Comparative Analysis of ML Models for Detecting Fraudulent Credit Card Transactions

Bairi Rohith Reddy (A20526972)

Bandi Brijeshkumaryadav (A20528371)

Pavitra sai vegiraju (A20525304)

April 21, 2024

Github: https://github.com/BairiRohithReddy/CSP571_Data-Preparation-and-Analysis/tree/main

Contents

1 Introduction

Credit card fraud is a significant concern for financial institutions and individuals alike, as it can lead to substantial financial losses and compromise personal information. With the increasing reliance on electronic transactions and the growing sophistication of fraudulent activities, developing effective techniques for detecting and preventing credit card fraud has become a crucial endeavor. In this report, we explore the application of machine learning techniques to detect fraudulent credit card transactions. By leveraging historical transaction data and advanced algorithms, we aim to build a robust system capableof identifying potentially fraudulent activities in real-time, thereby minimizing financial losses and safeguarding customer data.

1.1 Abstract

Credit card fraud poses a significant financial risk to both individuals and financial institutions. This report presents a comparative analysis of various machine learning models for detecting fraudulent credit card transactions. The study employs a systematic approach, including data preprocessing, feature engineering, and model training on an anonymized dataset of credit card transactions labeled as fraudulent or legitimate. Several algorithms, such as logistic regression, decision trees, gradient boosting machines, and artificial neural networks, are explored. To address the class imbalance issue prevalent in fraud detection datasets, the Synthetic Minority Over-sampling Technique (SMOTE) is applied to balance the training data. The performance of the models is evaluated using confusion matrices, which provide insights into the models' accuracy, precision, recall, and F1-scores. The impact of SMOTE on model performance is assessed by generating confusion matrices both before and after oversampling. The analysis aims to identify the most effective model or ensemble of models for accurately detecting fraudulent transactions while minimizing false positives. The comparative study contributes to the development of robust fraud detection systems, ultimately enhancing the security of electronic transactions and preventing financial losses.

2 Overview

2.1 Problem Statement

The primary objective of this study is to develop a reliable and efficient model for detecting fraudulent credit card transactions. Given a dataset containing transaction details and labels indicating whether a transaction is fraudulent or legitimate, the goal is to train a machine learning model that can accurately classify new transactions as either fraudulent or legitimate based on the provided features.

2.2 Problem Approach

To tackle this problem, we employed the power of machine learning algorithms and follow a systematic approach:

2.2.1 Data Preprocessing

The provided dataset contains anonymized features (V1-V28) resulting from a Principal Component Analysis (PCA) transformation, along with the transaction amount and a binary class label indicating whether the transaction is fraudulent or not. We performed necessary data preprocessing steps, such as handling missing values, scaling features, and splitting the data into training and testing sets.

At the same time we have implemented CRISP-DM methodology by creating a module in python to tackle the problems at hand.

2.2.2 Model Selection and Training

We will explore various machine learning algorithms, including but not limited to Logistic Regression, Decision Trees, Gradient Boosting Machines, and Artificial Neural Networks (ANNs). These models will be trained on the preprocessed dataset, and their performance will be evaluated using appropriate metrics, such as accuracy, precision, recall, and F1-score.

2.2.3 Imbalanced Data Handling

Credit card fraud datasets often suffer from class imbalance, where the number of legitimate transactions significantly outnumbers fraudulent transactions. To address this issue, we employed techniques like Synthetic Minority Over-Sampling Technique (SMOTE) resampling methods to balance the dataset and improve model performance.

2.2.4 Model Evaluation and Optimization

We evaluated the trained models using confusion matrices, which provide a visual representation of the correct and incorrect predictions made by each model. The confusion matrices will be plotted before and after applying SMOTE or other data balancing techniques, allowing us to assess the impact of these techniques on model performance.

3 Code and Explanation

3.1 Data Preprocessing and Exploratory Data Analysis

The initial steps involved in a credit card fraud detection project using machine learning techniques. It focuses on loading the necessary libraries and the credit card dataset, followed by an exploratory data analysis (EDA) phase. The EDA aims to gain insights into the data distributions and identify potential patterns or anomalies that may aid in the subsequent modeling process.

3.1.1 Libraries

The code begins by loading several libraries that will be utilized throughout the project: ranger: This library provides an implementation of the Random Forest algorithm, a popular ensemble learning method for classification and regression tasks.

1. **caret**: The caret package is a comprehensive machine learning library that facilitates the entire model trainingprocess, including data preprocessing, model tuning, and performance evaluation.

2. **Data Table:** This library offers an enhanced version of data frames, providing efficient data manipulation andsub setting capabilities.

3. **CaTools**: The ca Tools package provides utility functions for splitting data into training and test sets, amongother functionality.

4. **pROC**: This library focuses on visualizing, smoothing, and comparing receiver operating characteristic (ROC)curves, which are essential for evaluating the performance of binary classification models.

5. **rpart**: The rpart package implements recursive partitioning algorithms for decision trees, a widely used machine learning technique.

6. **rpart.plot:** This library provides functions for visualizing decision trees created using the rpart package.

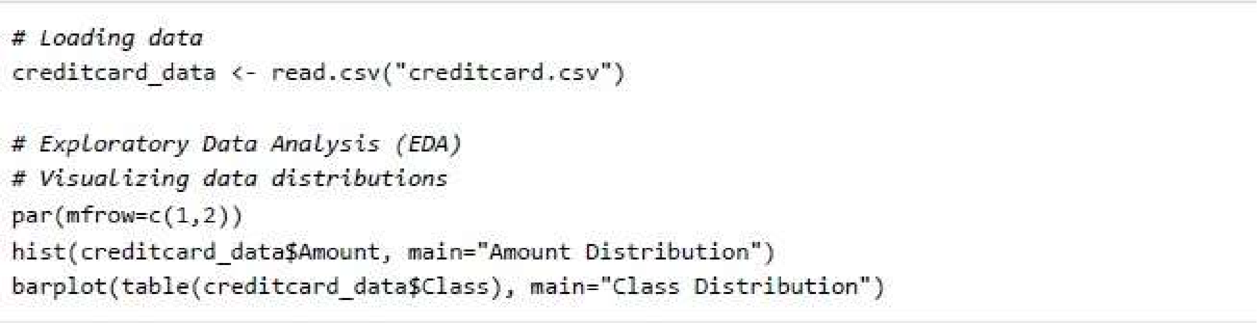
7. **neuralnet:** The neuralnet package allows for the training and evaluation of neural networks, a powerful classof machine learning models.

8. **gbm:** This library implements the Gradient Boosting Machine (GBM) algorithm, a highly effective ensemblemethod for regression and classification tasks.

9. **ROSE:** The ROSE package offers functionalities for dealing with imbalanced datasets, including oversamplingtechniques such as Synthetic Minority Over-Sampling Technique (SMOTE).

3.1.2 Data Loading

The next is to load the credit card dataset from a CSV file named” creditcard.csv”. The dataset is stored in the creditcard_data variable.



```
# Loading data
creditcard_data <- read.csv("creditcard.csv")

# Exploratory Data Analysis (EDA)
# Visualizing data distributions
par(mfrow=c(1,2))
hist(creditcard_data$Amount, main="Amount Distribution")
barplot(table(creditcard_data$Class), main="Class Distribution")
```

3.1.3 Exploratory Data Analysis (EDA)

The EDA phase aims to gain insights into the data distributions and identify potential patterns or anomalies. The code performs the following visualizations:

Amount Distribution: A histogram is plotted to visualize the distribution of the Amount feature, which likely represents the transaction amount. This visualization can help identify potential outliers or skewness in the data. Class Distribution: A bar plot is created to display the distribution of the Class feature, which is presumably the target variable indicating whether a transaction is fraudulent or legitimate. This visualization can reveal the class imbalance present in the dataset, which is a common challenge in fraud detection tasks. The visualizations are arranged side by side using the par (mfrow=c (1,2)) command for better comparison and interpretation.

Based on the insights gained from the EDA, further steps include:

1. Data Cleaning and Pre-processing: Depending on the data quality and presence of missing values or outliers, additional data cleaning and pre-processing techniques may be required.

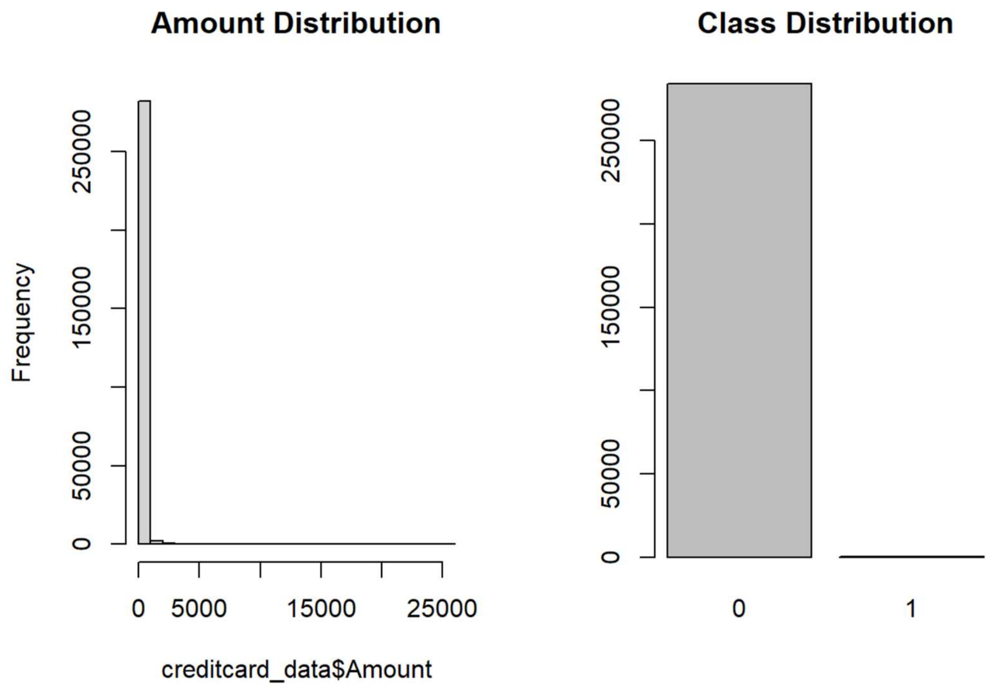
2. Feature Engineering: The dataset may contain additional features that could be engineered or transformed to improve the performance of the machine learning models.

3. Handling Class Imbalance: Given the likely class imbalance observed in the Class distribution, techniques like oversampling (e.g., SMOTE) or under sampling may be employed to address this issue.

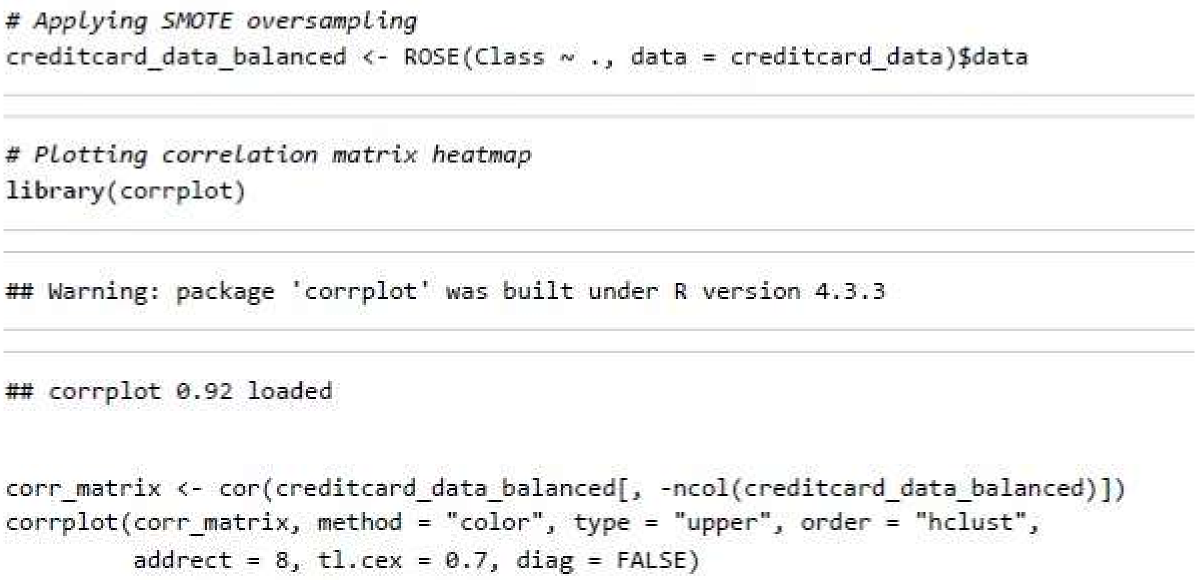
4. Model Training and Evaluation: The pre-processed data can be split into training and test sets, and various machine learning models, such as Random Forest, Decision Trees, Neural Networks, or Gradient Boosting Machines, can be trained and evaluated using appropriate performance metrics (e.g., precision, recall, F1- score, ROC-AUC).

5. Model Tuning and Optimization: Depending on the performance of the initial models, techniques like hyper-parameter tuning, ensemble methods, or feature selection may be employed to optimize the models further.

6. Model Deployment and Monitoring: Once a satisfactory model is obtained, it can be deployed into a production environment for real-time fraud detection, along with appropriate monitoring and maintenance procedures.



3.2 Handling Class Imbalance and Feature Correlation Analysis



```
# Applying SMOTE oversampling
creditcard_data_balanced <- ROSE(Class ~ ., data = creditcard_data)$data
```

```
# Plotting correlation matrix heatmap
library(corrplot)
```

```
## Warning: package 'corrplot' was built under R version 4.3.3
```

```
## corrplot 0.92 loaded
```

```
corr_matrix <- cor(creditcard_data_balanced[, -ncol(creditcard_data_balanced)])
corrplot(corr_matrix, method = "color", type = "upper", order = "hclust",
         addrect = 8, tl.cex = 0.7, diag = FALSE)
```

3.2.1 Addressing Class Imbalance with SMOTE Oversampling

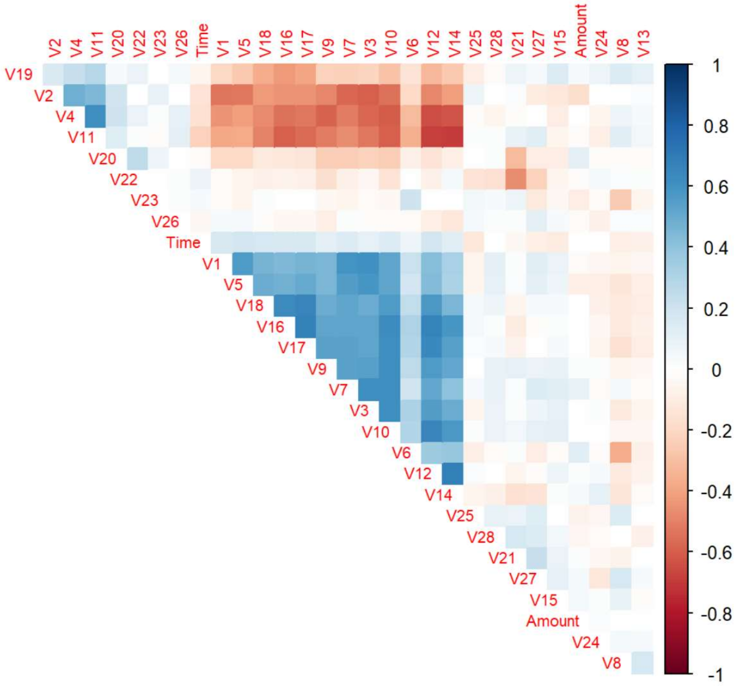
Previously we demonstrated the initial data loading and exploratory data analysis (EDA) phases. One of the observations from the EDA was the presence of class imbalance in the Class feature, which is likely the target variable indicating fraudulent or legitimate transactions. Class imbalance is a common challenge in fraud detection tasks, where the number of fraudulent transactions is typically much smaller than the legitimate ones. This imbalance can lead to biased models that perform well on the majority class but poorly on the minority class (fraudulent transactions), which is often the class of greater interest.

To address this issue, the code employs the Synthetic Minority Over-Sampling Technique (SMOTE), implemented in the ROSE package. SMOTE is a popular oversampling technique that generates synthetic instances of the minority class by interpolating between existing minority instances and their nearest neighbours'.

The ROSE function is called with the formula Class., indicating that the Class feature is the target variable, and all other features are predictors. The resulting balanced dataset is stored in the creditcard data balanced variable. By applying SMOTE oversampling, the class distribution in the dataset becomes more balanced, improving the ability of the machine learning models to learn patterns from the minority class effectively.

3.2.2 Feature Correlation Analysis

Another crucial step in the data pre-processing and feature engineering process is understanding the relationships and correlations between the features. Highly correlated features can introduce redundancy and multi collinearity, potentially affecting the performance and interpretability of the machine learning models. To visualize the feature correlations, we used the corrplot package to create a correlation matrix The cor function is used to compute the correlation matrix, excluding the last column (presumably the target variable Class). The resulting correlation matrix is then passed to the corrplot function, which generates a heat map visualization. The corrplot function has several arguments:



- method =” color”: Specifies that the correlation values should be represented using a color scale.

- type =” upper”: Indicates that only the upper triangle of the correlation matrix should be displayed.

- order =” hclust”: Specifies that the features should be ordered according to hierarchical clustering, grouping highly correlated features together.
- addrect = 8: Adds rectangles around correlated feature clusters with a correlation threshold of 0.8.

- tl.cex = 0.7: Sets the font size for the feature labels.

- diag = FALSE: Excludes the diagonal elements (correlations of features with themselves) from the plot.

- The correlation matrix heat map provides a visual representation of the feature correlations, with darker colours indicating stronger positive correlations and lighter colours representing negative correlations. Highly correlated features appear clustered together, and rectangles are drawn around clusters with correlations above the specified threshold (0.8 in this case).

This visualization helps identify potentially redundant features and can guide feature selection or engineering decisions. Features with high correlations may be candidates for removal or transformation to reduce multi collinearity and improve model performance and interpretability.

After addressing the class imbalance issue and analysing feature correlations, the next steps include:

1. Feature Selection or Engineering: Based on the correlation analysis, highly correlated features can be removed or transformed to reduce redundancy and multi collinearity.

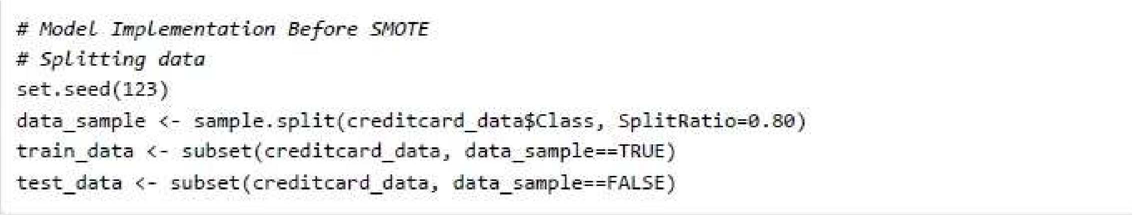
2. Data Partitioning: The balanced dataset should be partitioned into training and testing (or validation) sets for model training and evaluation.

3. Model Training and Evaluation: Various machine learning models, such as Random Forest, Decision Trees, Neural Networks, or Gradient Boosting Machines, can be trained on the balanced and pre-processed dataset. Appropriate performance metrics (e.g., precision, recall, F1-score, ROC-AUC) should be used to evaluate the models' performance on the test set.

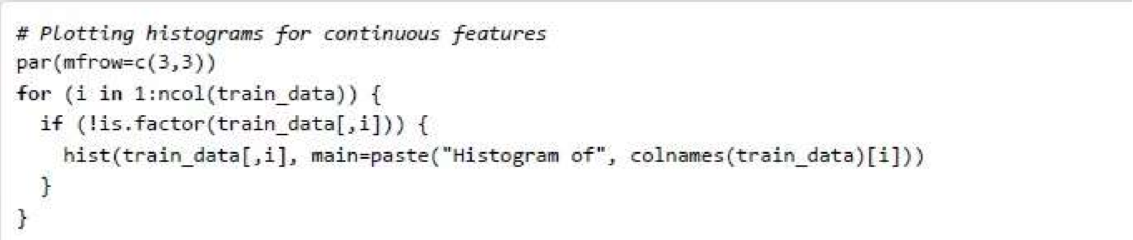
4. Model Tuning and Optimization: Techniques like hyper parameter tuning, ensemble methods, or additional feature engineering may be employed to optimize the models further and improve their performance.

5. Model Deployment and Monitoring: Once a satisfactory model is obtained, it can be deployed into a production environment for real-time fraud detection, along with appropriate monitoring and maintenance procedures.

3.3 Data Partitioning and Feature Distribution Analysis



```
# Model Implementation Before SMOTE
# Splitting data
set.seed(123)
data_sample <- sample.split(creditcard_data$Class, SplitRatio=0.80)
train_data <- subset(creditcard_data, data_sample==TRUE)
test_data <- subset(creditcard_data, data_sample==FALSE)
```



```
# Plotting histograms for continuous features
par(mfrow=c(3,3))
for (i in 1:ncol(train_data)) {
  if (!is.factor(train_data[,i])) {
    hist(train_data[,i], main=paste("Histogram of", colnames(train_data)[i]))
  }
}
```

3.3.1 Data Partitioning

Before applying the SMOTE oversampling technique, we split the original imbalanced dataset into training and testing sets. This step is crucial for evaluating the performance of machine learning models on unseen data and ensuring their generalization capabilities. The sample. Split function from the caTools package is used to create a logical vector indicating which observations should be included in the training set. The Split Ratio argument specifies that 80percent of the data should be used for training, and the remaining 20 percent for testing.

To ensure reproducibility of the results, the set Seed function is used to set a specific seed value for the random number generator. The subset function is then used to create the train data and test data datasets based on the logical vector obtained from sample. It is important to note that this data partitioning is performed before applying the SMOTE oversampling technique. The class imbalance issue will be addressed separately on the training set, as oversampling the test set could lead to overfitting and an overoptimistic evaluation of the model's performance.

3.3.2 Feature Distribution Analysis

Visualizing the distributions of the features in the training set can provide valuable insights into the data and guide further pre-processing or feature engineering decisions. The code snippet includes a loop to plot histograms for all continuous features in the training set.

