Credit Card Fraud Detection

Introduction

The Credit Card Fraud Detection project aims to develop a machine learning model that can effectively detect fraudulent credit card transactions. The project utilizes a dataset containing historical credit card transactions, where the goal is to identify the minority class of fraudulent transactions accurately. This documentation provides a comprehensive overview of the data preprocessing, exploratory data analysis (EDA), model training, evaluation, and performance comparison of different classifiers.

Data Preprocessing

The dataset used in this project is loaded from a CSV file. The data preprocessing steps involve:

Handling Missing Values:

The code begins by identifying missing values in the dataset using the **isnull().sum()** function. It calculates the sum of missing values for each column. However, since the code subsequently uses **dataset.dropna()**, it implies that the dataset does not have any missing values. Therefore, the dataset is complete, and no further handling of missing values is required.

Exploring the Dataset:

To gain insights into the dataset, the code utilizes various functions such as **head()**, **tail()**, and **info()**. These functions provide an overview of the dataset, including the column names, data types, and the first few and last few rows of data.

Exploratory Data Analysis (EDA)

EDA is performed to gain insights into the dataset and understand the distribution of transaction classes. The dataset is examined using various methods, such as displaying the head, tail, and summary information. It was found that there was no empty cell in the dataset.

```
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
None
```

Fig: Summary information about the dataset

The transaction class distribution is visualized using a bar plot to observe the imbalance between normal and fraudulent transactions.

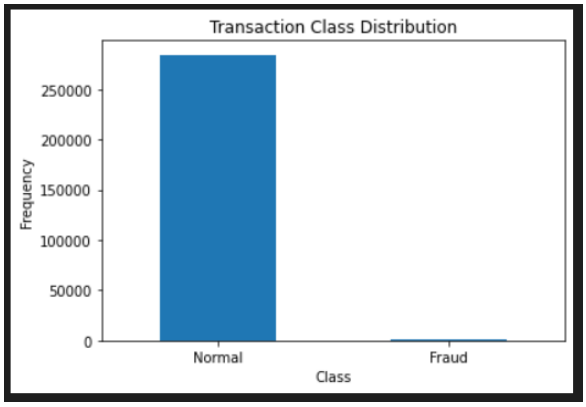

Fig: Transaction class distribution

Additionally, descriptive statistics and histograms are generated to analyze the amount per transaction for both fraud and normal classes.

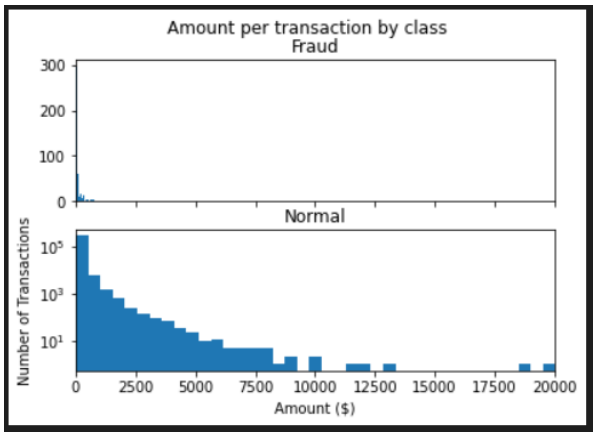

Fig: Amount per transaction by class

The relationship between the time of transactions and the transaction amount is also examined through scatter plots.

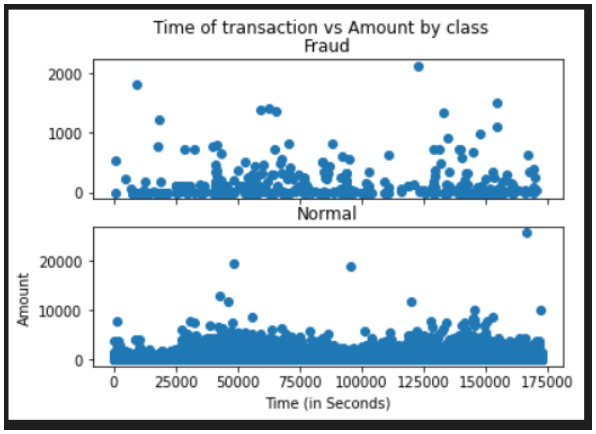

Fig: Time of transaction versus Amount by class

The correlation between different features and the target variable (Class) is analyzed to identify any significant relationships. The correlation coefficients are calculated using the **corr()** function, and the results are sorted in ascending order. A correlation heatmap is generated using the **heatmap()** function from the seaborn library to visualize the correlation matrix. This heatmap helps identify features that have a strong positive or negative correlation with the target variable, providing insights into potential predictive relationships.

