Northeastern University

Final Project Report

Olanrewaju Olajuyi: olajuyi.o@northeastern.edu

College of Professional Studies Northeastern University
Vancouver, BC

Spring 2024

Instructor: Prof.Ahmadi Abkenari, Fatemeh

May 15, 2024

R-Program: Predictive Modeling and Clustering Analysis for Bank Marketing

1. Introduction

The aim of this project is twofold: to construct predictive models for marketing campaign success prediction and to conduct clustering analysis on banking data. The dataset under examination is the Bank Marketing dataset 'bank-additional-full', which includes client profiles, marketing campaign details, and campaign outcomes. With 41,188 observations and 21 features encompassing categorical, numerical, and integer data, the dataset originates from a Portuguese banking firm's telemarketing sales campaign targeting long-term deposit sales prediction. The primary objective is to refine marketing strategies and decision-making processes. Sérgio Moro, Paulo Cortez, and Paulo Rita employed machine learning algorithms to develop predictive models, assessing their efficacy using metrics like ROC and AUC. The target feature, denoted as 'y', indicates whether a client subscribed ('yes') or not ('no') to the bank's long-term deposit product.

2. Data Exploration

The initial step involved exploring the dataset, understanding its structure, and identifying any missing values. The dataset consists of both numeric and categorical variables. Histograms and bar plots were created to visualize the distribution of numeric and categorical variables, respectively. Additionally, bivariate plots were generated to explore relationships between categorical variables and the target variable.
Below codes are from the data analysis we conducted on bank-additional-full dataset named **bank_df**, which appears to contain information related to banking activities. Let's interpret each part:

```
> str(bank_df)  # Structure of the dataset
'data.frame':   41188 obs. of  21 variables:
 $ age           : int  56 57 37 40 56 45 59 41 24 25 ...
 $ job           : chr  "housemaid" "services" "services" "admin." ...
 $ marital       : chr  "married" "married" "married" "married" ...
 $ education     : chr  "basic.4y" "high.school" "high.school" "basic.6y" ...
 $ default       : chr  "no" "unknown" "no" "no" ...
 $ housing       : chr  "no" "no" "yes" "no" ...
 $ loan          : chr  "no" "no" "no" "no" ...
 $ contact       : chr  "telephone" "telephone" "telephone" "telephone" ...
 $ month         : chr  "may" "may" "may" "may" ...
 $ day_of_week   : chr  "mon" "mon" "mon" "mon" ...
 $ duration      : int  261 149 226 151 307 198 139 217 380 50 ...
 $ campaign      : int  1 1 1 1 1 1 1 1 1 1 ...
 $ pdays         : int  999 999 999 999 999 999 999 999 999 999 ...
 $ previous      : int  0 0 0 0 0 0 0 0 0 0 ...
 $ poutcome      : chr  "nonexistent" "nonexistent" "nonexistent" "nonexistent" ...
 $ emp.var.rate  : num  1.1 1.1 1.1 1.1 1.1 1.1 1.1 1.1 1.1 1.1 ...
 $ cons.price.idx: num  94 94 94 94 94 ...
 $ cons.conf.idx : num  -36.4 -36.4 -36.4 -36.4 -36.4 -36.4 -36.4 -36.4 -36.4 -36.4 ...
 $ euribor3m     : num  4.86 4.86 4.86 4.86 4.86 ...
 $ nr.employed   : num  5191 5191 5191 5191 5191 ...
 $ y             : chr  "no" "no" "no" "no" ...
```