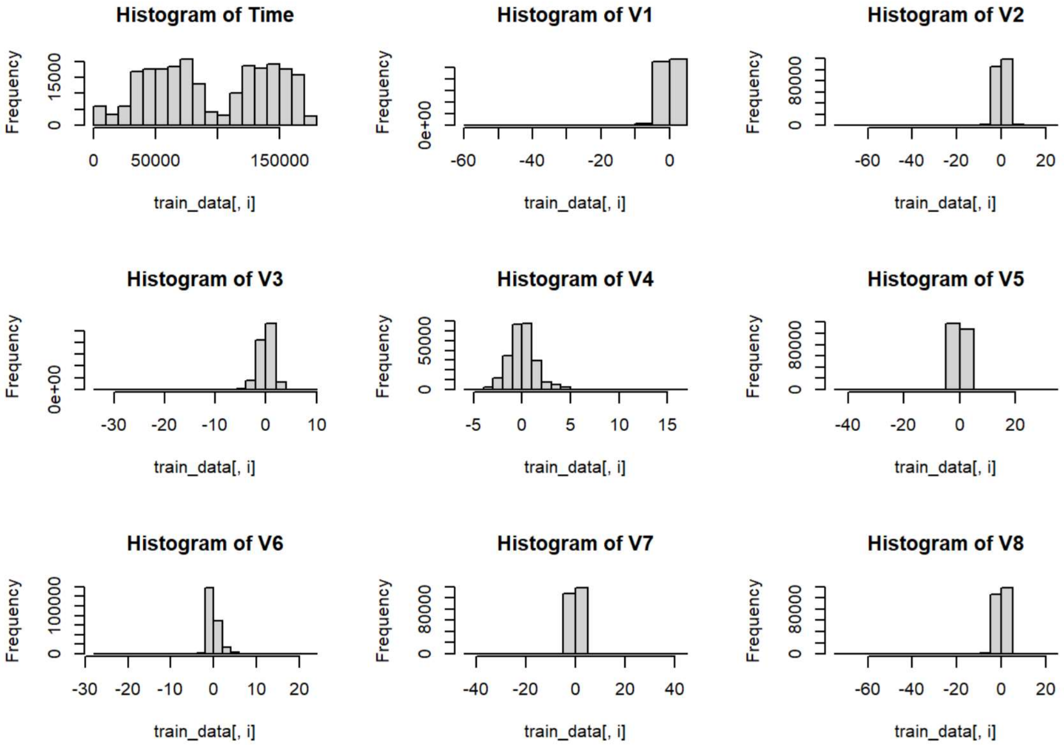
The loop iterates over all columns (features) in the train data dataset. For each column, it checks if the feature is not a factor

(i.e., Visualizing the feature distributions can reveal valuable information, such as:

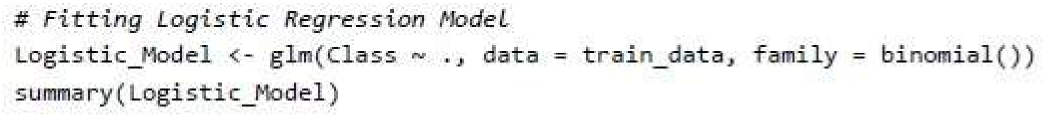
- Presence of outliers or extreme values
- Skewness or deviations from a normal distribution
- Potential multimodality or clustering of data points
- This information can guide further data pre-processing steps, such as handling outliers, applying transformations (e.g., log or Box-Cox transformations) to reduce skewness, or identifying potential feature interactions or nonlinearities.

After partitioning the data and analysing the feature distributions, the next include:

1. Applying SMOTE Oversampling: The SMOTE oversampling technique should be applied to the training set to address the class imbalance issue, as demonstrated in the previous code snippet.
2. Additional Data Pre-processing: Based on the insights gained from the feature distribution analysis, further data pre-processing steps, such as outlier handling, feature transformations, or feature scaling, may be necessary.
3. Feature Selection or Engineering: Depending on the observed feature distributions and correlations, feature selection or engineering techniques can be employed to improve the model's performance and interpretability.
4. Model Training and Evaluation: Various machine learning models can be trained on the pre-processed and balanced training set, and their performance can be evaluated on the held-out test set using appropriate metrics.
5. Model Tuning and Optimization: Techniques like hyper parameter tuning, ensemble methods, or additional feature engineering may be employed to optimize the models further and improve their performance.
6. Model Deployment and Monitoring: Once a satisfactory model is obtained, it can be deployed into a production environment for real-time fraud detection, along with appropriate monitoring and maintenance procedures.



3.4 Logistic Regression Modelling



```
# Fitting Logistic Regression Model
Logistic_Model <- glm(Class ~ ., data = train_data, family = binomial())
summary(Logistic_Model)
```

3.4.1 Fitting a Logistic Regression Model

After partitioning the data and analysing the feature distributions, we demonstrated the process of fitting a logistic regression model to the training data. Logistic regression is a widely used statistical model for binary classification problems, making it a suitable choice for the credit card fraud detection task. Logistic model (Generalized Linear Model) function from the base R package is used to fit the logistic regression model. The formula Class. specifies that the Class variable is the target (response) variable, and all other features in the train data dataset are used as predictors.

The family argument is set to binomial (), indicating that the model should use the binomial distribution, which is appropriate for binary classification problems like fraud detection, where the target variable (Class) can take one of two values (e.g., 0 for legitimate transactions and 1 for fraudulent transactions). The fitted logistic regression model is stored in the Logistic model object, and the summary function is called to display details

3.4.2Interpreting the Model Summary

The summary output of the logistic regression model provides valuable information for understanding the relationship between the predictors (features) and the target variable (Class). Here are some key components of the summary:

1. Coefficients: The summary displays the estimated coefficients for each predictor variable. These coefficients represent the change in the log-odds of the target variable for a unit increase in the corresponding predictor, holding all other predictors constant.

2. Standard Errors: The standard errors of the estimated coefficients are also reported, indicating the uncertainty associated with the coefficient estimates.

3. Z-value and Pr(>|z|): The z-value is the coefficient estimate divided by its standard error, and Pr(>|z|) is the corresponding p-value. These values help assess the statistical significance of each predictor variable in the model.

4. Null Deviance and Residual Deviance: These values measure the goodness-of-fit of the model. Lower residual deviance compared to the null deviance suggests that the model provides a better fit to the data than a null model with no predictors.

5. AIC (Akaike Information Criterion): The AIC is a measure of model quality that balances goodness-of-fit and model complexity. Smaller AIC values indicate better models.

6. By examining the coefficients, their significance levels, and the model's overall fit statistics, you can gain insights into the relationships between the predictors and the target variable, as well as the overall performance of the logistic regression model.

After fitting the logistic regression model, the next steps include:

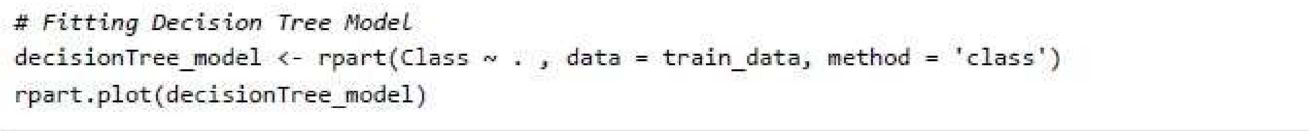
1. Model Evaluation: The performance of the logistic regression model should be evaluated on the held-out test set using appropriate performance metrics, such as accuracy, precision, recall, F1-score, and area under the receiver operating characteristic (ROC-AUC) curve.

2. Feature Importance Analysis: Techniques like calculating variable importance or examining the magnitude and sign of the coefficients can help identify the most relevant predictors for the fraud detection task.

3. Model Comparison: Alternative machine learning models, such as decision trees, random forests, or gradient boosting machines, can be trained and compared to the logistic regression model to identify the best- performing approach.
4. Model Tuning and Optimization: Regularization techniques like LASSO or ridge regression can be applied to the logistic regression model to improve its performance and reduce overfitting.

5. Model Deployment and Monitoring: Once a satisfactory model is obtained, it can be deployed into a production environment for real-time fraud detection, along with appropriate monitoring and maintenance procedures.

3.5 Visualizing Logistic Regression and Decision Tree Models



```
# Fitting Decision Tree Model
decisionTree_model <- rpart(Class ~ . , data = train_data, method = 'class')
rpart.plot(decisionTree_model)
```

3.5.1 Visualizing the Logistic Regression Model

The plot (Logistic model) issued to visualize the logistic regression model that was previously fitted to the training data. Some common plots produced by plot (Logistic model) include:

Residuals vs. Fitted Values: This plot displays the residuals (the difference between the observed and predicted values) against the fitted values. It can help identify patterns or violations of the homoscedasticity assumption (constant variance of residuals).

- Normal Q-Q Plot: This plot compares the quantiles of the residuals to the quantiles of a normal distribution. Deviations from the diagonal line may indicate a violation of the normality assumption for the residuals.

- Scale-Location Plot: This plot shows the square root of the standardized residuals against the fitted values. It can help identify potential heteroscedasticity (non-constant variance) or influential observations.

- Residuals vs. Leverage Plot: This plot displays the residuals against the leverage values, which measure the influence of each observation on the model. It can help identify outliers or influential points.

- Cook's Distance Plot: Cook's distance is a measure of the influence of each observation on the model coefficients. This plot can help identify observations that have a disproportionate influence on the model.

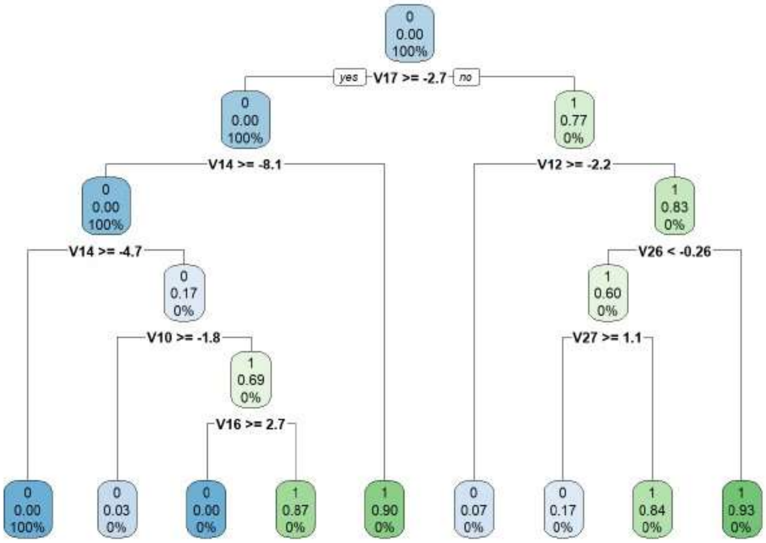
- Interpreting these diagnostic plots can provide valuable insights into the model's assumptions, potential violations, and the presence of influential observations or outliers. This information can guide further model refinement or the selection of alternative modelling approaches.

3.5.2 Fitting a Decision Tree Model

In addition to the logistic regression model, the code snippet demonstrates the process of fitting a decision tree model using the rpart (Recursive Partitioning and Regression Trees) package. The fitted decision tree model is stored in the decision Tree model object, and the rpart_plot function from the rpart_plot package is used to visualize the decision tree structure.

The resulting plot displays the decision tree, with each node representing a decision based on a specific feature, and the branches leading to the corresponding outcomes (classes). This visualization can help understand the hierarchical structure of the decision rules used by the model to make predictions.

After visualizing the logistic regression model and fitting the decision tree model, the next steps include:



1. Model Evaluation and Comparison: Both the logistic regression and decision tree models should be evaluated on the held-out test set using appropriate performance metrics, such as accuracy, precision, recall, F1-score, and area under the receiver operating characteristic (ROC-AUC) curve. The performance of these models can then be compared to identify the better-performing approach or to consider ensemble methods.

2. Decision Tree Pruning and Optimization: The decision tree model may be prone to overfitting, especially if the tree is too complex. Techniques like pruning or setting complexity parameters can help optimize the decision tree's performance and prevent overfitting.

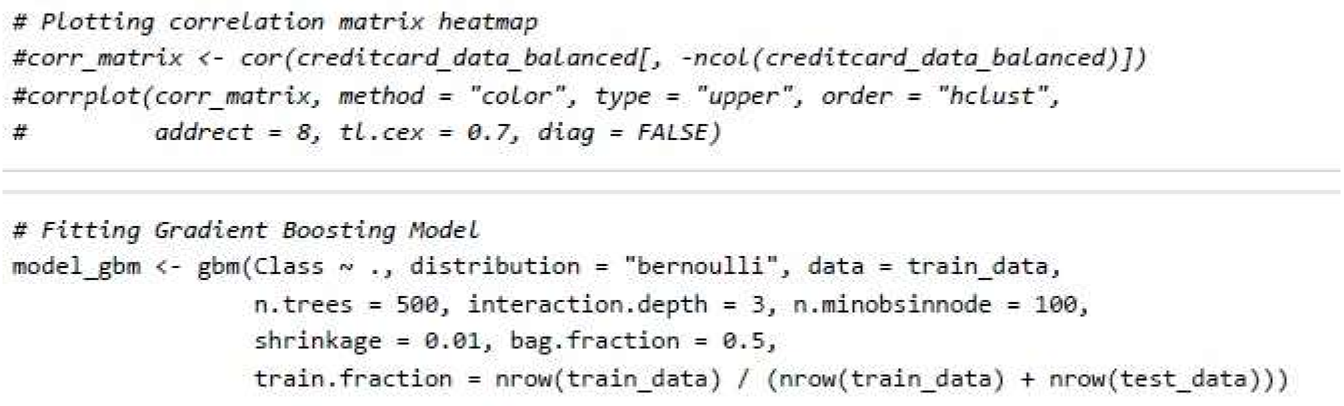
3. Feature Importance Analysis: For both models, techniques like calculating variable importance or examining the magnitude and sign of the coefficients (for logistic regression) can help identify the most relevant predictors for the fraud detection task.

4. Model Tuning and Optimization: Regularization techniques like LASSO or ridge regression can be applied to the logistic regression model, while hyper parameter tuning can be performed for the decision tree model to improve their performance and reduce overfitting.

5. Ensemble Methods: Ensemble methods like random forests or gradient boosting machines can be explored to combine multiple models and potentially improve the overall predictive performance.

6. Model Deployment and Monitoring: Once a satisfactory model (or ensemble) is obtained, it can be deployed into a production environment for real-time fraud detection, along with appropriate monitoring and maintenance procedures.

3.6 Gradient Boosting Model for Credit Card Fraud Detection



```
# Plotting correlation matrix heatmap
#corr_matrix <- cor(creditcard_data_balanced[, -ncol(creditcard_data_balanced)])
#corrplot(corr_matrix, method = "color", type = "upper", order = "hclust",
#         addrect = 8, tl.cex = 0.7, diag = FALSE)
```

```
# Fitting Gradient Boosting Model
model_gbm <- gbm(Class ~ ., distribution = "bernoulli", data = train_data,
                 n.trees = 500, interaction.depth = 3, n.minobsinnode = 100,
                 shrinkage = 0.01, bag.fraction = 0.5,
                 train.fraction = nrow(train_data) / (nrow(train_data) + nrow(test_data)))
```

3.6.2 Fitting a Gradient Boosting Model

The code snippet demonstrates the process of fitting a Gradient Boosting Machine (GBM) model using the gbm package. Gradient boosting is a powerful ensemble learning technique that combines multiple weak models (decision trees) to create a strong predictive model The gbm function is used to fit the Gradient Boosting Model, with the following parameters:

- Class.: The formula specifies that the Class variable is the target (response) variable, and all other features in the train data dataset is used as predictors distribution =" Bernoulli": The Bernoulli distribution issued, which is appropriate

- Data = train data: The training data is provided as input. n. trees = 500: The number of decision trees (base learners) to be included in

- Interaction_Depth = 3: This parameter controls the maximum depth of the individual decision trees, with a value of 3 allowing for up to three-way interactions between features.

- n.minobsinnode = 100: This parameter specifies the minimum number of observations required in each terminal node of the decision trees.

- shrinkage = 0.01: The shrinkage or learning rate parameter controls the contribution of each tree to the final model, with smaller values (like 0.01) leading to a slower but more robust learning process.

- Bag_fraction = 0.5: This parameter specifies the fraction of the training data to be used for fitting each individual decision tree, with a value of 0.5 indicating that 50

- Train_fraction: This parameter determines the fraction of the combined training and test data to be used for training the model. In this case, it is set to the ratio of the number of observations in the training set to the total number of observations.

- The fitted Gradient Boosting Model is stored in the model_bmobject.

After fitting the Gradient Boosting Model, the next steps include:

1. Model Evaluation and Comparison: The Gradient Boosting Model should be evaluated on the held-out test set using appropriate performance metrics, such as accuracy, precision, recall, F1-score, and area under the receiver operating characteristic (ROC-AUC) curve. Its performance can then be compared to the previously fitted models (e.g., logistic regression, decision trees) to identify the most effective approach.

2. Hyper parameter Tuning: The performance of the Gradient Boosting Model can be further optimized by tuning its hyper parameters, such as the number of trees, interaction depth, learning rate, and other parameters. Techniques like grid search or random search can be employed to find the optimal combination of hyper parameters.

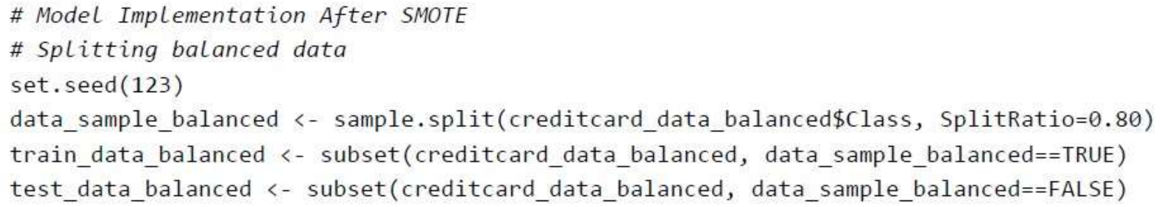
3. Feature Importance Analysis: Gradient Boosting Models provide a measure of feature importance, which can help identify the most relevant predictors for the fraud detection task. This information can guide feature selection or engineering efforts.

4. Ensemble Methods: The Gradient Boosting Model can be combined with other models, such as logistic regression or decision trees, using ensemble techniques like stacking or blending. Ensemble methods can often improve predictive performance by leveraging the strengths of multiple models.

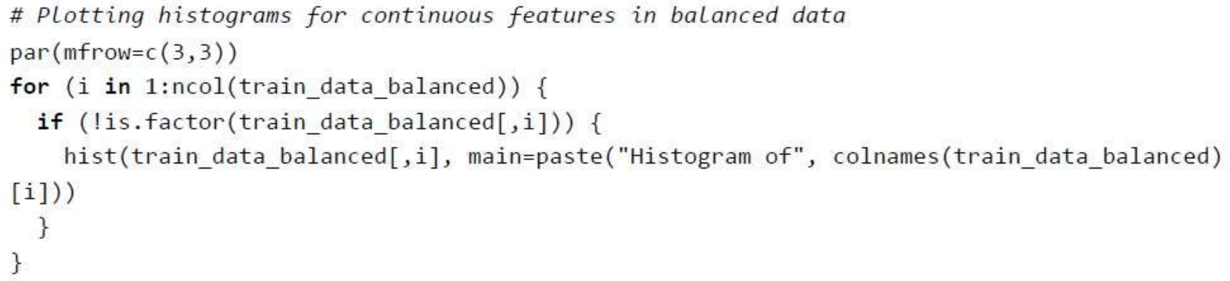
5. Model Interpretation and Explain ability: While Gradient Boosting Models are powerful predictive models, they can be complex and difficult to interpret. Techniques like Shapley Additive Explanations (SHAP) or partial dependence plots can be used to gain insights into the model's decision-making process and the impact of individual features on the predictions.

6. Model Deployment and Monitoring: Once a satisfactory model (or ensemble) is obtained, it can be deployed into a production environment for real-time fraud detection, along with appropriate monitoring and mainte- nance procedures.

3.7 Data Partitioning and Feature Distribution Analysis on Balanced Data



```
# Model Implementation After SMOTE
# Splitting balanced data
set.seed(123)
data_sample_balanced <- sample.split(creditcard_data_balanced$Class, SplitRatio=0.80)
train_data_balanced <- subset(creditcard_data_balanced, data_sample_balanced==TRUE)
test_data_balanced <- subset(creditcard_data_balanced, data_sample_balanced==FALSE)
```



```
# Plotting histograms for continuous features in balanced data
par(mfrow=c(3,3))
for (i in 1:ncol(train_data_balanced)) {
  if (!is.factor(train_data_balanced[,i])) {
    hist(train_data_balanced[,i], main=paste("Histogram of", colnames(train_data_balanced)
[i]))
  }
}
```

3.7.1 Data Partitioning on Balanced Data

After applying the SMOTE oversampling technique to balance the class distribution in the dataset, the process of partitioning the balanced data into training and testing sets. The sample. Split function is used to create a logical vector indicating which observations should be included in the training set, with a Split Ratio of 0.80, meaning 80 To ensure reproducibility of the results, the set. Seed function is used to set a specific seed value for the random number generator. The subset function is then used to create the train data balanced and test data balanced datasets based on the logical vector obtained It's important to note that this data partitioning is performed on the balanced dataset (creditcard_data_ balanced)

3.7.2 Feature Distribution Analysis on Balanced Data

Similar to the previous feature distribution analysis performed on the original training data, includes a loop to plot histograms for all continuous features in the balanced training set (train_data_balanced). Plotting histograms for continuous features in balanced data par (mfrow=c (3,3)) for (i in 1: ncol(train_data_balanced)) if (! is factor (train_data The par (mfrow=c (3,3)) command sets the plot layout to a 3x3 grid, allowing for multiple histograms to be displayed on the same plot. The loop iterates over all columns (features) in the train data balanced dataset. For each column, it checks if the feature is not a factor Visualizing the feature distributions in the balanced training set can reveal insights into the impact of the SMOTE oversampling technique on the data. It can help identify potential changes in the distribution shapes, presence of new outliers or extreme values, or alterations in the feature relationships due to the synthetic instances generated by SMOTE.

3.7.3 Next Steps

After partitioning the balanced data and analysing the feature distributions, the next steps include:

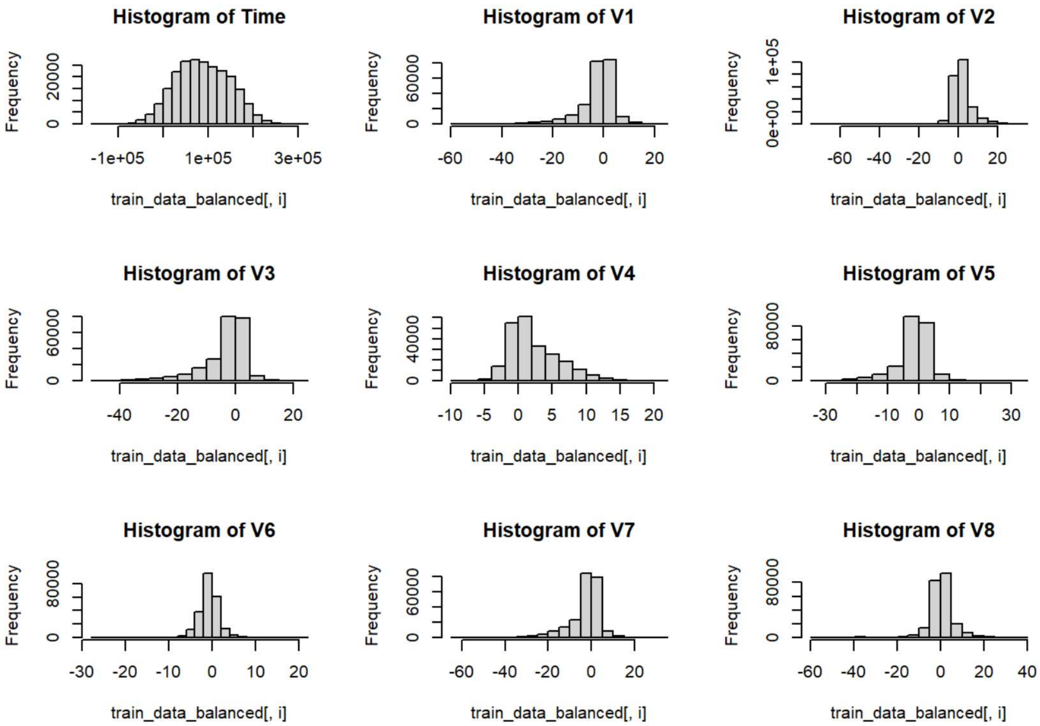
1. Model Training and Evaluation on Balanced Data: Various machine learning models (e.g., logistic regression, decision trees, random forests, gradient boosting machines) can be trained on the balanced training set (train_data_balanced) and evaluated on the balanced test set (test_data_balanced). This approach will provide a more realistic assess The performance of models trained on the balanced data can be compared to the performance of models trained on the original

2. Alternative Sampling Techniques: While SMOTE is a popular oversampling technique, other sampling methods, such as under sampling or a combination of over- and under sampling, can be explored to handle the class imbalance issue.

