Machine Learning Project

Credit Card Fraud Detection

Team Members:

Name	*ID*
Ramez Adel Shakran	19100730
Mahmoud Ibrahim	19100242
Hussein Ehab	19100556

Dr. Ghada Khoriba

Eng. Amira Tarek Mahmoud

Documentation

This a Python code that performs data processing, visualization, and machine learning tasks on a dataset of credit card transactions to predict fraudulent transactions. The code includes a variety of functions and methods that enable data exploration, feature engineering, model training, and model evaluation.

The code begins by importing several libraries that are needed for data processing and visualization, including numpy, pandas, matplotlib, seaborn, and scikit-learn. It also includes a line that enables inline plotting with matplotlib.

The first task of the code is to read in the credit card dataset using the pandas function read_csv, and display the first few rows using the head method. The dataset is stored in a file called "creditcard.csv" and contains features such as transaction time, amount, and various measures of transaction activity. The code then uses the info method to display information about the data types and null values in the dataset.

Next, the code changes the data type of the 'Class' column to 'category' and renames the categories to 'Fraudulent' and 'Non_Fraudulent'. This step is necessary because the 'Class' column indicates whether a transaction is fraudulent (1) or non-fraudulent (0). Renaming the categories makes it easier to interpret the results of the machine learning models later on.

The code then calculates the percentage of fraudulent and non-fraudulent transactions in the dataset, and creates a bar plot to visualize the distribution. The code also creates scatter plots to visualize the distribution of fraudulent and non-fraudulent transactions over time and over the amount paid. These plots provide valuable insights into the characteristics of fraudulent and non-fraudulent transactions, which can help inform the development of the machine learning models.

After exploring the data, the code splits the data into training and test sets using the train_test_split function from scikit-learn. The test set is reserved for evaluating the performance of the machine learning models, while the training set is used to train the models. The code also prints the count of instances for each class in the original dataset, as well as the training and test sets, to ensure that the stratification was successful.

The code then defines a function called clf_score that takes a classifier object as input and calculates the AUC score, plots the ROC curve, and generates a classification report for the model. The function uses the predict_proba method to get the predicted probabilities for the test data, the roc_auc_score function to calculate the AUC score, and the roc_curve function to generate the ROC curve. It also uses the predict method to get the predicted labels for the test data and the classification_report function to generate a classification report.

The code then trains and evaluates a K-Nearest Neighbors (KNN) classifier using the training data, and plots the ROC curve using the clf_score function. The KNN classifier is a simple and effective algorithm that works well on datasets with a small number of features. However, it may not perform as well on datasets with a large number of features or a high degree of class imbalance.

The code also includes code that trains and evaluates a decision tree classifier using cross-validation, and plots the mean accuracy score for each tree depth to help determine the optimal tree depth. The decision tree classifier is a more powerful and flexible algorithm that can handle larger datasets and imbalanced classes, but it may be prone to overfitting if the tree is not properly pruned.

After selecting the optimal tree depth, the code trains and evaluates a decision tree classifier with a maximum depth of 5, and plots the ROC curve using the clf_score function. The code then uses the GridSearchCV function from scikit-learn to perform a grid search for the best parameters for a random forest classifier, and prints the best parameters and the AUC score. The random forest classifier is a powerful and robust algorithm that combines the predictions of multiple decision trees, and is generally more accurate and stable than individual decision trees.

Next, the code uses the SMOTE (Synthetic Minority Over-sampling Technique) algorithm from the imblearn library to re-sample the training data to balance the class distribution. SMOTE is a powerful technique that synthesizes new minority class samples using a combination of interpolation and extrapolation. It is often used to improve the performance of machine learning models on imbalanced datasets, such as fraud detection tasks where the minority class (fraudulent transactions) is much smaller than the majority class (non-fraudulent transactions).

After re-sampling the data, the code trains and evaluates KNN, decision tree, and random forest models using the re-sampled data, and plots the ROC curves for each model using the clf_score function. The code also generates classification reports for the training and test sets using the classification_report function. These results provide a more accurate and robust evaluation of the model performance, as they are based on re-sampled data that is more balanced and representative of the true class distribution.

In Conclusion, after applying SMOTE and re-running the three models, the Random Forest model was found to be the best performer.