Figure 1.0: Dataset Structure Overview

The **str(bank_df)** command offers a comprehensive overview of the dataset **bank_df**. It reveals that **bank_df** is a data frame comprising 41,188 observations and 21 variables. Each variable is assigned a specific data type, including integers (int), characters (chr), and numerics (num). Notable variables encompass age, job occupation, marital status, education level, contact method, and the duration of each contact, among others. This summary provides valuable insights into the structure and composition of the dataset.

```
> head(bank_df)
  age       job marital   education default housing loan   contact month
1  56 housemaid married    basic.4y      no      no   no telephone   may
2  57  services married high.school unknown      no   no telephone   may
3  37  services married high.school      no     yes   no telephone   may
4  40    admin. married    basic.6y      no      no   no telephone   may
5  56  services married high.school      no      no  yes telephone   may
6  45  services married    basic.9y unknown      no   no telephone   may
  day_of_week duration campaign pdays previous    poutcome emp.var.rate
1         mon      261        1   999        0 nonexistent          1.1
2         mon      149        1   999        0 nonexistent          1.1
3         mon      226        1   999        0 nonexistent          1.1
4         mon      151        1   999        0 nonexistent          1.1
5         mon      307        1   999        0 nonexistent          1.1
6         mon      198        1   999        0 nonexistent          1.1
  cons.price.idx cons.conf.idx euribor3m nr.employed  y
1         93.994         -36.4     4.857        5191 no
2         93.994         -36.4     4.857        5191 no
3         93.994         -36.4     4.857        5191 no
4         93.994         -36.4     4.857        5191 no
5         93.994         -36.4     4.857        5191 no
6         93.994         -36.4     4.857        5191 no
```

Figure 2.0: First Few Rows of the Dataset

The **head(bank_df)** function is employed to present an initial glimpse into the bank_df dataset. It showcases the first six rows of the dataset, unveiling the values of various variables such as age, job occupation, marital status, education level, contact method, duration of contact, and campaign details for each observed individual. This brief preview facilitates an initial understanding of the dataset's contents and structure.

```
> colnames(bank_df)
 [1] "age"            "job"            "marital"        "education"
 [5] "default"        "housing"        "loan"           "contact"
 [9] "month"          "day_of_week"    "duration"       "campaign"
[13] "pdays"          "previous"       "poutcome"       "emp.var.rate"
[17] "cons.price.idx" "cons.conf.idx"  "euribor3m"      "nr.employed"
[21] "y"
```

Figure 3.0: Column Names of the Dataset

The **colnames(bank_df)** function retrieves the column names (variables) present in the **bank_df** dataset. It furnishes a comprehensive list of column identifiers, encompassing descriptors like age, job occupation, marital status, education level, contact method, duration of contact, campaign details, previous contact metrics, economic indicators, and the outcome variable denoted as 'y'. This output offers a succinct overview of the dataset's structural components, aiding in subsequent data manipulation and analysis tasks.

Overall, these codes are essential for exploring and understanding the structure, content, and variables of the dataset **bank_df**, which is crucial for our data analysis and modeling tasks.

Exploring Numeric Variable Distribution:
The plot visualizes the distribution of numeric variables in the dataset bank_df. Each histogram represents one numeric variable, including age, duration, campaign, pdays, previous, emp.var.rate, cons.price.idx, euribor3m, nr.employed, and cons.conf.idx. The x-axis of each histogram denotes the range of values for the corresponding variable, while the y-axis indicates the frequency or count of observations falling within each range.

Insights from Distribution Patterns:
The distribution of each variable provides insights into its central tendency, spread, and potential outliers. For instance, histograms with a skewed shape suggest asymmetry in the data distribution, while those with a symmetrical bell shape indicate a more uniform distribution. The color scheme, with histograms shaded in sky blue and white borders, enhances visibility, and distinguishes between the different plots. Overall, this visualization allows for a quick assessment of the numeric variables' distribution patterns, aiding in exploratory data analysis and identifying potential data characteristics.