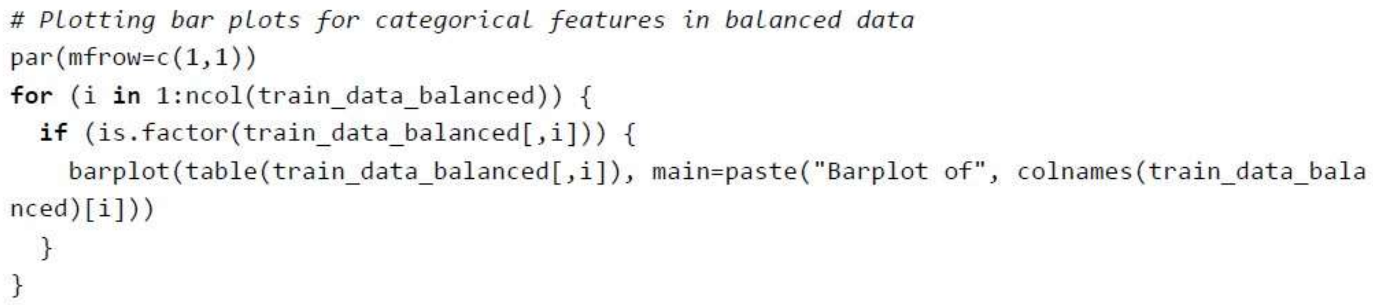
3. Model Tuning and Optimization: Techniques like hyper parameter tuning, ensemble methods, or additional feature engineering may be employed to optimize the models further and improve their performance on the balanced data.

4. Model Interpretation and Explain ability: Techniques like feature importance analysis, partial dependence plots, or Shapley Additive Explanations (SHAP) can be used to gain insights into the models' decision-making process and the impact of individual features on predictions, particularly in the context of the balanced data.

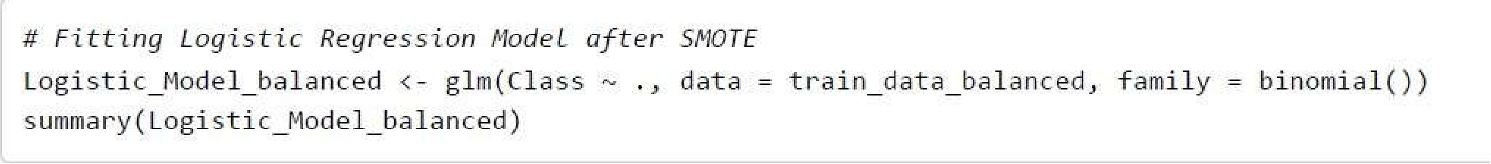
5. Model Deployment and Monitoring: Once a satisfactory model (or ensemble) is obtained, it can be deployed into a production environment for real-time fraud detection, along with appropriate monitoring and maintenance procedures.



3.8 Analysing Categorical Features and Fitting Logistic Regression Model on Balanced Data



```
# Plotting bar plots for categorical features in balanced data
par(mfrow=c(1,1))
for (i in 1:ncol(train_data_balanced)) {
  if (is.factor(train_data_balanced[,i])) {
    barplot(table(train_data_balanced[,i]), main=paste("Barplot of", colnames(train_data_bala
nced)[i]))
  }
}
```



```
# Fitting Logistic Regression Model after SMOTE
Logistic_Model_balanced <- glm(Class ~ ., data = train_data_balanced, family = binomial())
summary(Logistic_Model_balanced)
```

3.8.1 Visualizing Categorical Feature Distributions on Balanced Data

In addition to analysing the distributions of continuous features, it is also important to examine the distributions of categorical features in the balanced training data. The loop iterates over all columns (features) in the train_data_balanced dataset. For each column, it checks if the feature is a factor Visualizing the categorical feature distributions in the balanced data can reveal insights into the impact of the SMOTE oversampling technique on these features. It can help identify potential changes in the category proportions, the presence of new or rare categories, or the need for additional feature encoding or transformation techniques.

3.8.2 Fitting a Logistic Regression Model on Balanced Data

After analysing the feature distributions in the balanced data, the code snippet demonstrates the process of fitting a logistic regression model to the balanced training set (train_data_balanced). Fitting Logistic Regression Model after The glm (Generalized Linear Model) function is used to fit the logistic regression model, with the formula Class specifying that the Class variable is the target (response) variable, and all other features in the train_data_balanced dataset are used as predictors.

After analysing the categorical feature distributions and fitting the logistic regression model on the balanced data, the next steps include:

1. Model Evaluation and Comparison: The performance of the logistic regression model fitted on the balanced data (Logistic_model_balanced) should be evaluated using appropriate performance metrics (e.g., accuracy, precision, recall, F score, ROC−AUC) on the balanced test set .Its performance can then be compared to the logistic regression model fitted on the or Based on the insights gained from the categorical featured is distributions, additional feature

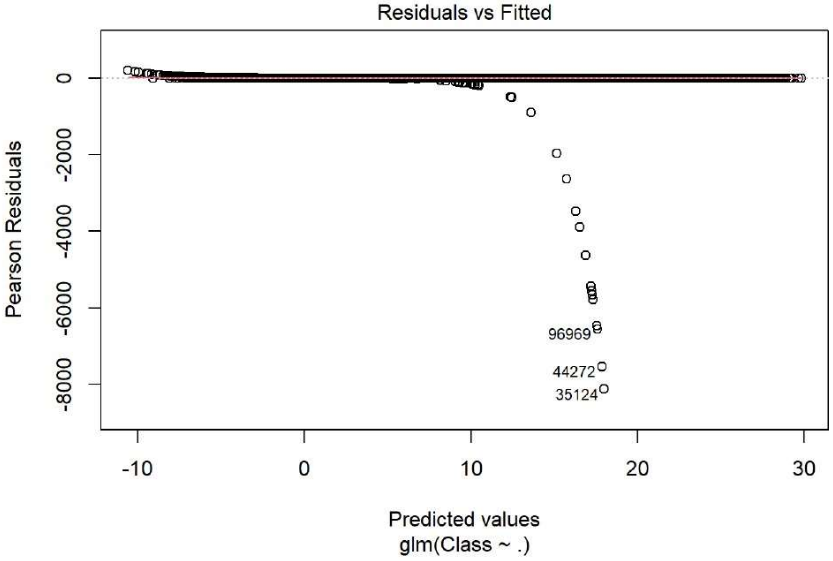
engineering techniques, such as on hot encoding, target encoding, or other encoding methods, may be employed to improve the representation and handling of categories.

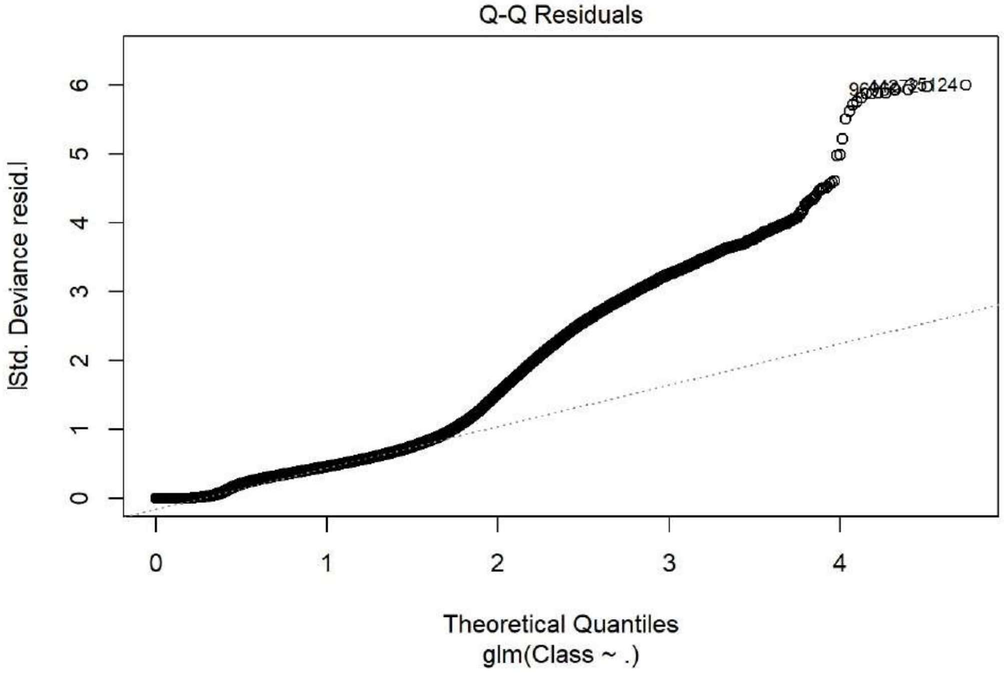
2. Model Interpretation and Feature Importance Analysis: Techniques like examining the coefficients and their significance levels, as well as calculating variable importance scores, can provide insights into the most relevant predictors for the fraud detection task in the context of the balanced data.

3. Alternative Modelling Approaches: In addition to logistic regression, other machine learning models, such as decision trees, random forests, gradient boosting machines, or neural networks, can be trained and evaluated on the balanced data to compare their performance and potentially leverage ensemble methods.

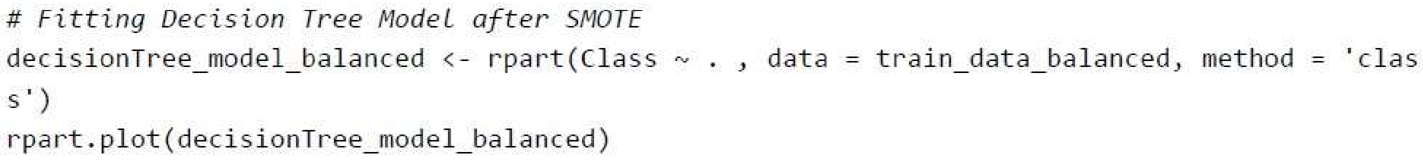
4. Hyper parameter Tuning and Optimization: Techniques like grid search or random search can be employed to tune the hyper parameters of the logistic regression model (e.g., regularization parameters) or other machine learning models to optimize their performance on the balanced data.

5. Model Deployment and Monitoring: Once a satisfactory model (or ensemble) is obtained, it can be deployed into a production environment for real-time fraud detection, along with appropriate monitoring and maintenance procedures.





3.9 Fitting a Decision Tree Model on Balanced Data



```
# Fitting Decision Tree Model after SMOTE
decisionTree_model_balanced <- rpart(Class ~ . , data = train_data_balanced, method = 'clas
s')
rpart.plot(decisionTree_model_balanced)
```

3.9.1 Decision Tree Modelling on Balanced Data

After fitting the logistic regression model on the balanced data, the process of fitting a decision tree model using the rpart (Recursive Partitioning and Regression Trees) package The fitted decision tree model is stored in the decision Tree_model_balanced object, and the rpart.plot function from the rpart.plot The resulting plot displays the decision tree, with each node representing a decision based on a specific feature, and the branches leading to the corresponding outcomes (classes). This visualization can help understand the hierarchical structure of the decision rules used by the model to make predictions on the balanced data.

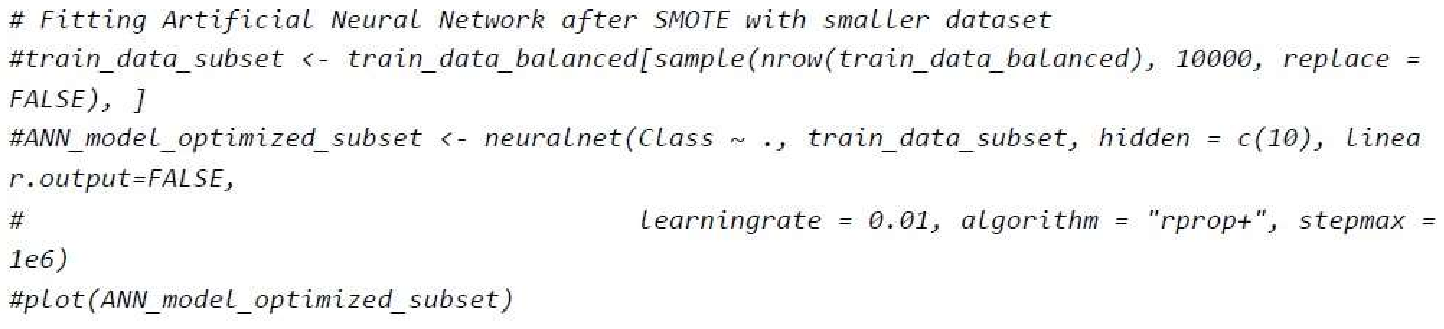
3.9.2 Importance of Training on Balanced Data

Training machine learning models, such as decision trees, on balanced data is crucial for addressing the class imbalance issue commonly encountered in fraud detection tasks. By applying techniques like SMOTE oversampling, the class distribution in the training data becomes more balanced, enabling the models to learn patterns from both the majority (legitimate transactions) and minority (fraudulent transactions) classes effectively. When trained on imbalanced data, decision tree models (and other machine learning algorithms) can become biased towards the majority class, leading to poor performance in detecting the minority class (fraudulent trans- actions). This issue can result in a high number of false negatives, where fraudulent transactions are misclassified as legitimate, potentially causing significant financial losses.

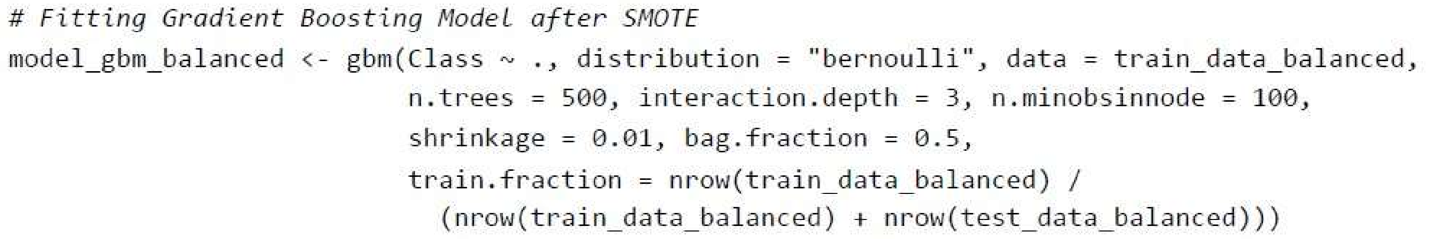
After fitting the decision tree model on the balanced data, the next steps include:

1. Feature Importance Analysis: Decision tree models provide a measure of feature importance, which can help identify the most relevant predictors for the fraud detection task in the context of the balanced data. This information can guide feature selection or engineering efforts.
2. Ensemble Methods: The decision tree model trained on the balanced data can be combined with other models using ensemble techniques like random forests, gradient boosting machines, or stacking. Ensemble methods can often improve predictive performance by leveraging the strengths of multiple models.
3. Model Interpretation and Explain ability: Decision tree models are generally more interpretable than some other machine learning models, such as neural networks or ensemble methods. Techniques like visualizing the decision rules or extracting decision paths can provide insights into the model’s decision-making process and the impact of individual features on predictions, particularly in the context of the balanced data.
4. Model Deployment and Monitoring: Once a satisfactory model (or ensemble) is obtained, it can be deployed into a production environment for real-time fraud detection, along with appropriate monitoring and maintenance procedures.

3.10 Fitting Gradient Boosting Model and Artificial Neural Network on Balanced Data



```
# Fitting Artificial Neural Network after SMOTE with smaller dataset
#train_data_subset <- train_data_balanced[sample(nrow(train_data_balanced), 10000, replace =
FALSE), ]
#ANN_model_optimized_subset <- neuralnet(Class ~ ., train_data_subset, hidden = c(10), linea
r.output=FALSE,
#                                        learningrate = 0.01, algorithm = "rprop+", stepmax =
1e6)
#plot(ANN_model_optimized_subset)
```



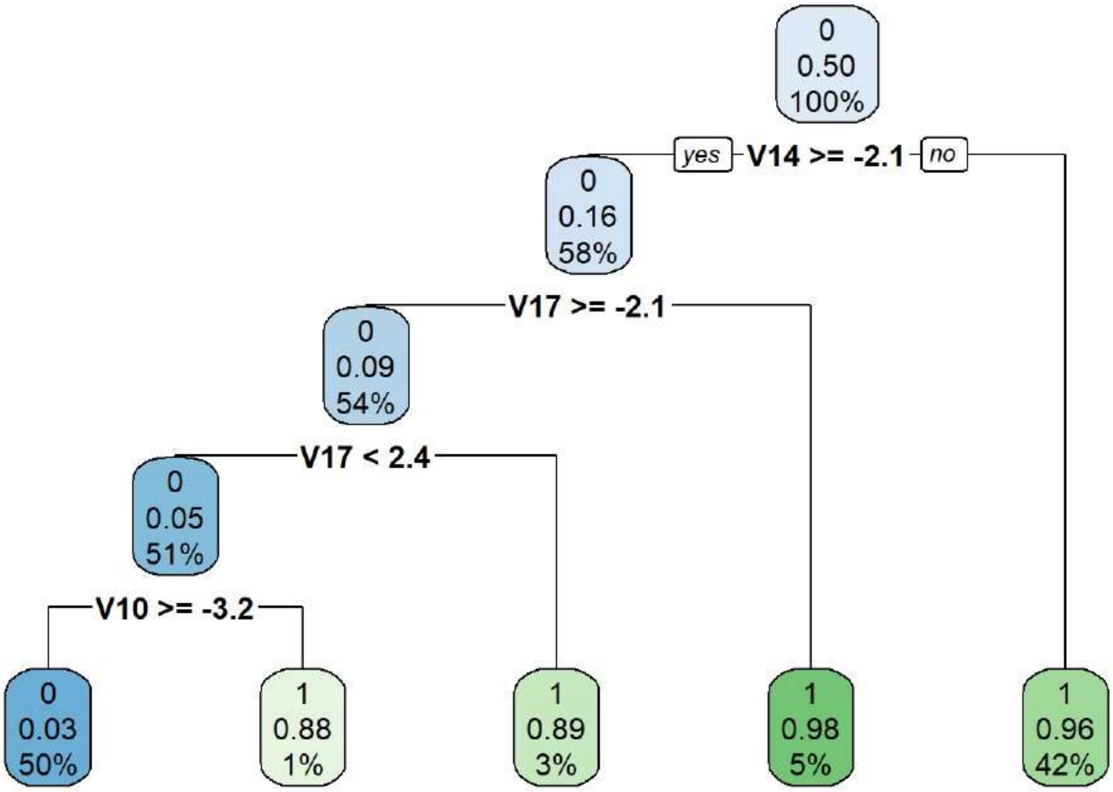
```
# Fitting Gradient Boosting Model after SMOTE
model_gbm_balanced <- gbm(Class ~ ., distribution = "bernoulli", data = train_data_balanced,
                          n.trees = 500, interaction.depth = 3, n.minobsinnode = 100,
                          shrinkage = 0.01, bag.fraction = 0.5,
                          train.fraction = nrow(train_data_balanced) /
                            (nrow(train_data_balanced) + nrow(test_data_balanced)))
```

3.10.1 Fitting a Gradient Boosting Model on Balanced Data

The gbm function is used to fit the Gradient Boosting Model, with the following parameters:

- Class.: The formula specifies that the Class variable is the target (response) variable, and all other features in the train_data_balanced dataset are used as predictors distribution =" bernoulli":
- data = train_data_balanced: The balanced training data is provided as input.n. trees = 500:
- interaction.depth = 3: This parameter controls the maximum depth of the individual decision trees, with a value of 3 allowing for up to three-way interactions between features.
- n.minobsinnode = 100: This parameter specifies the minimum number of observations required in each terminal node of the decision trees.
- shrinkage = 0.01: The shrinkage or learning rate parameter controls the contribution of each tree to the final model, with smaller values (like 0.01) leading to a slower but more robust learning process.
- bag.fraction = 0.5: This parameter specifies the fraction of the training data to be used for fitting each individual decision tree, with a value of 0.5 indicating that 50
- train_ fraction: This parameter determines the fraction of the combined training and test data to be used for training the model. In this case, it is set to the ratio of the number of observations in the balanced training set to the total number of observations in both the balanced training and test sets.
- The fitted Gradient Boosting Model is stored in the model_gbm balanced object.

