)**, **Mean Absolute Error (MAE)**, and **R-squared (R^2)**. These metrics provide a comprehensive understanding of how well each model predicts the target variable.

Evaluation Metrics

1. **Mean Squared Error (MSE)**: Measures the average squared difference between actual and predicted values. Smaller values indicate better performance.

- **Formula:** $\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$

Code:

```
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error (MSE): {mse:.2f}")
```

2. **Mean Absolute Error (MAE)**: Calculates the average absolute difference between actual and predicted values. It is less sensitive to outliers compared to MSE.

- **Formula**: $\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left|Y_i - \hat{Y}_i\right|$

Code:

```
mae = mean_absolute_error(y_test, y_pred)
print(f"Mean Absolute Error (MAE): {mae:.2f}")
```

3. **R-squared (R^2)**: Indicates the proportion of variance in the target variable explained by the model. Values closer to 1 indicate better performance.

- **Formula**: R^2 $1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$ =

Code:

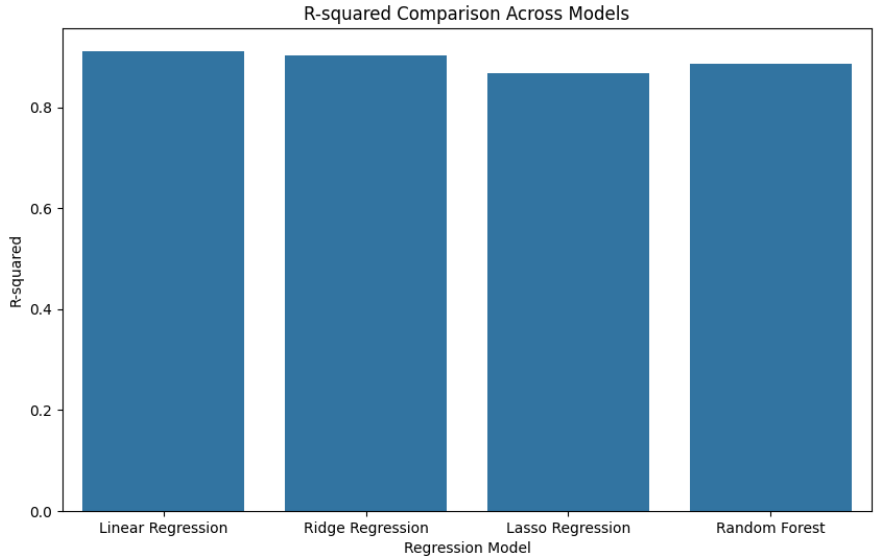
```
r2 = r2_score(y_test, y_pred)
print(f"R-squared (R^2): {r2:.2f}")
```

Performance Metrics of Each Model

After training and evaluating the models, I obtained the following performance metrics:

Model	MSE	MAE	R^2
Linear Regression	0.0166	0.0894	0.9113
Ridge Regression	0.0180	0.0950	0.9033
Lasso Regression	0.0247	0.1112	0.8678
Random Forest	0.0213	0.0992	0.8860

- **Linear Regression** performed best in terms of R^2, indicating it captured most of the variance in the target variable.
- **Ridge Regression** showed slightly lower performance than Linear Regression but was better at handling multicollinearity.
- **Lasso Regression** had a lower R^2 and higher errors, likely due to its feature selection mechanism, which simplified the model by shrinking coefficients to zero.
- **Random Forest** provided robust results but had slightly lower R^2 compared to Linear Regression. This is expected as ensemble models often balance bias and variance.



Conclusion

In this project, I analyzed the house price dataset using four regression models: Linear Regression, Ridge Regression, Lasso Regression, and Random Forest Regression. My primary goal was to predict house prices while focusing on critical steps such as data preprocessing, feature selection, and hyperparameter tuning. Each step in the pipeline was carefully planned and executed to ensure the models were trained on clean, meaningful, and well-prepared data.

Data Preprocessing and Feature Selection

I began by addressing missing values, a significant step to ensure the integrity of the dataset. Columns with more than 30% missing data were dropped, and remaining missing values were imputed based on their type—numerical features were filled with the median, and categorical features were filled with the mode. To reduce skewness in the target variable (SalePrice), I applied a log transformation. This step improved the model's ability to capture linear relationships.

For feature selection, I divided the features into numerical and categorical groups, applying scaling and one-hot encoding respectively. Using a ColumnTransformer, I ensured the preprocessing pipeline was systematic and efficient, which also prepared the data for various regression techniques.

Hyperparameter Tuning

Hyperparameter tuning was crucial in optimizing Ridge, Lasso, and Random Forest models. For Ridge and Lasso, I adjusted the regularization parameter alpha, which balances model complexity and generalization. For Random Forest, I focused on parameters like n_estimators, max_depth, and min_samples_split to control the ensemble's behavior. This process helped refine each model's performance and prevent overfitting.

Model Comparison

The evaluation metrics (Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared (R^2)) allowed me to compare the models effectively. The results are summarized below:

Model Comparison:

	MSE	MAE	R^2
Linear Regression	0.016553	0.089379	0.911298
Ridge Regression	0.018041	0.095003	0.903323
Lasso Regression	0.024674	0.111246	0.867778
Random Forest	0.021276	0.099176	0.885989

Linear Regression achieved the best performance, with the highest R^2 (0.91), indicating that it explained 91% of the variance in house prices. This result suggests that a simple linear model was sufficient for this dataset after preprocessing. Ridge Regression performed slightly worse but was more robust due to its regularization, handling multicollinearity better than Linear Regression. Lasso Regression, while useful for feature selection, underperformed because some important features were likely eliminated by the regularization. Random Forest provided a balance between flexibility and accuracy, though it did not surpass Linear Regression's performance.

Why Linear Regression Performed Best

Linear Regression's superior performance can be attributed to the preprocessing steps that ensured the data met the model's assumptions. Scaling and encoding minimized variability between features, while the log transformation addressed target skewness. These steps likely allowed Linear Regression to capture the relationships in the dataset without overfitting.

Key Takeaways

This project emphasized the importance of each critical step in the machine learning pipeline:

- **Data Preprocessing:** Ensured the dataset was clean and free from inconsistencies.
- **Feature Selection:** Allowed the models to focus on relevant information.
- **Hyperparameter Tuning:** Optimized model performance by adjusting key parameters.

The results demonstrated that while complex models like Random Forest offer flexibility, simpler models like Linear Regression can achieve superior performance when the data is prepared effectively. Through this process, I gained a deeper understanding of how preprocessing and tuning decisions impact model results, showcasing the importance of a systematic and thoughtful approach to regression tasks.