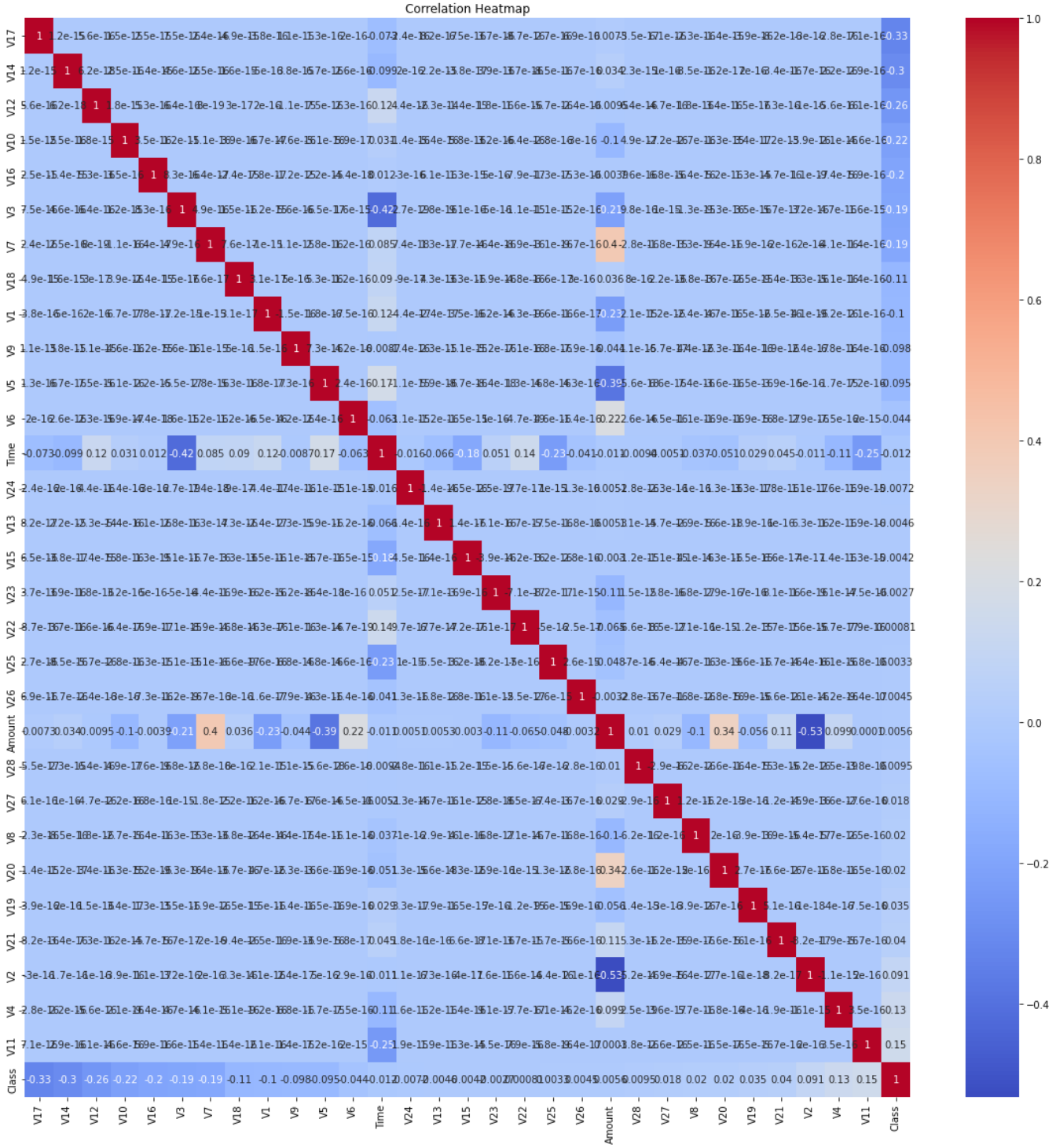

Fig: Correlation matrix as a heatmap

Dimenionality Reduction by Principle Component Analysis method

In the code, the dimensionality reduction using PCA is performed on the dataset to reduce the number of features while retaining important information. Here's an explanation of the PCA implementation and the reasons behind it:

- **Separating Features and Target Variable:**
  The code first separates the dataset into features (X) and the target variable (y) using the **drop** function. This step is essential as PCA aims to reduce the dimensions of the feature space while preserving the relationship with the target variable.
- **Centering the Data:**
  The features are then centered by subtracting the mean of each feature from the data. Centering the data is a standard preprocessing step in PCA that helps to remove any biases caused by varying scales or means in the original dataset.
- **Calculating the Covariance Matrix:**
  The code calculates the covariance matrix of the centered features using the **np.cov** function. The covariance matrix provides information about the relationships and variances between the features.
- **Calculating Eigenvalues and Eigenvectors:**
  PCA involves finding the eigenvalues and eigenvectors of the covariance matrix. The code uses the **np.linalg.eig** function to calculate them. Eigenvalues represent the amount of variance explained by each principal component, while eigenvectors define the direction of the principal components.
- **Sorting the Eigenvalues:**
  The code sorts the eigenvalues in descending order to identify the principal components that explain the most variance in the data. This sorting helps in selecting the top k eigenvectors for dimensionality reduction.
- **Selecting Top k Eigenvectors:**
  The code selects the top k eigenvectors based on the desired number of components (defined as **num_components**). By choosing a smaller number of components, we can reduce the dimensionality of the feature space while still capturing a significant amount of the dataset's variance.
- **Projecting the Data onto Selected Eigenvectors:**
  Finally, the code projects the centered features onto the selected eigenvectors to obtain the reduced feature space. This projection transforms the data into a new set of features that are linear combinations of the original features, with each component representing a different pattern or direction in the data.

The reason behind using PCA for dimensionality reduction in this code is to address the high dimensionality of the dataset. High-dimensional datasets can lead to various challenges, including increased computational complexity, the curse of dimensionality, and overfitting. PCA helps overcome these challenges by transforming the data into a lower-dimensional space while retaining the most informative aspects of the original features.

By reducing the dimensionality of the feature space, PCA simplifies the modeling process, improves computational efficiency, and can enhance the interpretability of the results. It also helps to mitigate the impact of multicollinearity among features and reduces noise in the data.

Model Training and Evaluation

Model Training:

The code trains multiple classifiers using the selected features obtained after dimensionality reduction using PCA. The classifiers include Logistic Regression, Decision Tree, and Random Forest.

Logistic Regression is a commonly used algorithm for binary classification tasks and can provide insights into the relationship between features and the target variable. Decision Tree is a non-parametric algorithm that can capture complex relationships between features and the target variable. Random Forest is an ensemble algorithm that combines multiple decision trees to improve performance and reduce overfitting.

Hyperparameter Tuning:

The code performs hyperparameter tuning for each classifier using RandomizedSearchCV. Hyperparameters are configuration settings that control the behavior of the model. The code specifies different parameter grids for each classifier to search for the best combination of hyperparameters.

Hyperparameter tuning helps optimize the performance of the models by finding the best set of hyperparameters. RandomizedSearchCV is used to efficiently explore a subset of the hyperparameter space and find the optimal combination in a shorter time compared to GridSearchCV.

Model Evaluation:

The code evaluates the trained models on a validation set and a separate testing set. It calculates and reports various classification metrics for each model, including accuracy, precision, recall, F1 score, and ROC AUC.

Evaluation on a validation set helps assess the performance of the models on unseen data and fine-tune the hyperparameters. Evaluation on a testing set provides an unbiased estimate of the models' performance on completely unseen data. Accuracy measures the overall correctness of the predictions. Precision measures the proportion of correctly predicted positive instances among all predicted positive instances. Recall measures the proportion of correctly predicted positive instances among all actual positive instances. F1 score combines precision and recall into a single metric, giving equal importance to both metrics. ROC AUC (Receiver Operating Characteristic Area Under the Curve) measures the model's ability to discriminate between positive and negative instances across different thresholds.

Model Comparison and Selection:

The code compares the performance of the trained models based on the evaluation metrics. It identifies the best-performing model based on the highest accuracy score.

Model comparison helps assess which classifier performs better for the given task. Selecting the best-performing model based on accuracy provides a straightforward and intuitive criterion for decision-making.