After fitting the Gradient Boosting Model on the balanced data the next steps include:



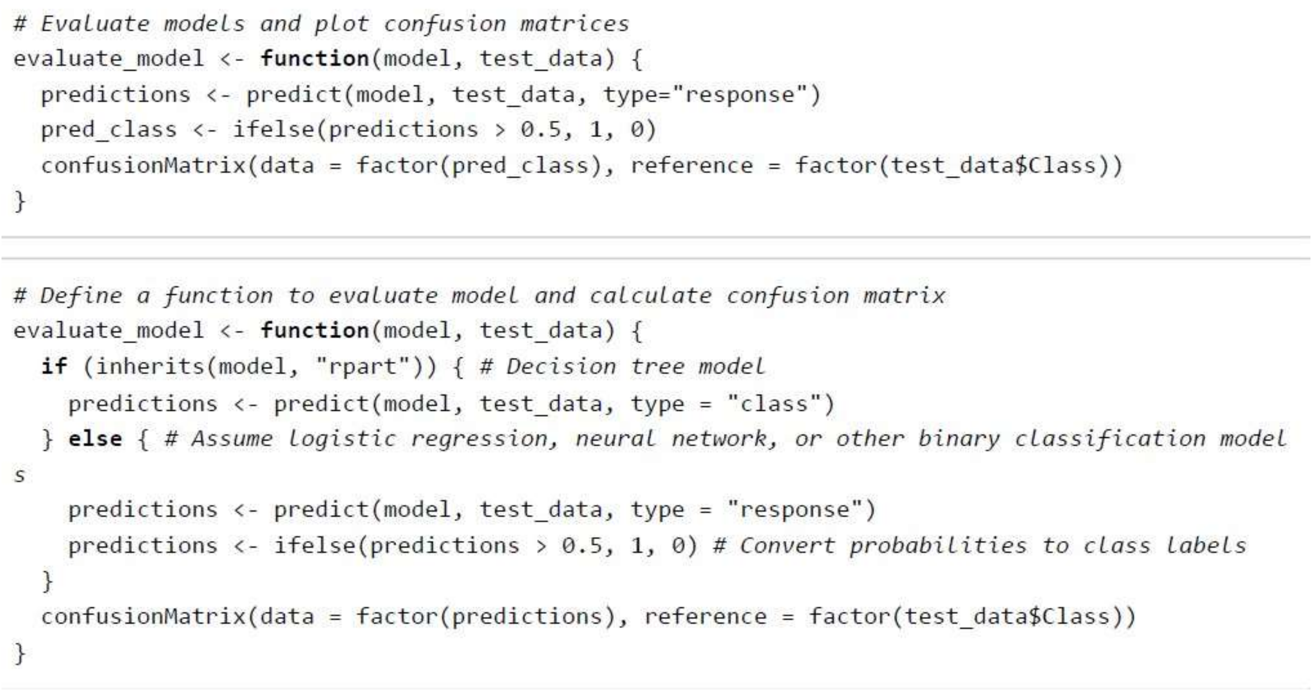
1. Model Evaluation and Comparison: The performance of the Gradient Boosting Model (model_gbm_balanced) and the Artificial score, ROC−AUC) on the balanced test set. Their performance can then be compared to other machine learning models trained on Both the Gradient Boosting Model have several hyper parameters that can be tuned too.
2. Feature Importance Analysis: Gradient Boosting Models and Neural Networks provide measures of feature importance, which can help identify the most relevant predictors for the fraud detection task in the context of the balanced data. This information can guide feature selection or engineering efforts.

3. Ensemble Methods: The Gradient Boosting Model and the Artificial Neural Network model can be combined with other models using ensemble techniques like stacking or blending. Ensemble methods can often improve predictive performance by leveraging the strengths of multiple models.

4. Model Interpretation and Explain ability: While Gradient Boosting Models are powerful predictive models, they can be complex and difficult to interpret. Techniques like Shapley Additive Explanations (SHAP) or partial dependence plots can be used to gain insights into the model's decision-making process and the impact of individual features on predictions. For Neural Networks, techniques like saliency maps or concept activation vectors can be explored to improve interpretability.

5. Model Deployment and Monitoring: Once a satisfactory model (or ensemble) is obtained, it can be deployed into a production environment for real-time fraud detection, along with appropriate monitoring and maintenance procedures.

3.11 Model Evaluation and Confusion Matrix Generation



```
# Evaluate models and plot confusion matrices
evaluate_model <- function(model, test_data) {
  predictions <- predict(model, test_data, type="response")
  pred_class <- ifelse(predictions > 0.5, 1, 0)
  confusionMatrix(data = factor(pred_class), reference = factor(test_data$Class))
}
```

```
# Define a function to evaluate model and calculate confusion matrix
evaluate_model <- function(model, test_data) {
  if (inherits(model, "rpart")) { # Decision tree model
    predictions <- predict(model, test_data, type = "class")
  } else { # Assume logistic regression, neural network, or other binary classification model
s
    predictions <- predict(model, test_data, type = "response")
    predictions <- ifelse(predictions > 0.5, 1, 0) # Convert probabilities to class labels
  }
  confusionMatrix(data = factor(predictions), reference = factor(test_data$Class))
}
```

3.11.1 Defining a Function to Evaluate Models and Calculate Confusion Matrices

After obtaining the class predictions, the confusion Matrix function from the caret package is called to generate the confusion matrix. The confusion Matrix function takes two arguments:

- data: A factor vector of the predicted class labels.
- reference: A factor vector of the true class labels from the test dataset.
- The confusion matrix provides valuable information about the model’s performance, including the true pos- itive, false positive, true negative, and false negative counts, as well as various performance metrics such as accuracy, precision, recall, and F1-score.

3.12 Interpreting Confusion Matrices

The confusion matrix provides a comprehensive overview of a model's performance by summarizing the correct and incorrect predictions for each class. In the context of credit card fraud detection, the confusion matrix can be interpreted as follows:

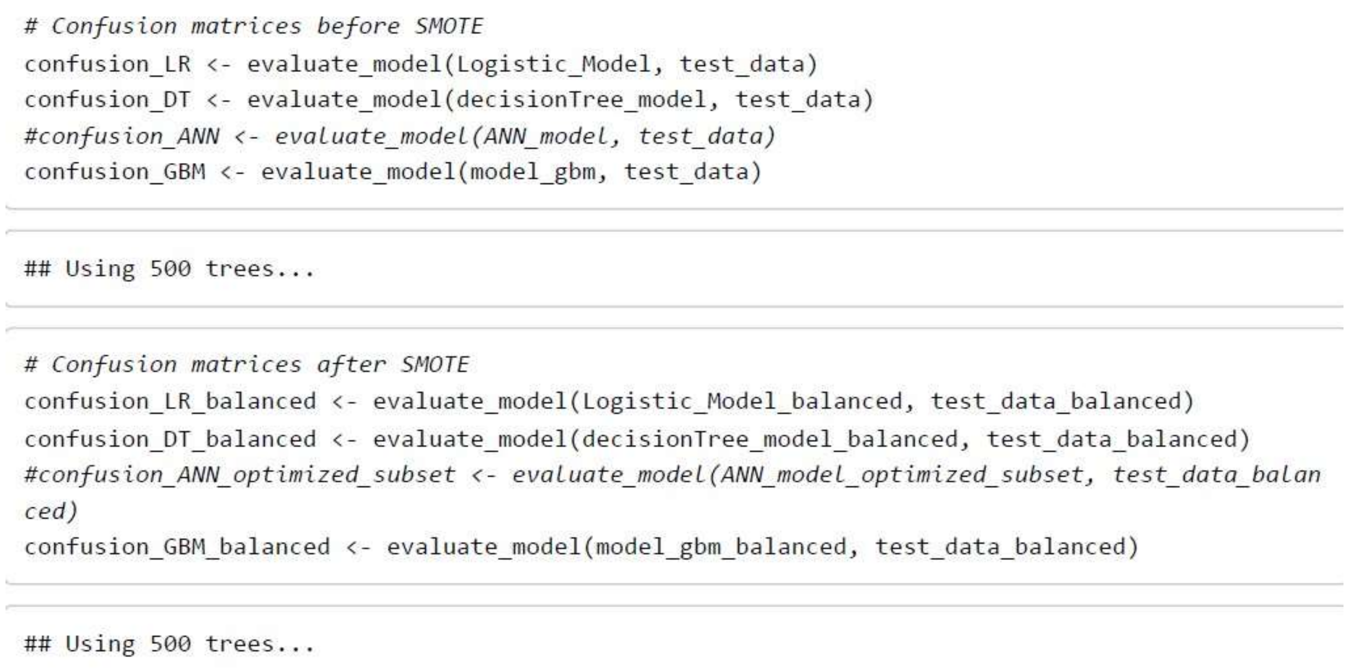
- True Positives (TP): The number of fraudulent transactions correctly identified as fraudulent.
- False Positives (FP): The number of legitimate transactions incorrectly identified as fraudulent.
- True Negatives (TN): The number of legitimate transactions correctly identified as legitimate.
- False Negatives (FN): The number of fraudulent transactions incorrectly identified as legitimate.

Based on these values, various performance metrics can be calculated:

- Accuracy: The overall proportion of correct predictions.
- Precision: The proportion of true positives among all positive predictions.
- Recall (Sensitivity): The proportion of true positives among all actual positive instances.
- F1-score: The harmonic mean of precision and recall, providing a balanced measure of performance.

By analysing the confusion matrix and these performance metrics, you can assess the model's ability to correctly identify fraudulent transactions (true positives) while minimizing the number of false positives (legitimate transactions incorrectly flagged as fraudulent) and false negatives (fraudulent transactions missed by the model). The evaluate model function provides a convenient way to evaluate and compare the performance of different machine learnin gmb.

3.13 Model Evaluation and Comparison Using Confusion Matrices



```
# Confusion matrices before SMOTE
confusion_LR <- evaluate_model(Logistic_Model, test_data)
confusion_DT <- evaluate_model(decisionTree_model, test_data)
#confusion_ANN <- evaluate_model(ANN_model, test_data)
confusion_GBM <- evaluate_model(model_gbm, test_data)
```

```
## Using 500 trees...
```

```
# Confusion matrices after SMOTE
confusion_LR_balanced <- evaluate_model(Logistic_Model_balanced, test_data_balanced)
confusion_DT_balanced <- evaluate_model(decisionTree_model_balanced, test_data_balanced)
#confusion_ANN_optimized_subset <- evaluate_model(ANN_model_optimized_subset, test_data_balan
ced)
confusion_GBM_balanced <- evaluate_model(model_gbm_balanced, test_data_balanced)
```

```
## Using 500 trees...
```

3.13.1 Generating Confusion Matrices Before SMOTE Oversampling

Model_gbm: A gradient boosting machine model trained on the imbalanced data. By generating confusion matrices for these models before applying SMOTE oversampling, you can assess their performance on the imbalanced dataset and identify potential issues related to class imbalance, such as a bias towards the majority class (legitimate transactions) or poor performance in detecting the minority class (fraudulent transactions).

3.13.2 Generating Confusion Matrices After SMOTE Oversampling

A gradient boosting machine model trained on the balanced data after SMOTE oversampling.

3.13.3 Comparing Model Performance

With the confusion matrices generated for each model, both before and after SMOTE oversampling, you can compare their performance and evaluate the effectiveness of the oversampling technique. Some key aspects to consider in the comparison include:

1. True Positive Rate (Recall): Evaluate the models' ability to correctly identify fraudulent transactions (true positives) and compare the recall scores before and after SMOTE oversampling. Ideally, the recall should improve after oversampling, indicating better detection of the minority class.
2. False Positive Rate: Evaluate the models' tendency to incorrectly flag legitimate transactions as fraudulent (false positives) and compare the false positive rates before and after oversampling. Ideally, the false positive rate should

remain reasonably low, as misclassifying legitimate transactions can lead to customer dissatisfaction and additional operational costs.

3. Precision and F1-score: Evaluate the models' overall precision (the proportion of true positives among all positive predictions) and F1-score (the harmonic mean of precision and recall) before and after oversampling. These metrics provide a balanced assessment of the models' performance, considering both the true positive and false positive rates.
4. Accuracy: While accuracy is a commonly used metric, it should be interpreted with caution in imbalanced datasets, as a high accuracy can be achieved by simply predicting the majority class correctly. Nevertheless, comparing the accuracy before and after oversampling can provide insights into the models' overall performance.

By analysing and comparing the confusion matrices and the associated performance metrics, you can identify the models that benefit the most from the SMOTE oversampling technique and select the most effective model or ensemble of models for the credit card fraud detection task.

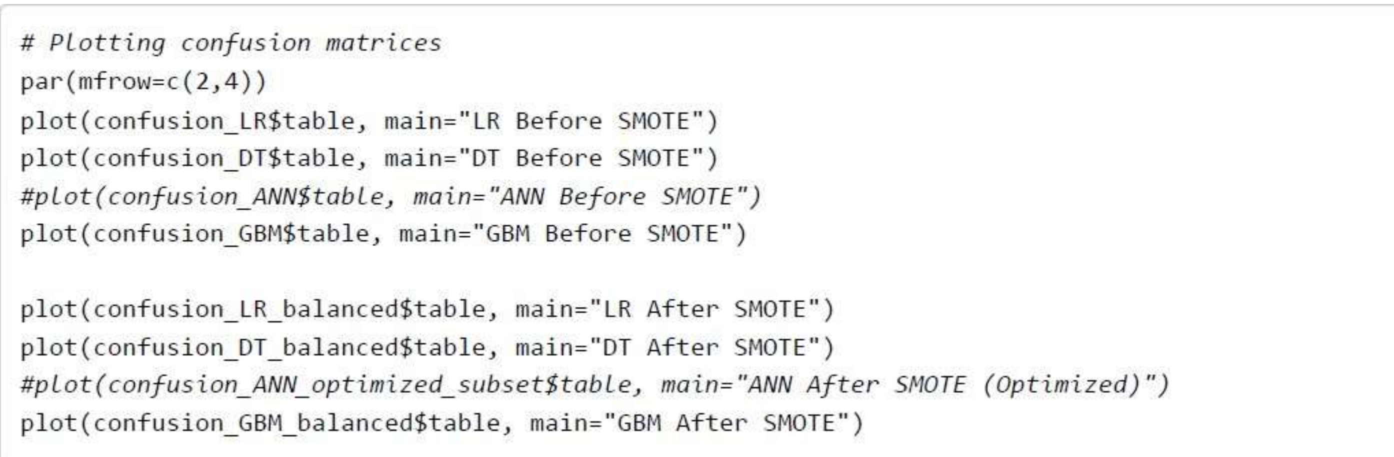
3.14 Plotting confusion matrices

Confusion matrices are a powerful tool for evaluating the performance of classification models in machine learning tasks. They provide a visual representation of the number of correct and incorrect predictions made by the model, allowing you to assess the model's accuracy, precision, recall, and other relevant metrics.

3.14.1 Code Explanation

By setting up a grid layout using the par(mfrow=c(2,4)) function, which creates a 2x4 grid of plots. This arrangement allows for the display of multiple confusion matrices in a compact and organized manner.

Next, to plot the confusion matrices for different classification models before and after applying the SMOTE (Synthetic Minority Over-Sampling Technique) technique. SMOTE is a popular method used to address imbalanced datasets by generating synthetic instances of the minority class.



```
# Plotting confusion matrices
par(mfrow=c(2,4))
plot(confusion_LR$table, main="LR Before SMOTE")
plot(confusion_DT$table, main="DT Before SMOTE")
#plot(confusion_ANN$table, main="ANN Before SMOTE")
plot(confusion_GBM$table, main="GBM Before SMOTE")

plot(confusion_LR_balanced$table, main="LR After SMOTE")
plot(confusion_DT_balanced$table, main="DT After SMOTE")
#plot(confusion_ANN_optimized_subset$table, main="ANN After SMOTE (Optimized)")
plot(confusion_GBM_balanced$table, main="GBM After SMOTE")
```

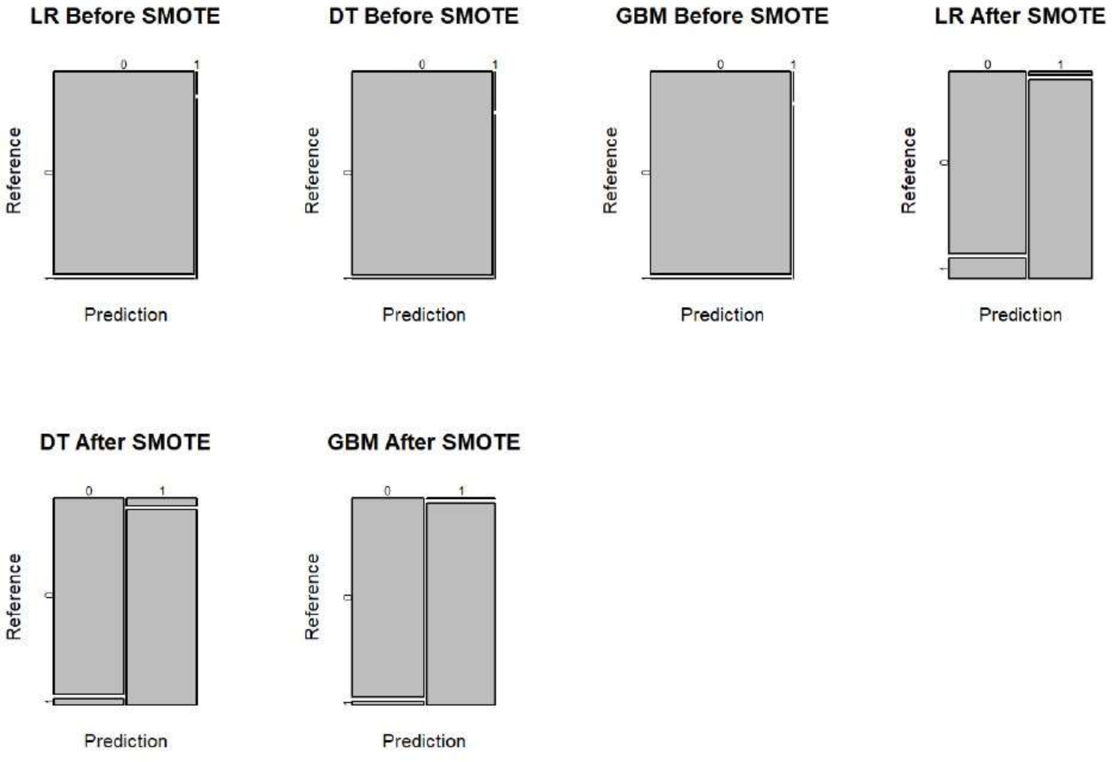
3.14.2 Interpretation and Analysis

By plotting the confusion matrices before and after applying SMOTE, we can visually compare the performance of the classification models on the original imbalanced dataset and the synthetically balanced dataset. This comparison can provide valuable insights into how the models handle class imbalance and how the SMOTE technique affects their performance.

The confusion matrices display the true positive, true negative, false positive, and false negative counts, which can be used to calculate various performance metrics, such as accuracy, precision, recall, and F1-score. These metrics can help us evaluate the models' ability to correctly classify instances from different classes and identify potential biases or limitations.

Additionally, by analysing the patterns in the confusion matrices, we can gain a deeper understanding of the types of errors the models are making and potentially identify areas for improvement or further investigation.

This the use of confusion matrices for evaluating the performance of classification models in machine learning tasks. By plotting the confusion matrices before and after applying SMOTE, we can assess the impact of the data balancing technique on the models' performance and make informed decisions regarding model selection and optimization.

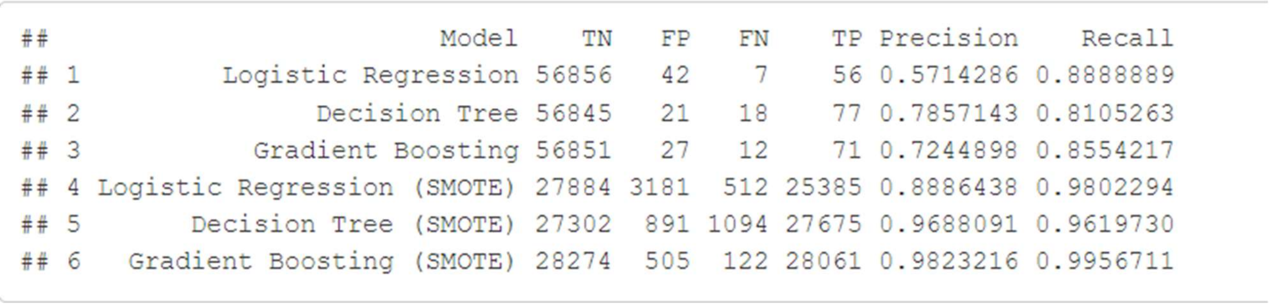


3.15 Performance Metrics

In the context of binary classification, several performance metrics are used to evaluate the effectiveness of a model. The following metrics have been calculated and reported for the different algorithms:

True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN)

These are the basic counts of correct and incorrect predictions made by the model:



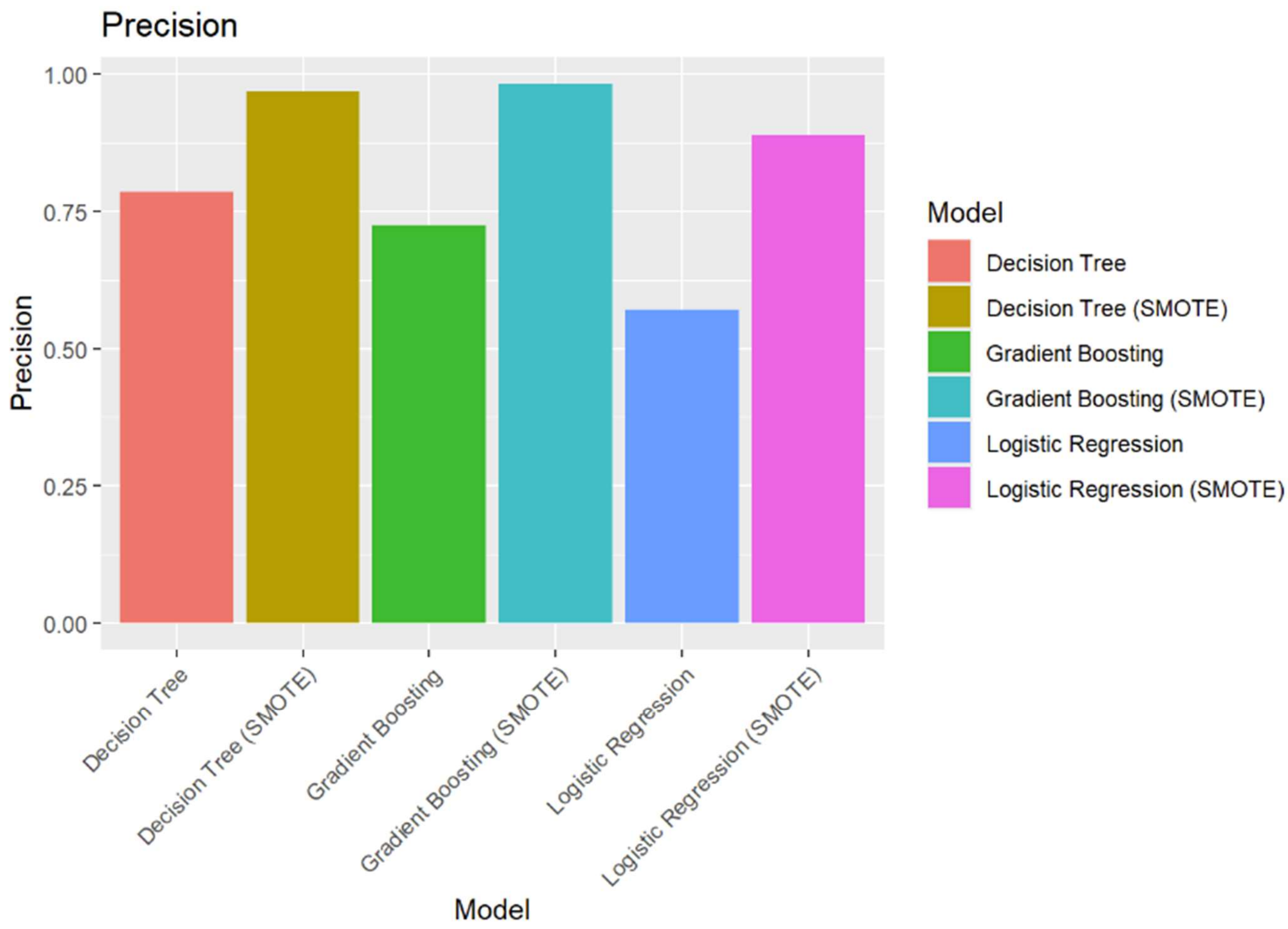
```
##                         Model    TN   FP   FN    TP Precision    Recall
## 1         Logistic Regression 56856   42    7    56 0.5714286 0.8888889
## 2               Decision Tree 56845   21   18    77 0.7857143 0.8105263
## 3           Gradient Boosting 56851   27   12    71 0.7244898 0.8554217
## 4 Logistic Regression (SMOTE) 27884 3181  512 25385 0.8886438 0.9802294
## 5       Decision Tree (SMOTE) 27302  891 1094 27675 0.9688091 0.9619730
## 6   Gradient Boosting (SMOTE) 28274  505  122 28061 0.9823216 0.9956711
```

3.15.1 Precision

Precision is the proportion of true positive predictions out of all positive predictions made by the model.

Precision = TP / (TP + FP)

The precision values for the different models are:

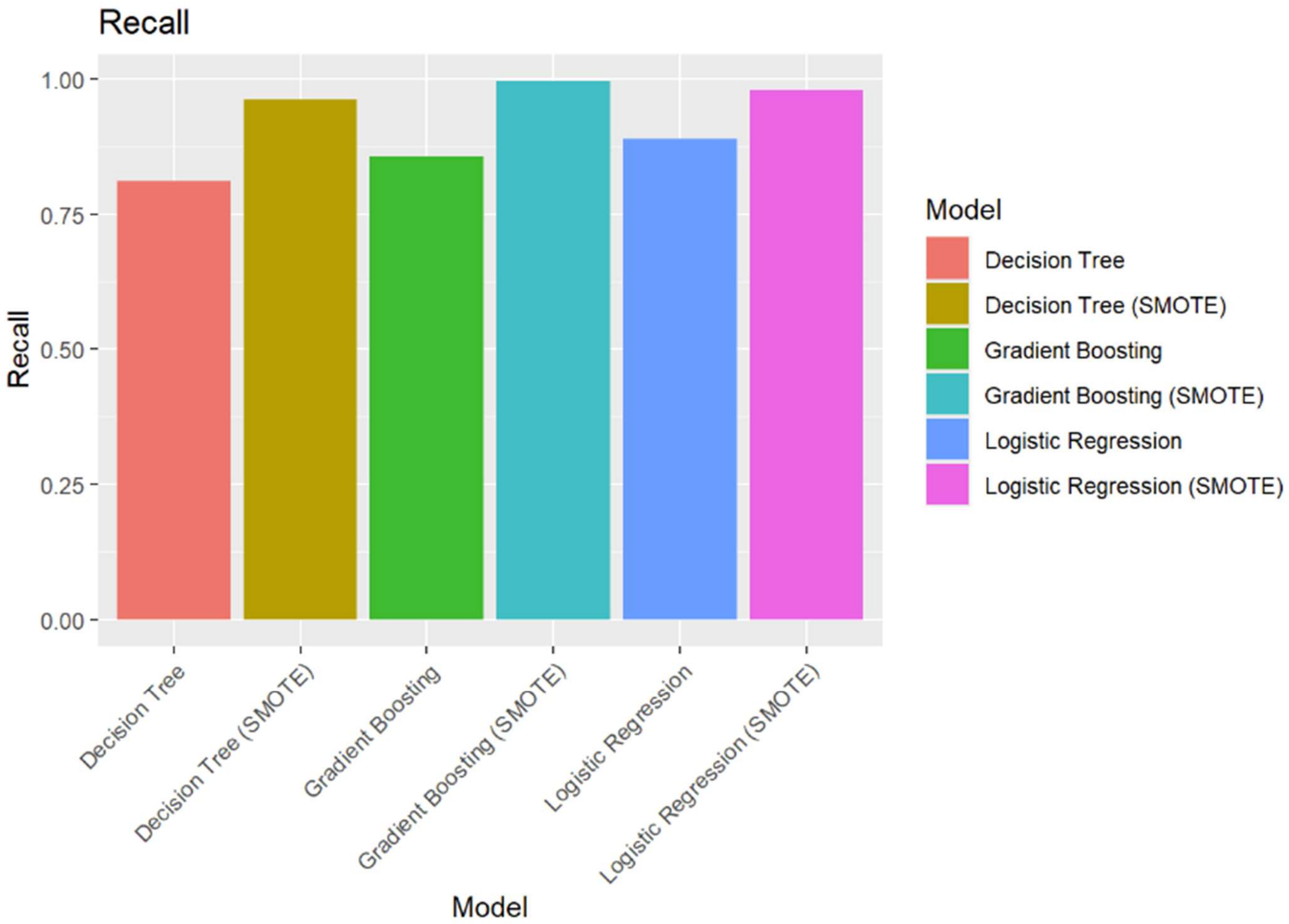


3.15.2 Recall

Recall, also known as sensitivity or true positive rate, is the proportion of actual positive instances that were correctly identified by the model.

Recall = TP / (TP + FN)

The recall values for the different models are:



A higher precision value indicates that the model makes fewer false positive predictions, while a higher recall value indicates that the model is better at identifying positive instances correctly. The trade-off between precision and recall can be evaluated to determine the most suitable model for the specific use case.

3.16 References:

[1] G. O. [1]Andrea Dal Pozzolo, Olivier Caelen, Reid A. Johnson and Gianluca Bontempi. Calibrating Probability with Undersampling for Unbalanced Classification. In Symposium on Computational Intelligence and Data Mining (CIDM), IEEE, 2015

[2] [2]http://mlg.ulb.ac.be(URL).

[3] [3]http://www.businesswire.com/news/home/20150804007054/en/Global-Card-Fraud-Losses-Reach-16.31-Billion

[4] [4]http://www.kaggle.com(URL).

[5] [5]The Nilson Report

[6] [6]http://www.thinksaveretire.com/2015/09/14/how-credit-card-fraud-detection-works/(URL).

[7] [7]SMOTE: Synthetic Minority Over-sampling Technique. Nitesh V. Chawla chawla@csee.usf.edu. Department of Computer Science and Engineering

[8] [8]Area Under the Precision-Recall Curve: Point Estimates and Confidence Intervals Kendrick Boyd1 , Kevin H. Eng , and C. David Page

[9] [9]http://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html(URL).

[10] [11]Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[11] [10]Jones E, Oliphant E, Peterson P, et al. SciPy: Open Source Scientific Tools for Python, 2001-, http://www.scipy.org/ [Online; accessed 2017-03-30]

[12] [11]Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37

```
# Loading necessary libraries
library(ranger)
```

```
## Warning: package 'ranger' was built under R version 4.3.3
```

```
library(caret)
```

```
## Loading required package: ggplot2
```

```
## Warning: package 'ggplot2' was built under R version 4.3.3
```

```
## Loading required package: lattice
```

```
library(data.table)
library(caTools)
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
## Attaching package: 'pROC'
```

```
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
library(rpart)
library(rpart.plot)
```

```
## Warning: package 'rpart.plot' was built under R version 4.3.3
```

```
library(neuralnet)
```

```
## Warning: package 'neuralnet' was built under R version 4.3.3
```

```
library(gbm)
```

```
## Warning: package 'gbm' was built under R version 4.3.3
```

```
## Loaded gbm 2.1.9
```

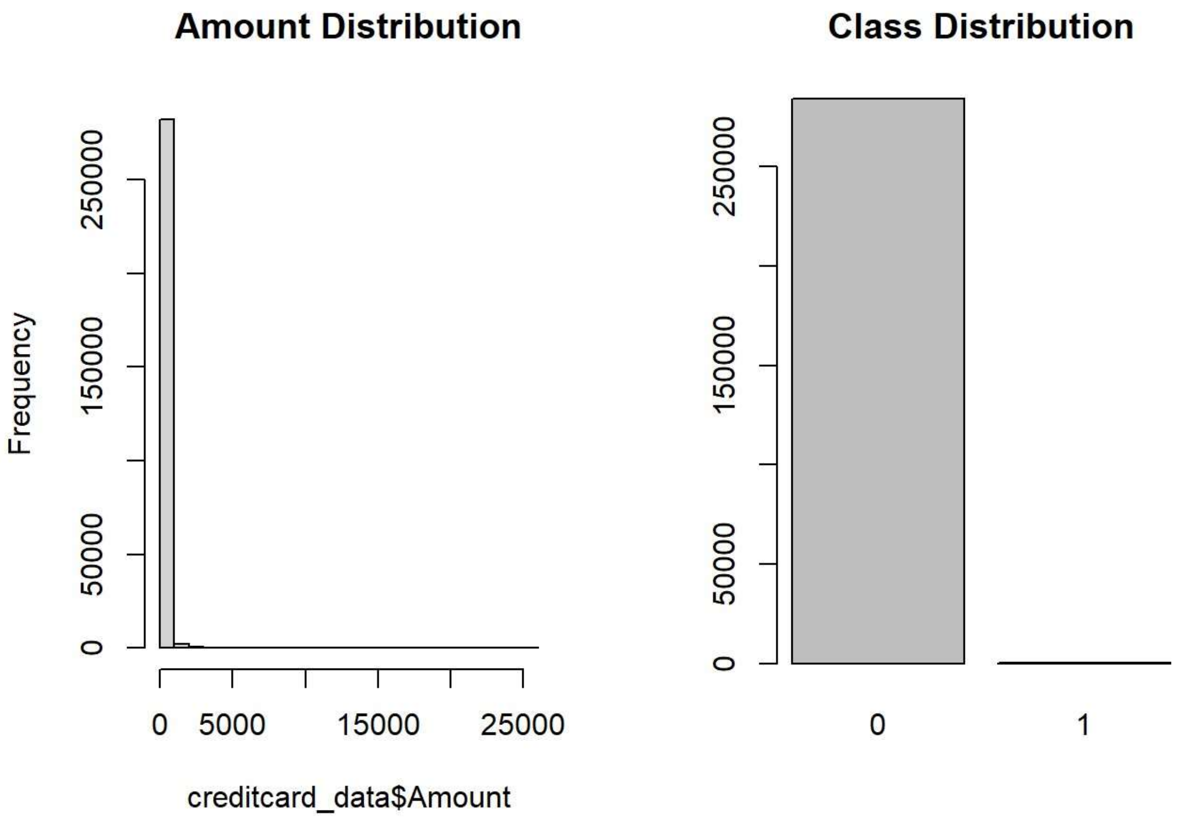
```
## This version of gbm is no longer under development. Consider transitioning to gbm3, http
s://github.com/gbm-developers/gbm3
```

```
library(ROSE) # For SMOTE oversampling
```

```
## Warning: package 'ROSE' was built under R version 4.3.3
```

```
## Loaded ROSE 0.0-4
```

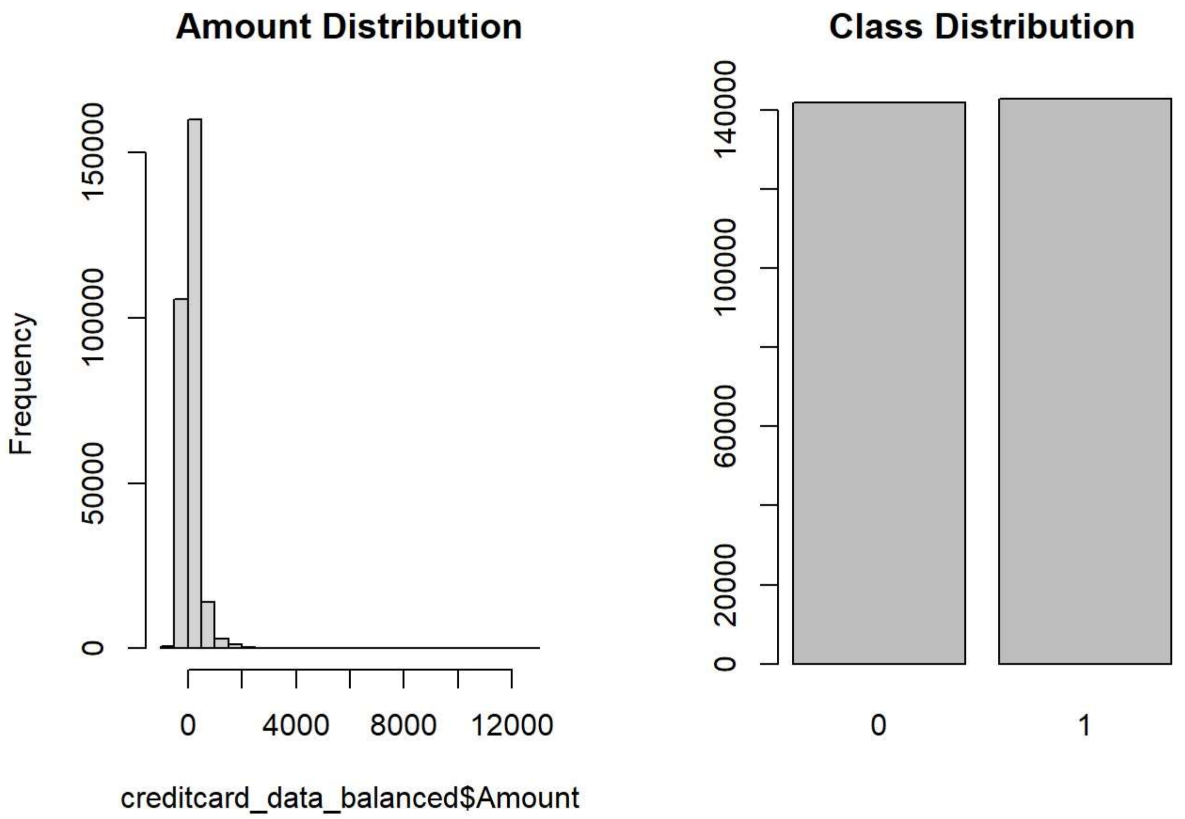
```
# Loading data
creditcard_data <- read.csv( "creditcard.csv" )

# Exploratory Data Analysis (EDA)
# Visualizing data distributions
par(mfrow=c( 1,2))
hist(creditcard_data$Amount, main= "Amount Distribution" )
barplot(table(creditcard_data$Class), main= "Class Distribution" )
```



```
class_distribution <- table(creditcard_data$Class)
print(class_distribution)
```

```
##
##      0      1
## 284315    492
```

```
# Applying SMOTE oversampling
creditcard_data_balanced <- ROSE(Class ~ ., data = creditcard_data)$data
par(mfrow=c( 1,2))

hist(creditcard_data_balanced$Amount, main= "Amount Distribution" )
barplot(table(creditcard_data_balanced$Class), main=  "Class Distribution" )
```

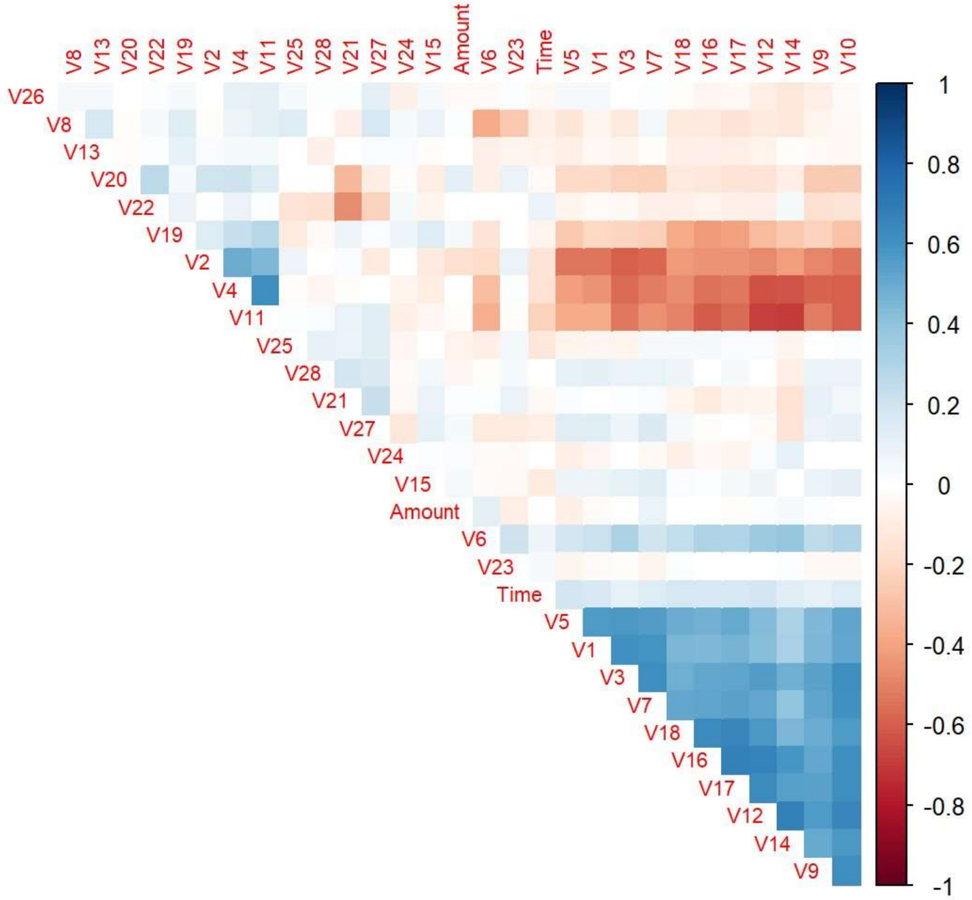


```
# Plotting correlation matrix heatmap
library(corrplot)
```

```
## Warning: package 'corrplot' was built under R version 4.3.3
```

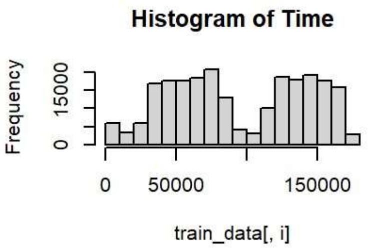
```
## corrplot 0.92 loaded
```

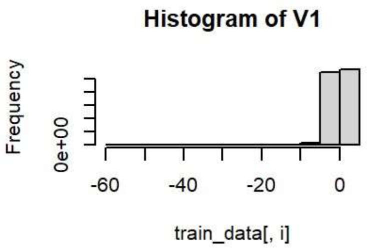
```
corr_matrix <- cor(creditcard_data_balanced[, -ncol(creditcard_data_balanced)])
corrplot(corr_matrix, method =  "color", type = "upper", order = "hclust",
         addrect = 8, tl.cex = 0.7, diag = FALSE)
```

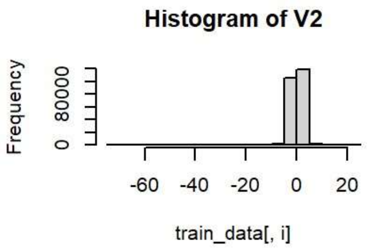


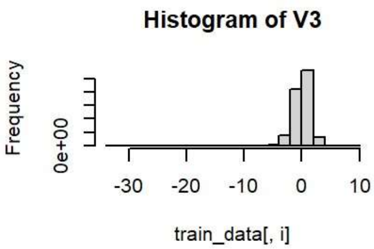
```
# Model Implementation Before SMOTE
# Splitting data
set.seed( 123)
data_sample <- sample.split(creditcard_data$Class, SplitRatio=  0.80)
train_data <- subset(creditcard_data, data_sample== TRUE)
test_data <- subset(creditcard_data, data_sample== FALSE)
```

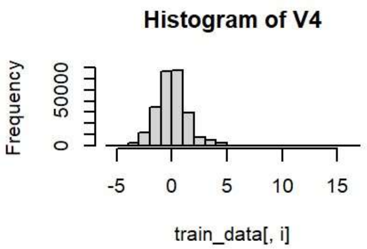
```
# Plotting histograms for continuous features
par(mfrow=c( 3,3))
for (i in 1:ncol(train_data))  {
  if (!is.factor(train_data[,i])) {
    hist(train_data[,i], main=paste( "Histogram of" , colnames(train_data)[i]))
  }
}
```