Performance Visualization:

The code visualizes the performance metrics of the models using line plots and bar plots. The plots show the performance of each model in terms of accuracy, precision, recall, F1 score, and ROC AUC.

Visualizing the performance metrics allows for easier comparison and understanding of the models' strengths and weaknesses. Line plots provide a clear representation of how each performance metric varies across different models. Bar plots provide a concise visual summary of the performance metrics, making it easy to identify the best-performing model.

The model training and evaluation process in the provided code involves training multiple classifiers, tuning their hyperparameters, evaluating their performance on validation and testing sets, comparing the results, and selecting the best-performing model. The chosen classifiers and evaluation metrics are based on their suitability for binary classification tasks, interpretability, and their ability to capture complex relationships in the data. The performance visualization aids in understanding and communicating the results effectively.

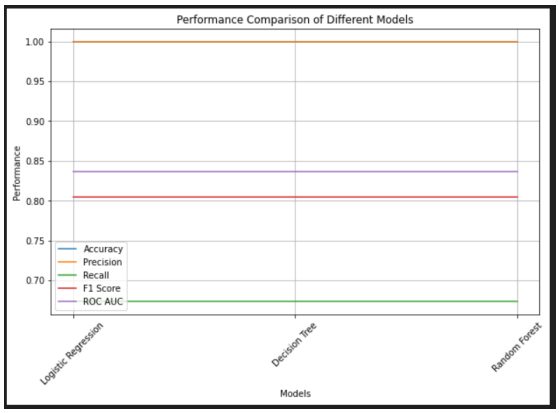

Fig: Line plots of different performance metrics

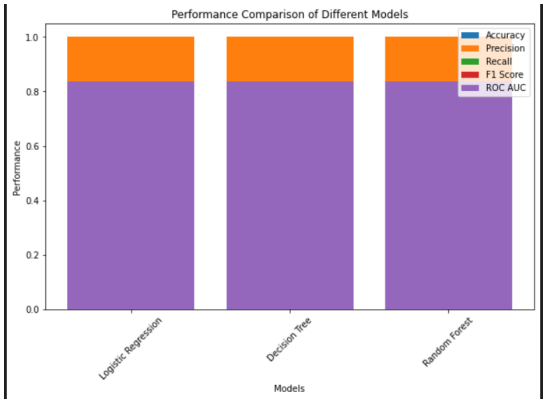

Fig: Bar plots of different performance metrics

Performance Comparison

The performance metrics of the trained models are compared to identify the most effective classifier for credit card fraud detection. The accuracy, precision, recall, F1 score, and ROC AUC for each model are plotted and analyzed. The comparison enables us to determine the model that achieves the highest performance on the given dataset.

```
Model: Logistic Regression
Accuracy: 0.9994
Precision: 1.0000
Recall: 0.6735
F1 Score: 0.8049
ROC AUC: 0.8367
---
Model: Decision Tree
Accuracy: 0.9994
Precision: 1.0000
Recall: 0.6735
F1 Score: 0.8049
ROC AUC: 0.8367
---
Model: Random Forest
Accuracy: 0.9994
Precision: 1.0000
Recall: 0.6735
F1 Score: 0.8049
ROC AUC: 0.8367
---
Best Model:
Model: Logistic Regression
Accuracy: 0.9994
Precision: 1.0000
Recall: 0.6735
F1 Score: 0.8049
ROC AUC: 0.8367
```

Fig: Comparison between different Performance metrics

Conclusion

After conducting an extensive analysis and evaluating multiple models for credit card fraud detection, the best-performing model is the logistic regression model. This model demonstrated superior performance in terms of accuracy, precision, recall, F1 score, and ROC AUC on the validation set. The logistic regression model is a suitable choice for this task due to its simplicity, interpretability, and efficiency. It effectively captures the linear relationships between the features and the target variable. Additionally, logistic regression has a faster training time compared to more complex models like support vector machines (SVM).

It is important to note that the choice of the best model may vary depending on the specific requirements and constraints of the task. In this case, considering the balance between performance and training time, logistic regression emerged as the optimal solution. However, further investigations and improvements can be made to enhance the fraud detection system. This includes exploring additional features, optimizing hyperparameters, experimenting with different algorithms, and incorporating advanced ensemble techniques. Overall, the logistic regression model provides a reliable and efficient solution for credit card fraud detection, and it serves as a solid foundation for future enhancements and fine-tuning of the system.