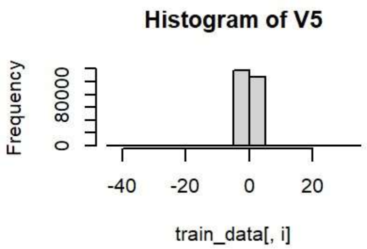

Histogram of Time

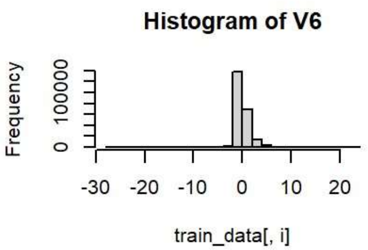

Histogram of V1

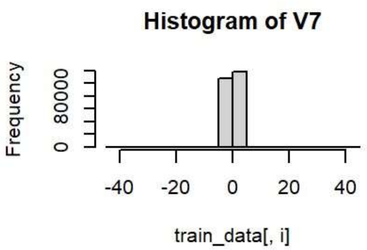

Histogram of V2

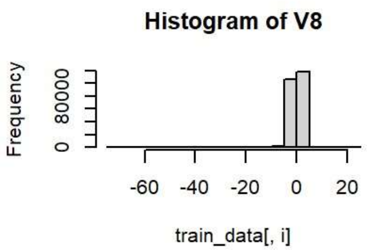

Histogram of V3

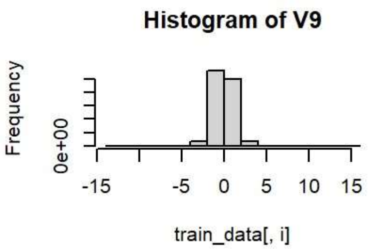

Histogram of V4

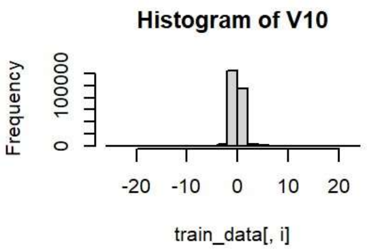

Histogram of V5

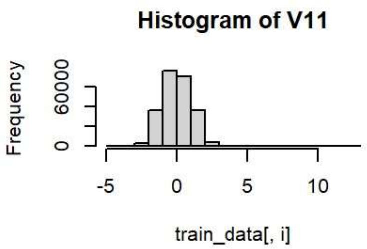

Histogram of V6

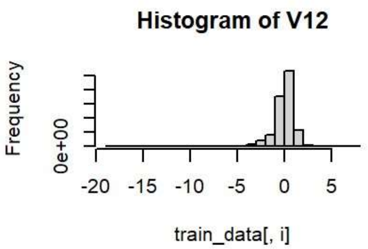

Histogram of V7

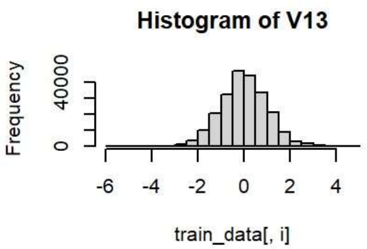

Histogram of V8

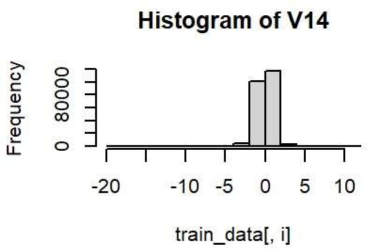

Histogram of V9

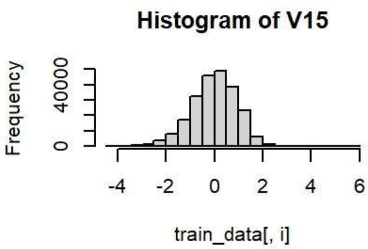

Histogram of V10

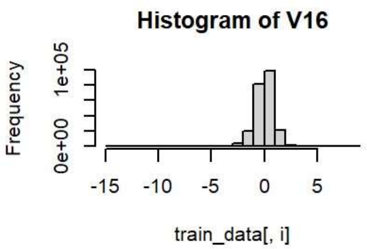

Histogram of V11

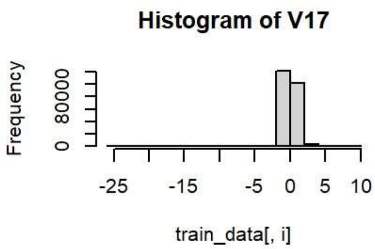

Histogram of V12

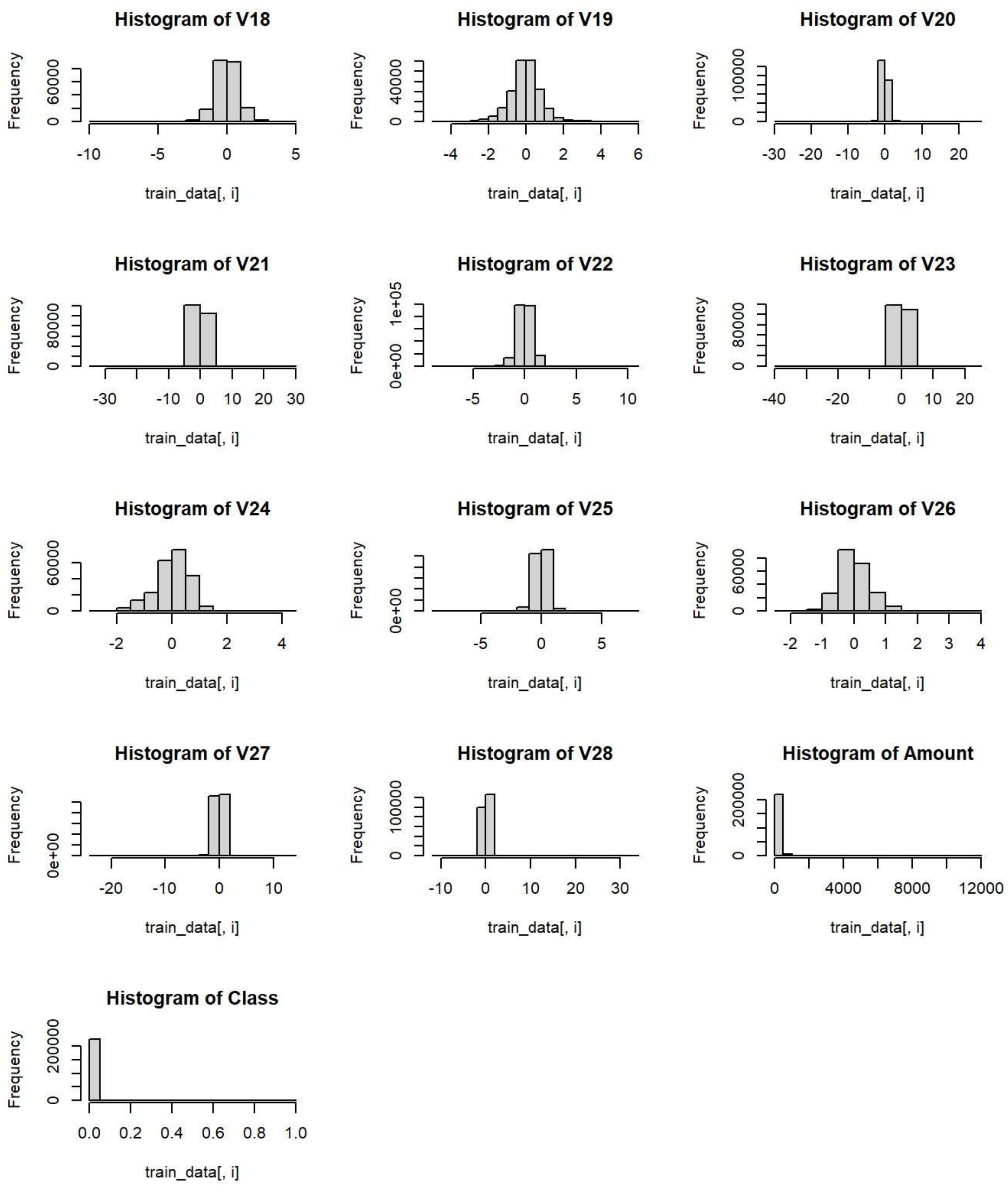

Histogram of V13


Histogram of V14


Histogram of V15


Histogram of V16

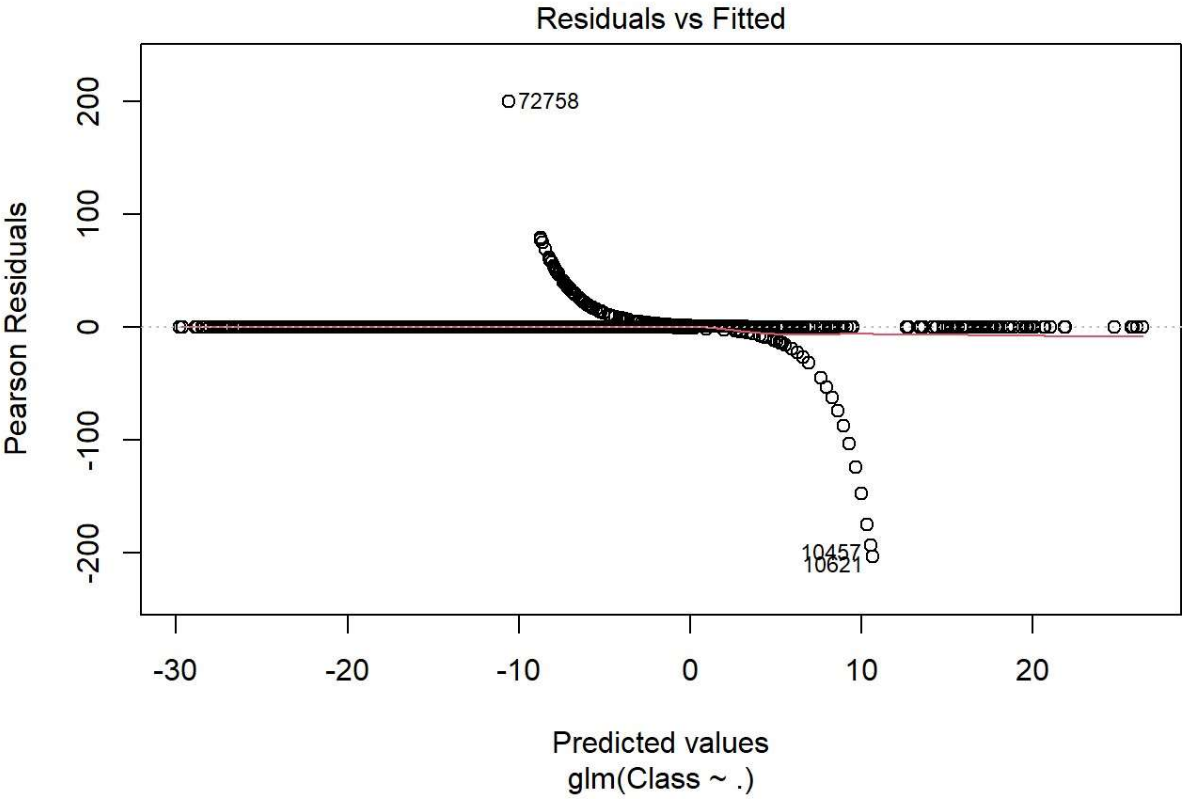

Histogram of V17

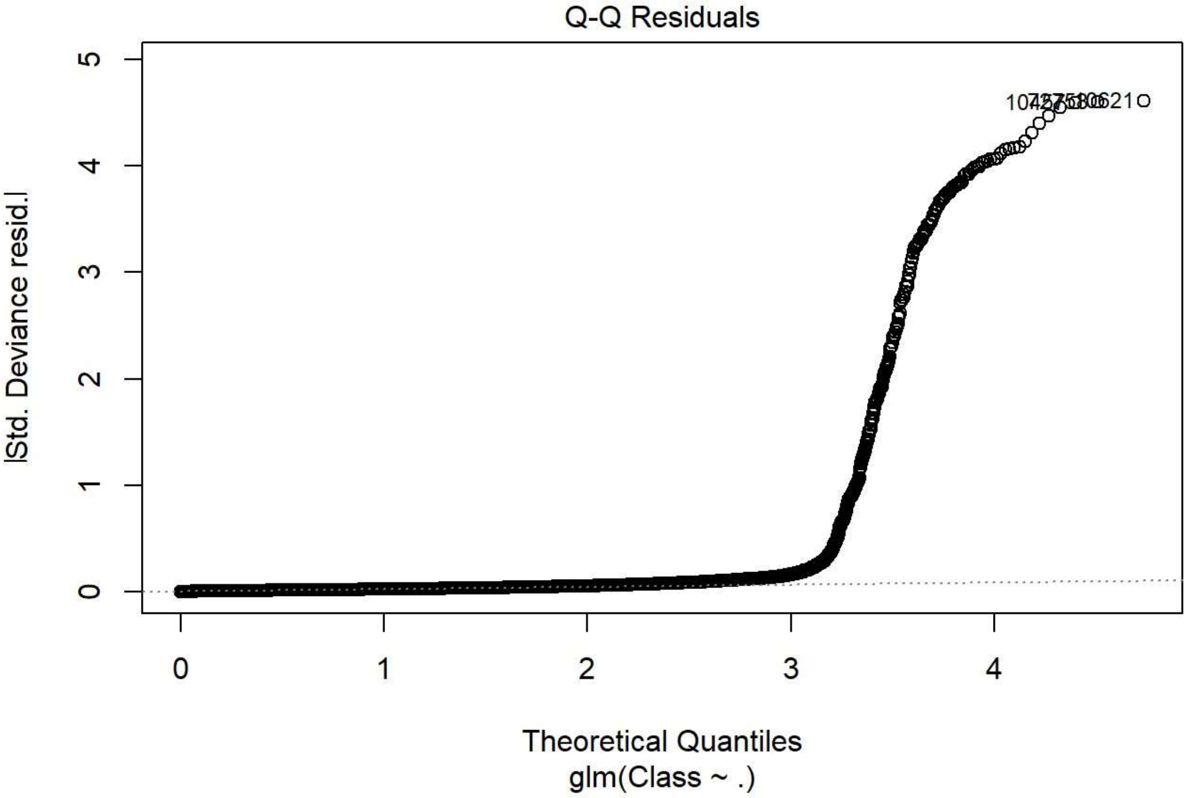


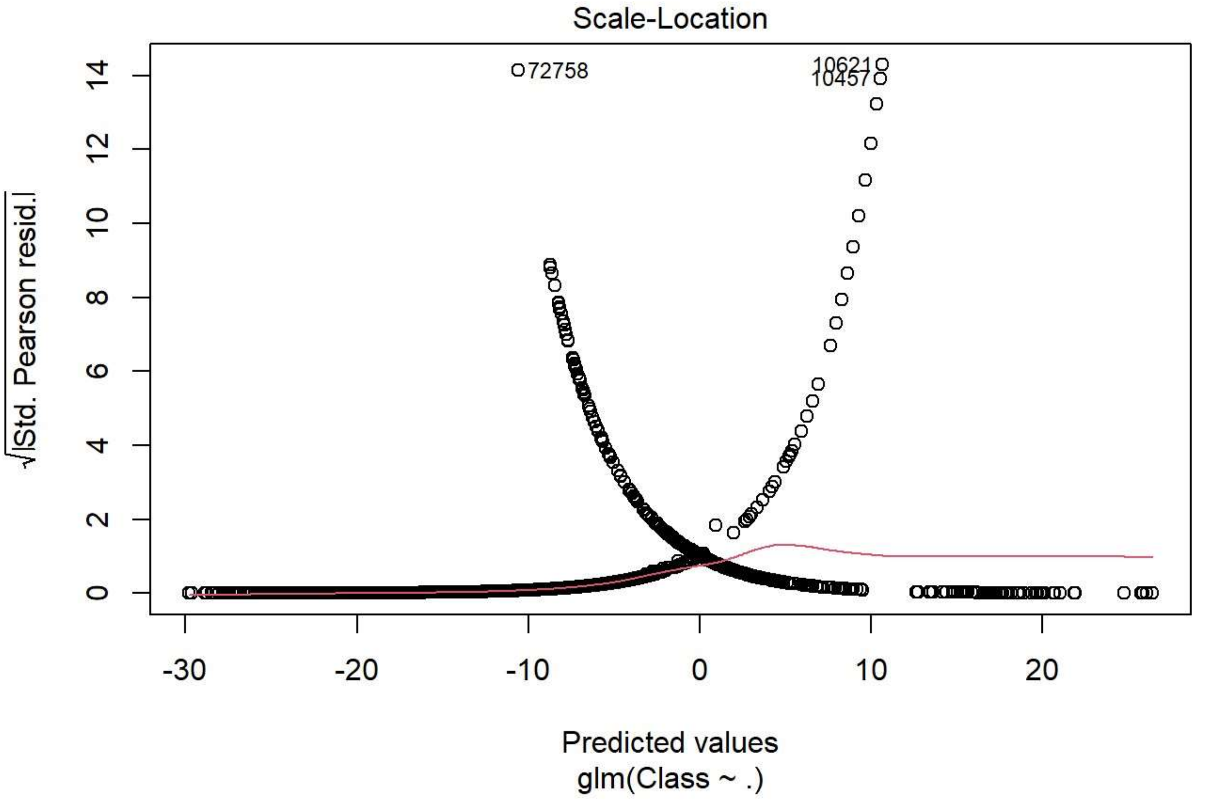
```
# Fitting Logistic Regression Model
Logistic_Model <- glm(Class ~ ., data = train_data, family = binomial())
summary(Logistic_Model)
```

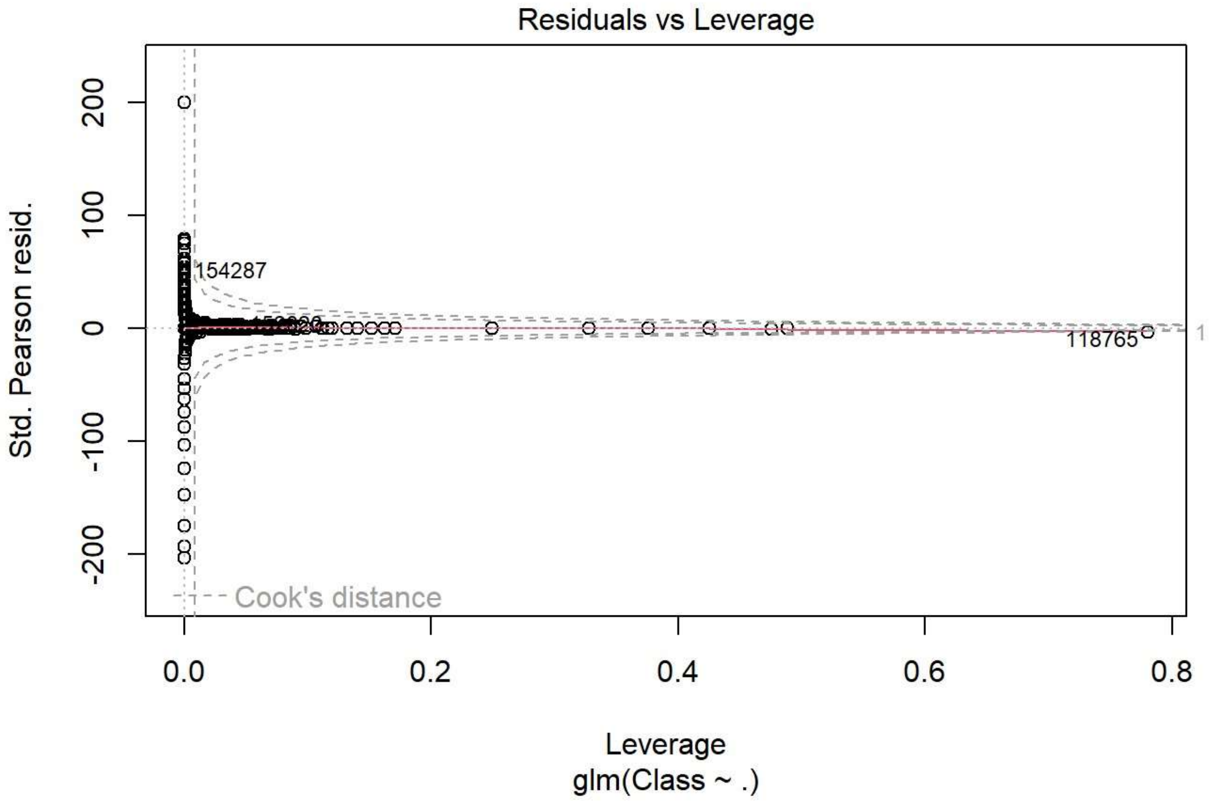
```
## ##
Call:
## glm(formula = Class ~ ., family = binomial(), data = train_data)
## ##
Coefficients:
##                Estimate Std. Error z value Pr(>|z|)
## (Intercept) -8.564e+00  2.866e-01 -29.887  < 2e-16 ***
## Time        -2.057e-06  2.567e-06  -0.801 0.423035    ##
V1           7.883e-02  4.497e-02   1.753 0.079640 .
## V2           1.315e-02  5.988e-02   0.220 0.826155
## V3           3.916e-04  5.897e-02   0.007 0.994702
## V4           6.767e-01  7.827e-02   8.645  < 2e-16 ***
## V5           1.048e-01  7.400e-02   1.417 0.156603    ##
V6          -1.442e-01  8.378e-02  -1.722 0.085111 .
## V7          -1.100e-01  6.967e-02  -1.578 0.114485
## V8          -1.482e-01  3.564e-02  -4.158 3.20e-05 ***
## V9          -3.521e-01  1.187e-01  -2.966 0.003013 **
## V10         -7.830e-01  9.848e-02  -7.950 1.86e-15 ***
## V11         -2.581e-02  9.154e-02  -0.282 0.777964
## V12          9.925e-02  9.609e-02   1.033 0.301674
## V13         -3.485e-01  9.261e-02  -3.763 0.000168 ***
## V14         -5.340e-01  6.778e-02  -7.879 3.29e-15 ***
## V15         -1.112e-01  9.602e-02  -1.158 0.246719
## V16         -1.285e-01  1.387e-01  -0.927 0.353923
## V17          1.883e-02  7.635e-02   0.247 0.805231
## V18         -8.744e-02  1.422e-01  -0.615 0.538759
## V19          9.464e-02  1.062e-01   0.891 0.373023
## V20         -4.680e-01  8.210e-02  -5.700 1.19e-08 ***
## V21          3.928e-01  6.778e-02   5.795 6.84e-09 ***
## V22          6.351e-01  1.464e-01   4.339 1.43e-05 ***
## V23         -7.515e-02  5.957e-02  -1.262 0.207116
## V24          2.491e-01  1.696e-01   1.469 0.141953
## V25         -9.095e-02  1.445e-01  -0.629 0.529118
## V26          1.353e-01  2.033e-01   0.665 0.505793
## V27         -8.573e-01  1.184e-01  -7.238 4.56e-13 ***
## V28         -3.219e-01  8.944e-02  -3.599 0.000319 ***
## Amount       1.167e-03  3.696e-04   3.157 0.001593 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 5799.1  on 227845  degrees of freedom
## Residual deviance: 1790.3  on 227815  degrees of freedom
## AIC: 1852.3
## ## Number of Fisher Scoring iterations:
12
```

```
plot(Logistic_Model)
```

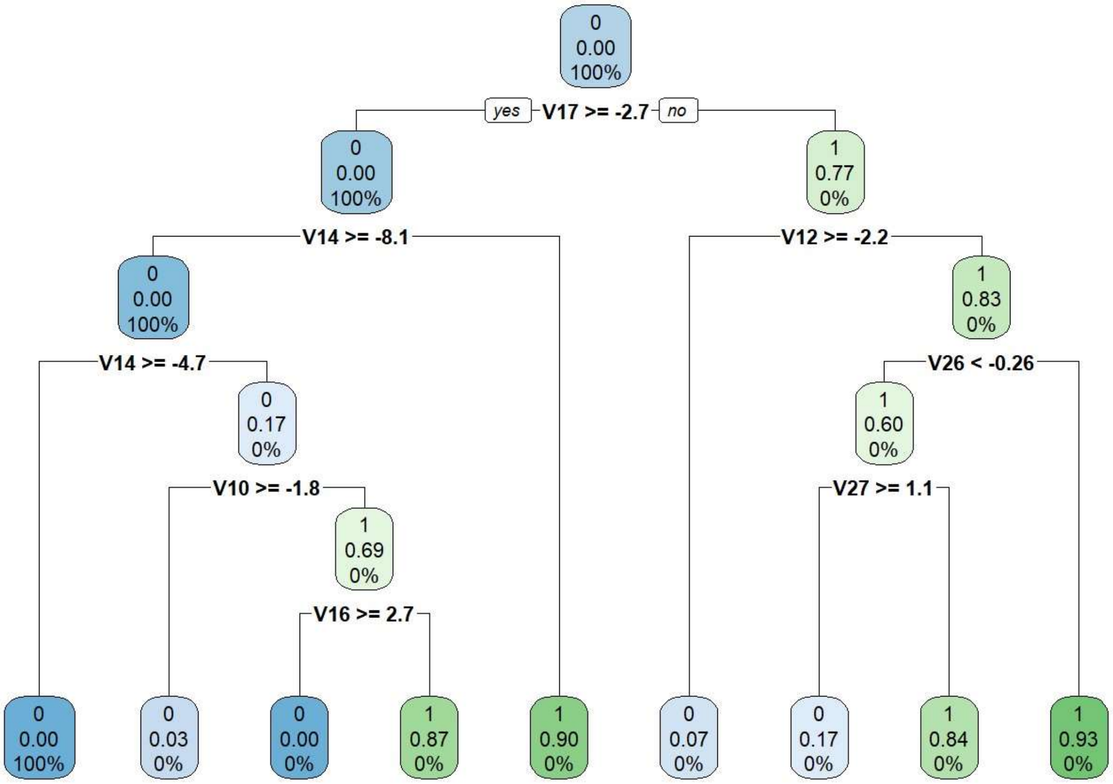

Residuals vs Fitted
Pearson Residuals
200
100
0
-100
-200
72758
10457
10621
-30
-20
-10
0
10
20
Predicted values
glm(Class ~ .)


Q-Q Residuals
|Std. Deviance resid.|
5
4
3
2
1
0
10621
0
1
2
3
4
Theoretical Quantiles
glm(Class ~ .)





```
# Fitting Decision Tree Model
decisionTree_model <- rpart(Class ~ . , data = train_data, method =   'class')
rpart.plot(decisionTree_model)
```

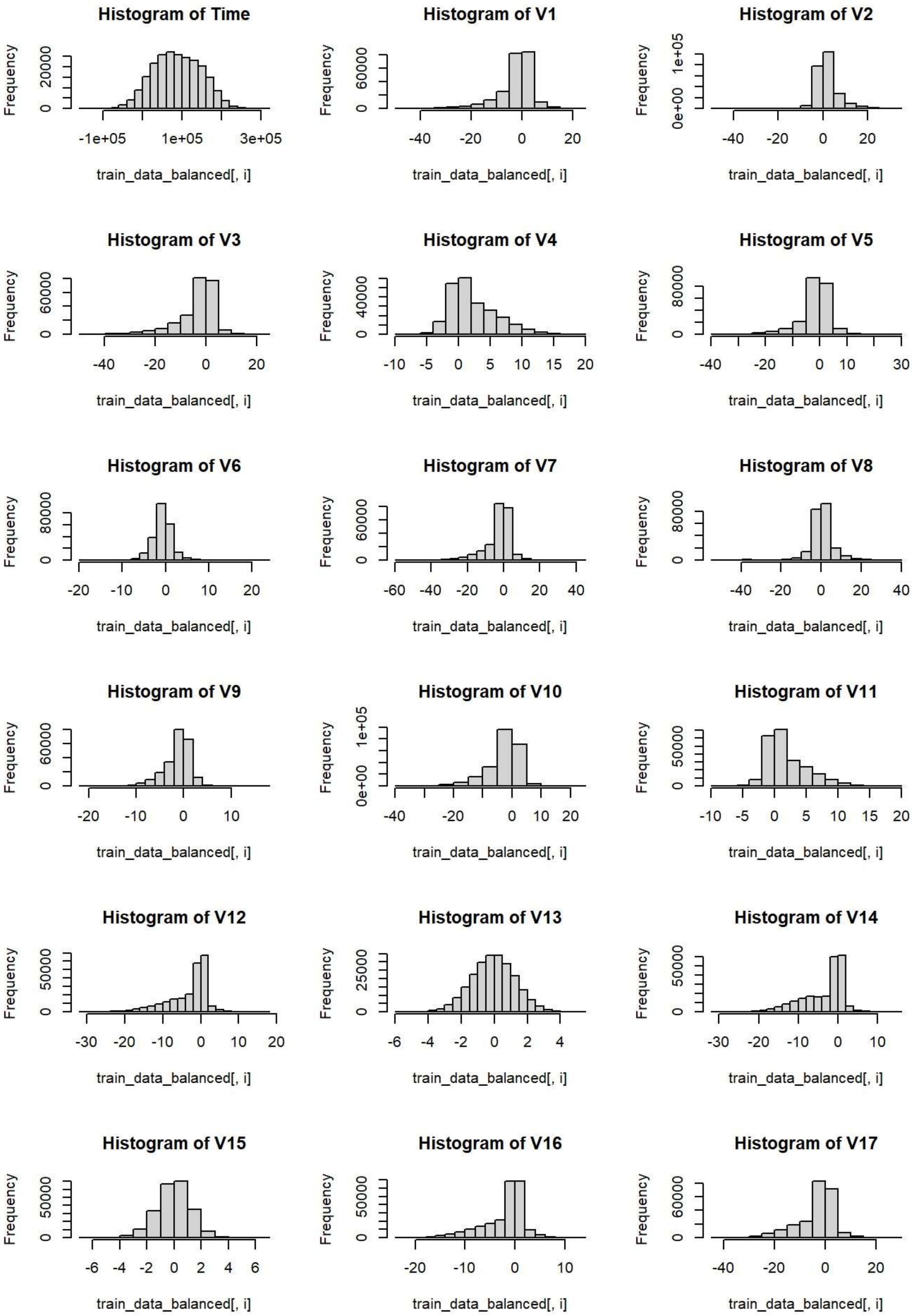


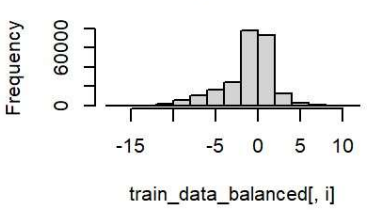
```
# Plotting correlation matrix heatmap
#corr_matrix <- cor(creditcard_data_balanced[, -ncol(creditcard_data_balanced)])
#corrplot(corr_matrix, method = "color", type = "upper", order = "hclust",
#         addrect = 8, tl.cex = 0.7, diag = FALSE)
```

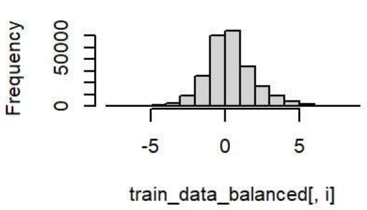
```
# Fitting Gradient Boosting Model
model_gbm <- gbm(Class ~ ., distribution =  "bernoulli" , data = train_data,
                 n.trees = 500, interaction.depth =  3, n.minobsinnode =  100,
                 shrinkage = 0.01, bag.fraction = 0.5,
                 train.fraction = nrow(train_data) / (nrow(train_data) + nrow(test_data)))
```

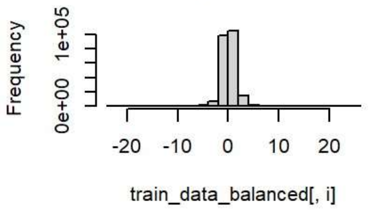
```
# Model Implementation After SMOTE
# Splitting balanced data
set.seed( 123)
data_sample_balanced <- sample.split(creditcard_data_balanced$Class, SplitRatio=  0.80)
train_data_balanced <- subset(creditcard_data_balanced, data_sample_balanced==  TRUE)
test_data_balanced <- subset(creditcard_data_balanced, data_sample_balanced==  FALSE)
```

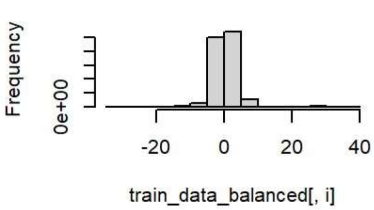
```
# Plotting histograms for continuous features in balanced data
par(mfrow=c( 3,3))
for (i in 1:ncol(train_data_balanced))  {
  if (!is.factor(train_data_balanced[,i])) {
    hist(train_data_balanced[,i], main=paste( "Histogram of" , colnames(train_data_balanced)
[i]))
  }
}
```

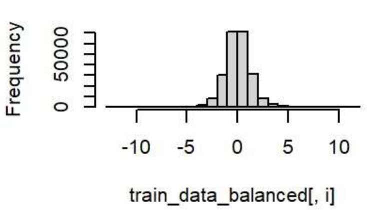

Histogram of Time
Frequency
0
20000
-1e+05
1e+05
3e+05
train_data_balanced[, i]
Histogram of V1
Frequency
0
60000
-40
-20
0
20
train_data_balanced[, i]
Histogram of V2
Frequency
0e+00
1e+05
-40
-20
0
20
train_data_balanced[, i]
Histogram of V3
Frequency
0
60000
-40
-20
0
20
train_data_balanced[, i]
Histogram of V4
Frequency
0
40000
-10
-5
0
5
10
15
20
train_data_balanced[, i]
Histogram of V5
Frequency
0
80000
-40
-20
0
10
30
train_data_balanced[, i]
Histogram of V6
Frequency
0
80000
-20
-10
0
10
20
train_data_balanced[, i]
Histogram of V7
Frequency
0
60000
-60
-40
-20
0
20
40
train_data_balanced[, i]
Histogram of V8
Frequency
0
80000
-40
-20
0
20
40
train_data_balanced[, i]
Histogram of V9
Frequency
0
60000
-20
-10
0
10
train_data_balanced[, i]
Histogram of V10
Frequency
0e+00
1e+05
-40
-20
0
10
20
train_data_balanced[, i]
Histogram of V11
Frequency
0
50000
-10
-5
0
5
10
15
20
train_data_balanced[, i]
Histogram of V12
Frequency
0
50000
-30
-20
-10
0
10
20
train_data_balanced[, i]
Histogram of V13
Frequency
0
25000
-6
-4
-2
0
2
4
train_data_balanced[, i]
Histogram of V14
Frequency
0
50000
-30
-20
-10
0
10
train_data_balanced[, i]
Histogram of V15
Frequency
0
50000
-6
-4
-2
0
2
4
6
train_data_balanced[, i]
Histogram of V16
Frequency
0
50000
-20
-10
0
10
train_data_balanced[, i]
Histogram of V17
Frequency
0
60000
-40
-20
0
20
train_data_balanced[, i]

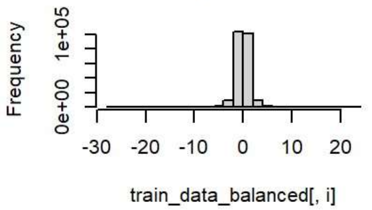

Histogram of V18

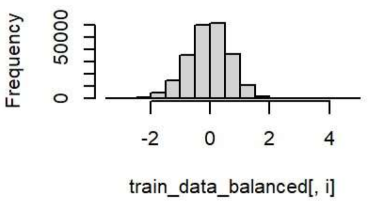

Histogram of V19

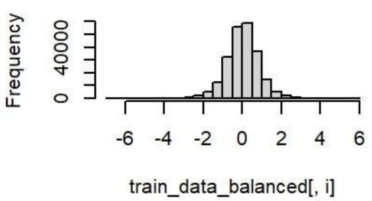

Histogram of V20

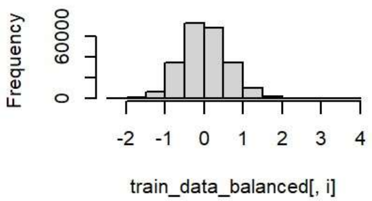

Histogram of V21

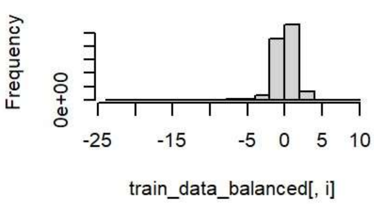

Histogram of V22

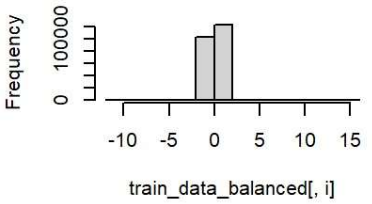

Histogram of V23

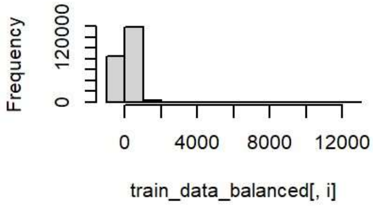

Histogram of V24

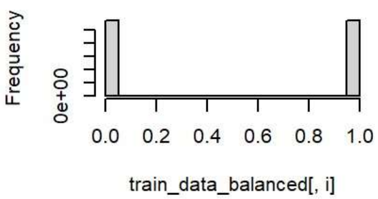

Histogram of V25


Histogram of V26


Histogram of V27


Histogram of V28


Histogram of Amount


Histogram of Class

```r
# Plotting bar plots for categorical features in balanced data
par(mfrow=c(1,1))
for (i in 1:ncol(train_data_balanced)) {
  if (is.factor(train_data_balanced[,i])) {
    barplot(table(train_data_balanced[,i]), main=paste("Barplot of", colnames(train_data_balanced)[i]))
  }
}
```

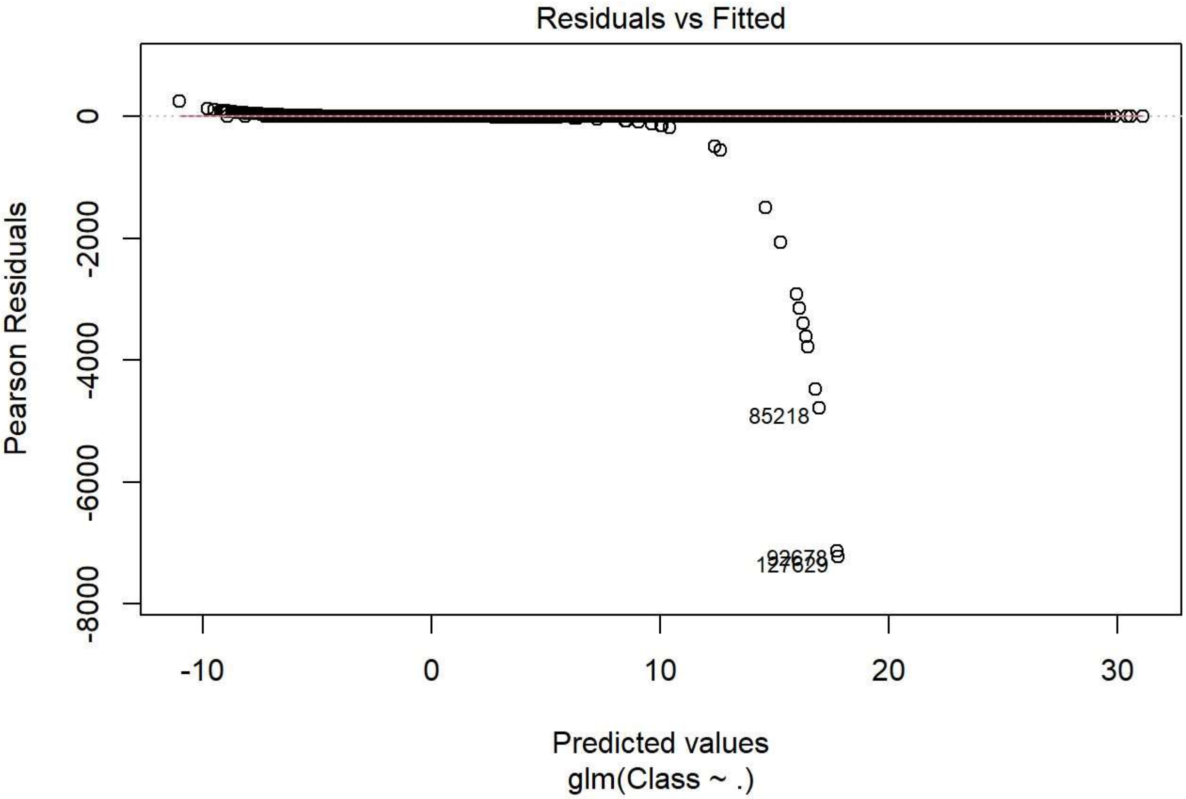
```r
# Fitting Logistic Regression Model after SMOTE
Logistic_Model_balanced <- glm(Class ~ ., data = train_data_balanced, family = binomial())
```

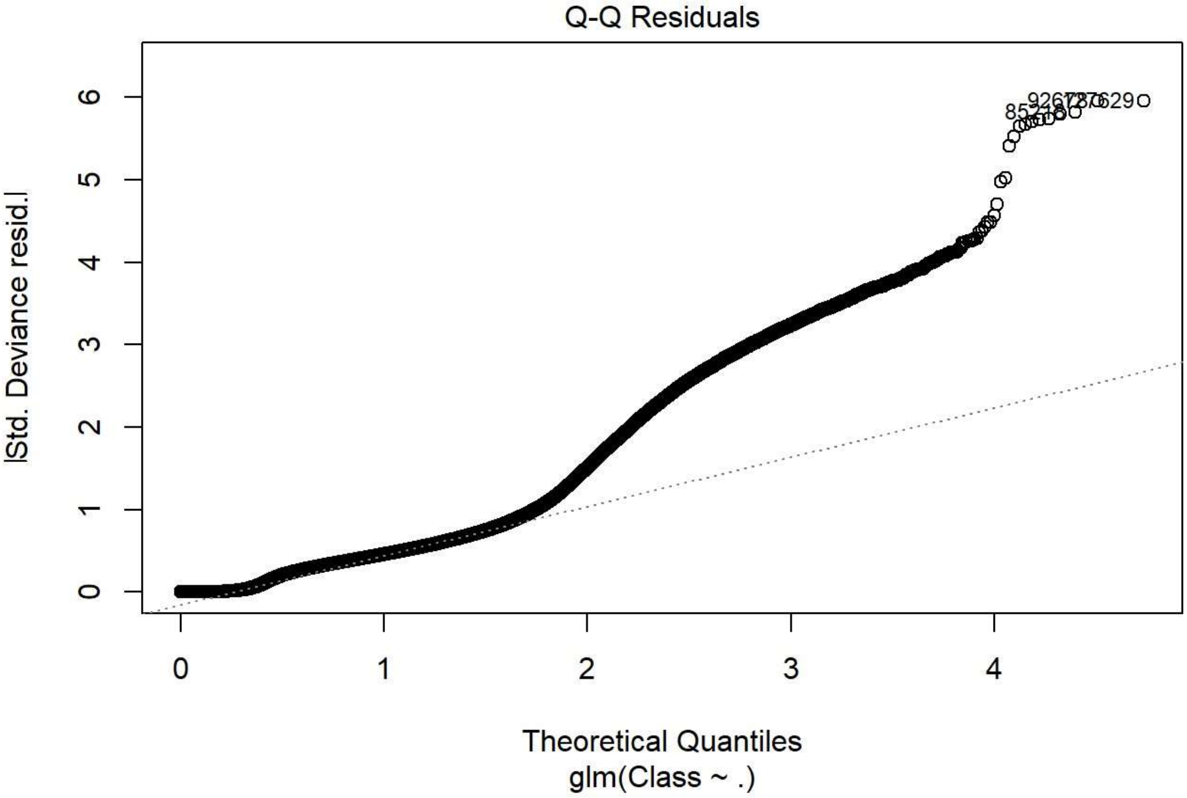
```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

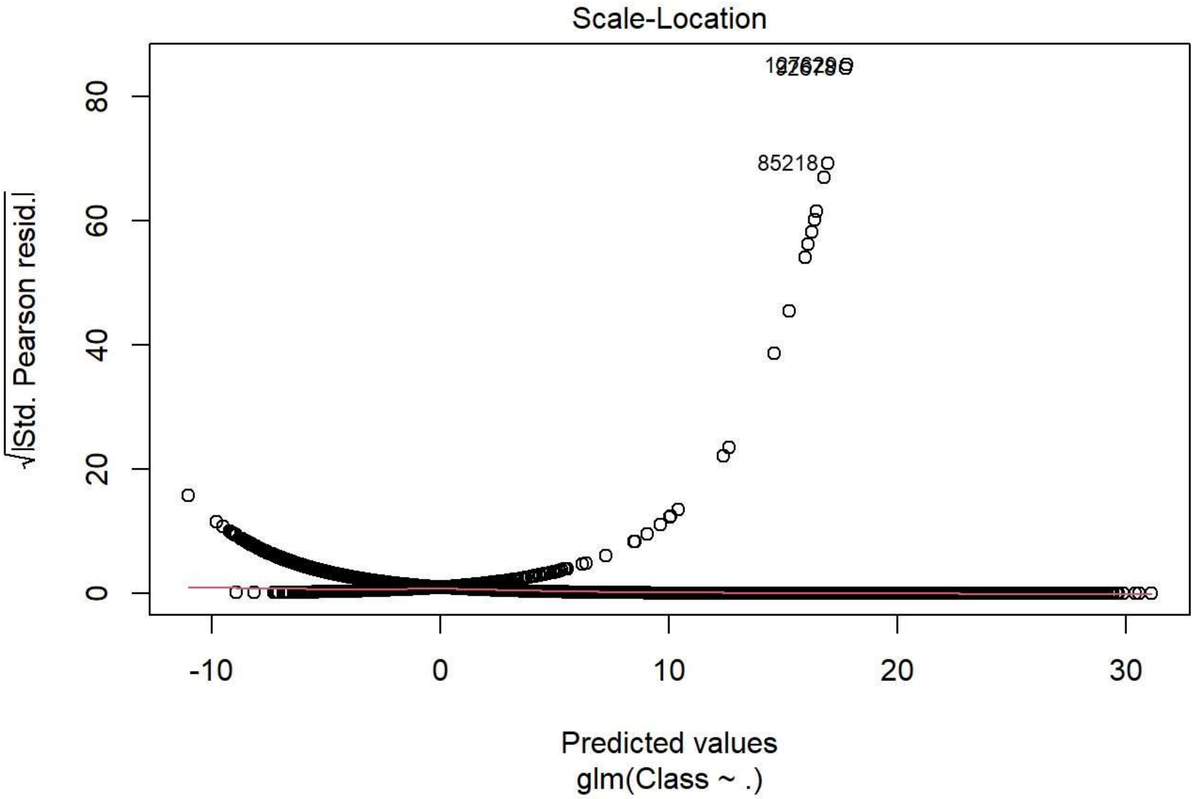
```r
summary(Logistic_Model_balanced)
```

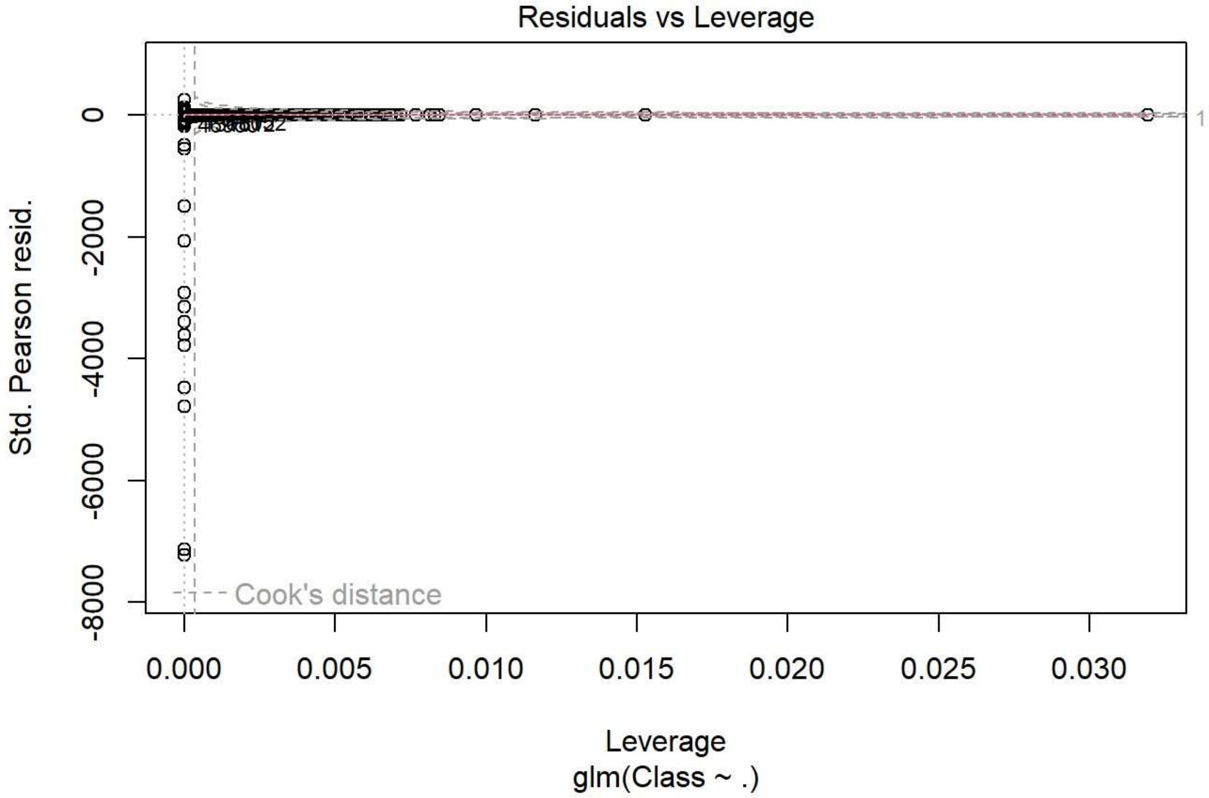
```
## ## Call: ## glm(formula = Class ~ ., family = binomial(), data =
train_data_balanced)
## ##
Coefficients:
##               Estimate Std. Error  z value Pr(>|z|)
## (Intercept) -2.257e+00  1.778e-02 -126.927  < 2e-16 ***
## Time        -1.487e-06  1.526e-07   -9.745  < 2e-16 ***
## V1          -3.726e-02  2.343e-03  -15.903  < 2e-16 ***
## V2           7.078e-02  3.326e-03   21.279  < 2e-16 ***
## V3          -7.703e-02  2.469e-03  -31.200  < 2e-16 ***
## V4           4.644e-01  4.232e-03  109.735  < 2e-16 ***
## V5           3.104e-02  3.044e-03   10.199  < 2e-16 ***
## V6          -1.608e-01  5.303e-03  -30.316  < 2e-16 ***
## V7          -2.879e-02  2.465e-03  -11.678  < 2e-16 ***
## V8          -5.509e-02  2.553e-03  -21.580  < 2e-16 ***
## V9          -1.300e-01  5.246e-03  -24.779  < 2e-16 ***
## V10         -1.558e-01  3.503e-03  -44.476  < 2e-16 ***
## V11          2.728e-01  5.082e-03   53.673  < 2e-16 ***
## V12         -2.260e-01  3.684e-03  -61.348  < 2e-16 ***
## V13         -1.765e-01  6.660e-03  -26.501  < 2e-16 ***
## V14         -4.010e-01  3.913e-03 -102.496  < 2e-16 ***
## V15         -5.588e-02  7.220e-03   -7.740 9.91e-15 ***
## V16         -1.197e-01  4.256e-03  -28.119  < 2e-16 ***
## V17         -3.424e-02  2.493e-03  -13.733  < 2e-16 ***
## V18          1.297e-02  5.234e-03    2.478   0.0132 *
## V19         -1.807e-02  6.815e-03   -2.652   0.0080 **
## V20         -7.779e-02  7.672e-03  -10.139  < 2e-16 ***
## V21          5.949e-02  4.273e-03   13.924  < 2e-16 ***
## V22          1.136e-01  8.068e-03   14.078  < 2e-16 ***
## V23         -7.971e-02  6.635e-03  -12.013  < 2e-16 ***
## V24         -6.428e-02  1.221e-02   -5.264 1.41e-07 ***
## V25          6.410e-03  1.188e-02    0.539   0.5896
## V26         -1.746e-01  1.444e-02  -12.092  < 2e-16 ***
## V27          7.806e-02  1.072e-02    7.281 3.30e-13 ***
## V28          2.066e-01  1.905e-02   10.845  < 2e-16 ***
## Amount       9.249e-04  2.882e-05   32.086  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 315858  on 227844  degrees of freedom
## Residual deviance:  99216  on 227814  degrees of freedom
## AIC: 99278
## ## Number of Fisher Scoring iterations:
8
```

```
plot(Logistic_Model_balanced)
```

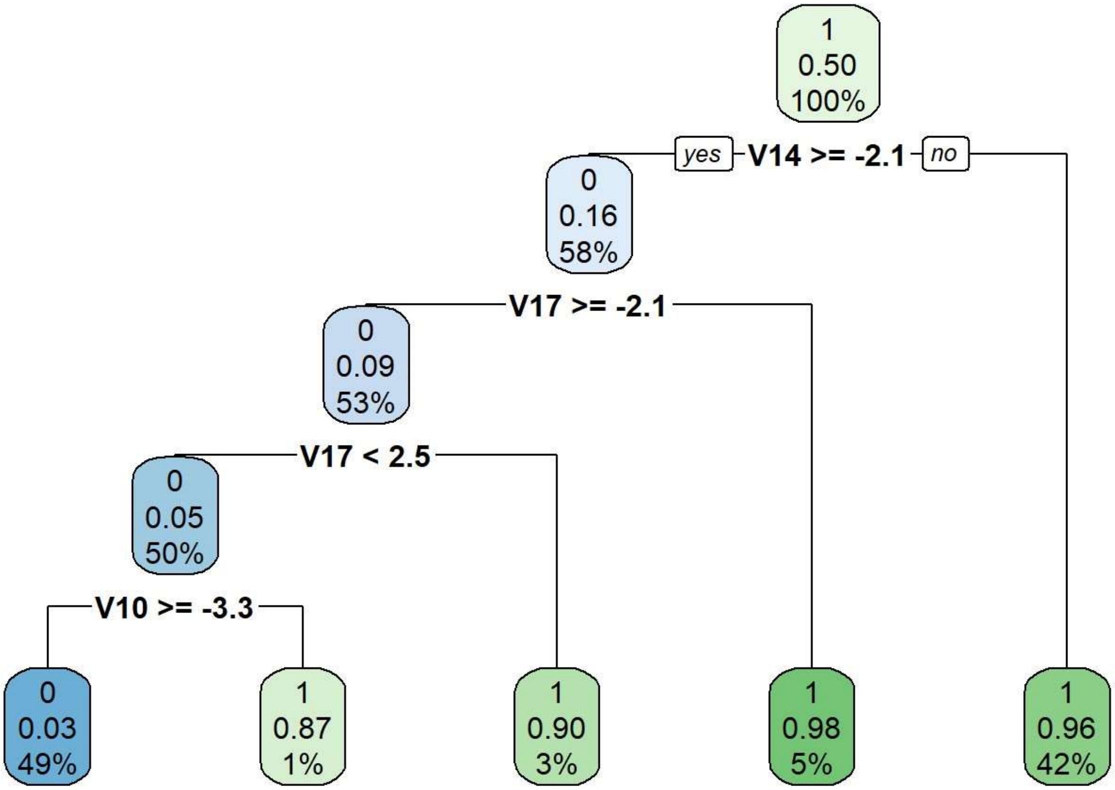

Residuals vs Fitted
Pearson Residuals
0
-2000
-4000
-6000
-8000
-10
0
10
20
30
85218
92678
127629
Predicted values
glm(Class ~ .)


Q-Q Residuals
|Std. Deviance resid.|
0
1
2
3
4
5
6
0
1
2
3
4
92678
127629
85218
Theoretical Quantiles
glm(Class ~ .)


Scale-Location
127629
92678
85218
80
60
40
20
0
√|Std. Pearson resid.|
-10
0
10
20
30
Predicted values
glm(Class ~ .)


Residuals vs Leverage
1
Cook's distance
0
-2000
-4000
-6000
-8000
Std. Pearson resid.
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Leverage
glm(Class ~ .)

```
# Fitting Decision Tree Model after SMOTE
decisionTree_model_balanced <- rpart(Class ~ . , data = train_data_balanced, method =    'class')
rpart.plot(decisionTree_model_balanced)
```



```
# Fitting Artificial Neural Network after SMOTE with smaller dataset
#train_data_subset <- train_data_balanced[sample(nrow(train_data_balanced), 10000, replace = FALSE), ]
#ANN_model_optimized_subset <- neuralnet(Class ~ ., train_data_subset, hidden = c(10), linear.output=FALSE,
#                                        learningrate = 0.01, algorithm = "rprop+", stepmax = 1e6)
#plot(ANN_model_optimized_subset)
```

```
# Fitting Gradient Boosting Model after SMOTE
model_gbm_balanced <- gbm(Class ~ ., distribution =  "bernoulli", data = train_data_balanced,
                          n.trees = 500, interaction.depth = 3, n.minobsinnode = 100,
                          shrinkage = 0.01, bag.fraction = 0.5,
                          train.fraction = nrow(train_data_balanced) /
                            (nrow(train_data_balanced) + nrow(test_data_balanced )))
```

```r
# Evaluate models and plot confusion matrices
evaluate_model <- function(model, test_data) {
  predictions <- predict(model, test_data, type= "response")
  pred_class <- ifelse(predictions > 0.5, 1, 0)
  confusionMatrix(data = factor(pred_class), reference = factor(test_data$Class))
}
```

```r
# Define a function to evaluate model and calculate confusion matrix
evaluate_model <- function(model, test_data) {
  if (inherits(model, "rpart")) { # Decision tree model
    predictions <- predict(model, test_data, type = "class")
  } else { # Assume logistic regression, neural network, or other binary classification model
s
    predictions <- predict(model, test_data, type = "response")
    predictions <- ifelse(predictions > 0.5, 1, 0) # Convert probabilities to class labels
  }
  confusionMatrix(data = factor(predictions), reference = factor(test_data$Class))
}
```

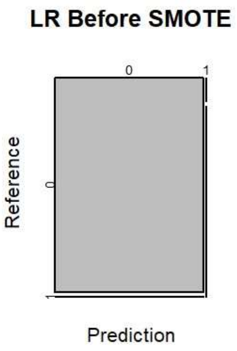
```r
# Confusion matrices before SMOTE
confusion_LR <- evaluate_model(Logistic_Model, test_data)
confusion_DT <- evaluate_model(decisionTree_model, test_data)
#confusion_ANN <- evaluate_model(ANN_model, test_data)
confusion_GBM <- evaluate_model(model_gbm, test_data)
```

```
## Using 500 trees...
```

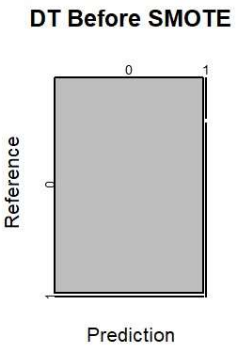
```r
# Confusion matrices after SMOTE
confusion_LR_balanced <- evaluate_model(Logistic_Model_balanced, test_data_balanced)
confusion_DT_balanced <- evaluate_model(decisionTree_model_balanced, test_data_balanced)
#confusion_ANN_optimized_subset <- evaluate_model(ANN_model_optimized_subset, test_data_balan
ced)
confusion_GBM_balanced <- evaluate_model(model_gbm_balanced, test_data_balanced)
```

```
## Using 500 trees...
```

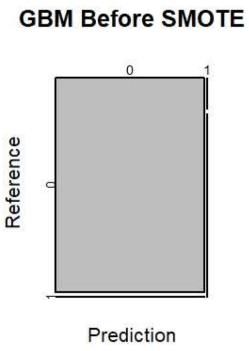
```r
# Plotting confusion matrices
par(mfrow=c(2,4))
plot(confusion_LR$table, main= "LR Before SMOTE")
plot(confusion_DT$table, main= "DT Before SMOTE")
#plot(confusion_ANN$table, main="ANN Before SMOTE")
plot(confusion_GBM$table, main= "GBM Before SMOTE")

plot(confusion_LR_balanced$table, main= "LR After SMOTE")
plot(confusion_DT_balanced$table, main= "DT After SMOTE")
#plot(confusion_ANN_optimized_subset$table, main="ANN After SMOTE (Optimized)")
plot(confusion_GBM_balanced$table, main= "GBM After SMOTE")
```



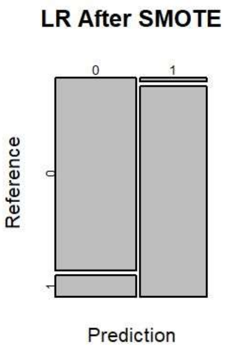
LR Before SMOTE



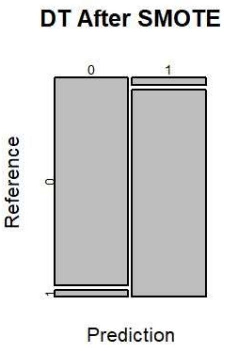
DT Before SMOTE



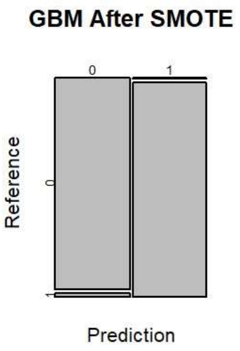
GBM Before SMOTE



LR After SMOTE



DT After SMOTE



GBM After SMOTE

```
# Function to calculate performance metrics
calc_performance <- function(cm) {   tn <-
cm$table[1, 1]   fp <- cm$table[1, 2]   fn
<- cm$table[2, 1]   tp <- cm$table[2, 2]
    precision <- tp / (tp +
fp)   recall <- tp / (tp +
fn)

  return(list(TN = tn, FP = fp, FN = fn, TP = tp, Precision = precision, Recall = recall)) }

# Calculate performance metrics for all models
perf_LR <- calc_performance(confusion_LR) perf_DT
<- calc_performance(confusion_DT) perf_GBM <-
calc_performance(confusion_GBM)

perf_LR_balanced <- calc_performance(confusion_LR_balanced) perf_DT_balanced
<- calc_performance(confusion_DT_balanced) perf_GBM_balanced <-
calc_performance(confusion_GBM_balanced)

# Create a data frame to store the results
model_performance <- data.frame(
  Model = c("Logistic Regression", "Decision Tree", "Gradient Boosting",
            "Logistic Regression (SMOTE)", "Decision Tree (SMOTE)", "Gradient Boosting (SMOT
E)"),
  TN = c(perf_LR$TN, perf_DT$TN, perf_GBM$TN, perf_LR_balanced$TN, perf_DT_balanced$TN, perf_
GBM_balanced$TN),   FP = c(perf_LR$FP, perf_DT$FP, perf_GBM$FP, perf_LR_balanced$FP,
perf_DT_balanced$FP, perf_
GBM_balanced$FP),   FN = c(perf_LR$FN, perf_DT$FN, perf_GBM$FN, perf_LR_balanced$FN,
perf_DT_balanced$FN, perf_
GBM_balanced$FN),   TP = c(perf_LR$TP, perf_DT$TP, perf_GBM$TP, perf_LR_balanced$TP,
perf_DT_balanced$TP, perf_
GBM_balanced$TP),   Precision = c(perf_LR$Precision, perf_DT$Precision, perf_GBM$Precision,
perf_LR_balanced$Pr ecision, perf_DT_balanced$Precision, perf_GBM_balanced$Precision),   Recall
= c(perf_LR$Recall, perf_DT$Recall, perf_GBM$Recall, perf_LR_balanced$Recall, perf_D
T_balanced$Recall, perf_GBM_balanced$Recall)
)

# Print model performance print(model_performance)
```

```
##                           Model    TN   FP   FN    TP Precision    Recall
## 1         Logistic Regression 56856   42    7    56 0.5714286 0.8888889
## 2               Decision Tree 56845   21   18    77 0.7857143 0.8105263
## 3           Gradient Boosting 56851   27   12    71 0.7244898 0.8554217
## 4 Logistic Regression (SMOTE) 27884 3181  512 25385 0.8886438 0.9802294
## 5       Decision Tree (SMOTE) 27302  891 1094 27675 0.9688091 0.9619730
## 6   Gradient Boosting (SMOTE) 28274  505  122 28061 0.9823216 0.9956711
```

```r
# Plot performance metrics
library(ggplot2)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

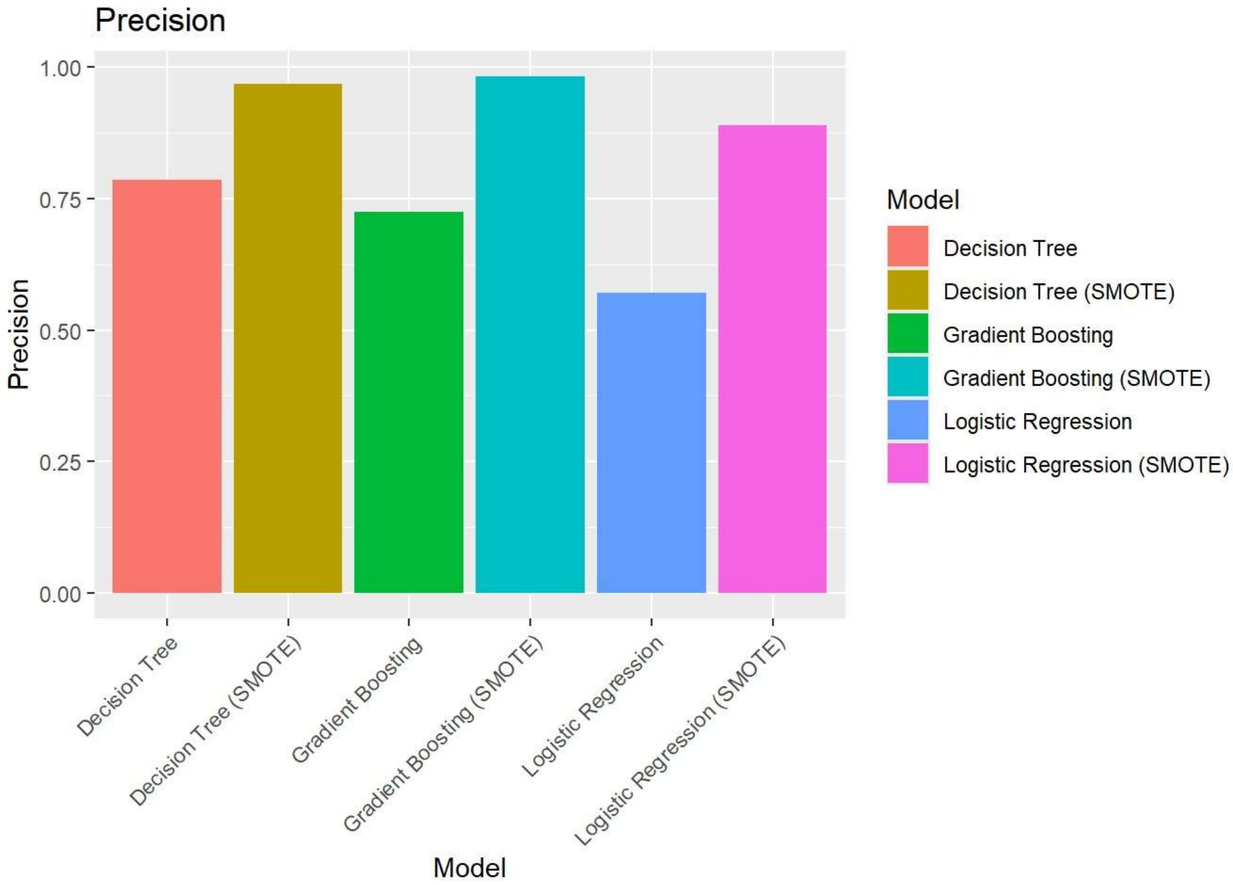
```
## The following object is masked from 'package:neuralnet':
##
##     compute
```

```
## The following objects are masked from 'package:data.table':
##
##     between, first, last
```

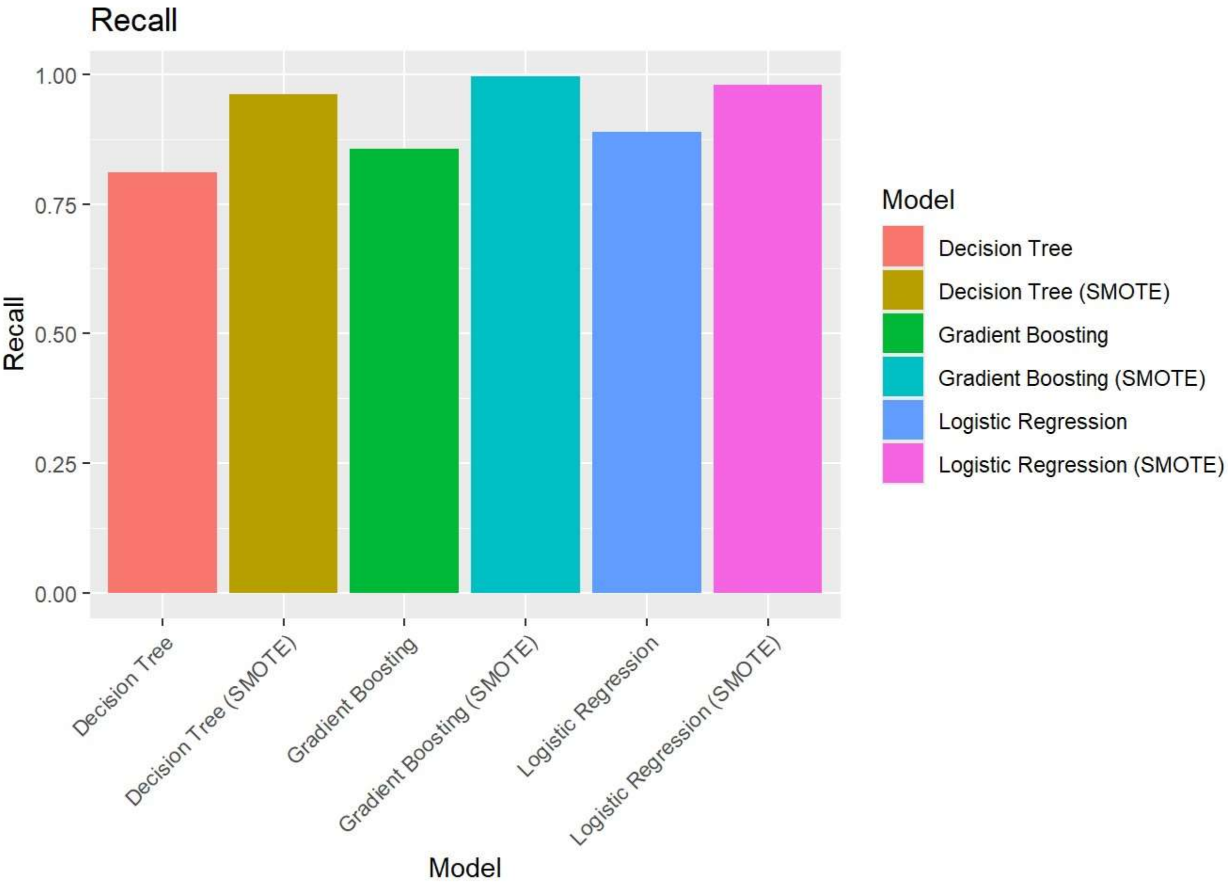
```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
ggplot(model_performance, aes(x = Model, y = Precision, fill = Model)) +
  geom_bar(stat = "identity", position = "dodge") +
  ggtitle("Precision") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



```
ggplot(model_performance, aes(x = Model, y = Recall, fill = Model)) +
  geom_bar(stat = "identity", position = "dodge") +
  ggtitle("Recall") +
  theme(axis.text.x = element_text(angle =  45, hjust = 1))
```


Recall
Recall
1.00
0.75
0.50
0.25
0.00
Decision Tree
Decision Tree (SMOTE)
Gradient Boosting
Gradient Boosting (SMOTE)
Logistic Regression
Logistic Regression (SMOTE)
Model
Model
Decision Tree
Decision Tree (SMOTE)
Gradient Boosting
Gradient Boosting (SMOTE)
Logistic Regression
Logistic Regression (SMOTE)