- **Numeric Variable Distribution Insights**

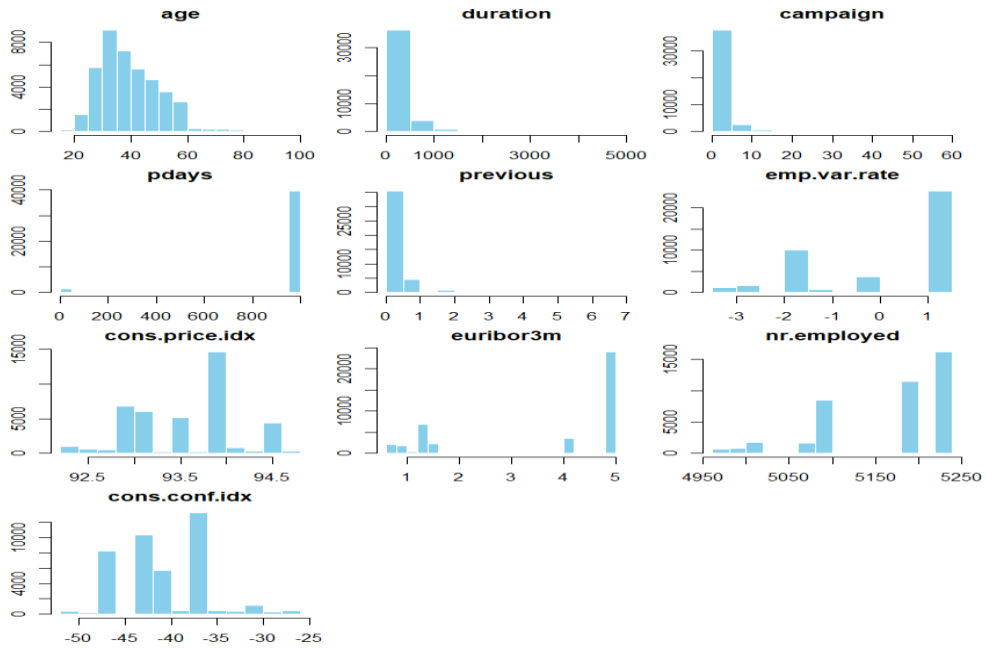


Fig 4: Exploring Numeric Variable Distribution in bank_df Dataset

3. Data Preprocessing

Data preprocessing steps included handling missing values, dealing with multicollinearity, and encoding categorical variables. Unknown values in categorical features such as marital status, housing, and education were removed or reclassified. Numeric variables with high correlation were identified and removed to address multicollinearity issues. Categorical variables were converted to factors for modeling purposes.

- **Exploring Categorical Variables: Frequency Distribution Analysis**

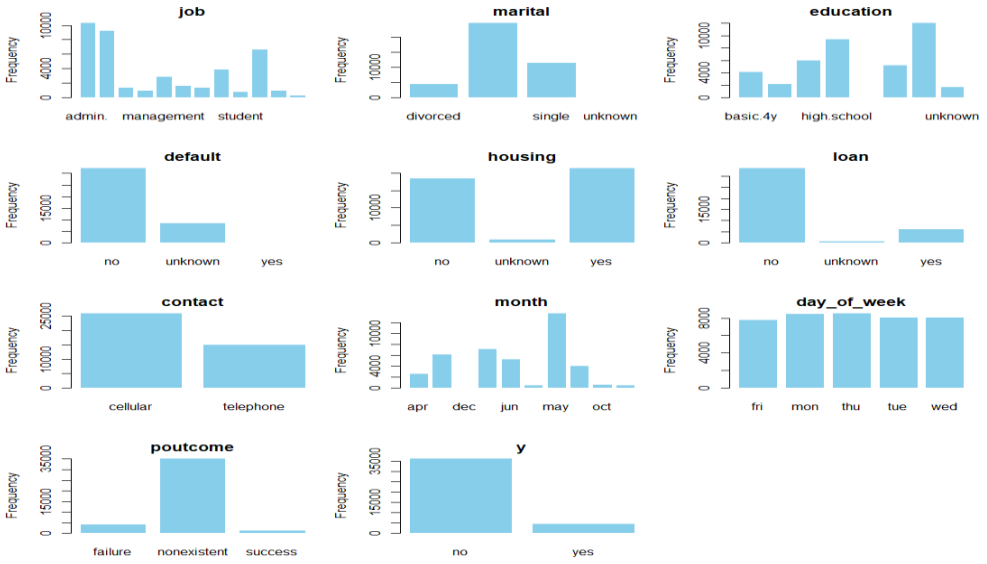


Fig 5: Frequency Distribution of Categorical Variables in bank_df

The bar charts generated for each categorical variable in the dataset **bank_df**, covering aspects such as job, marital status, education, default status, housing loan status, personal loan status, contact method, month of last contact, day of the week of last contact, previous campaign outcome, and the target variable 'y' indicating subscription to the bank's long-term deposit product. Each chart visualizes the frequency distribution of categories within a specific variable. The x-axis represents the categories of the variable, while the y-axis denotes the frequency of each category.

- **Analyzing Correlations: Heatmap Visualization**

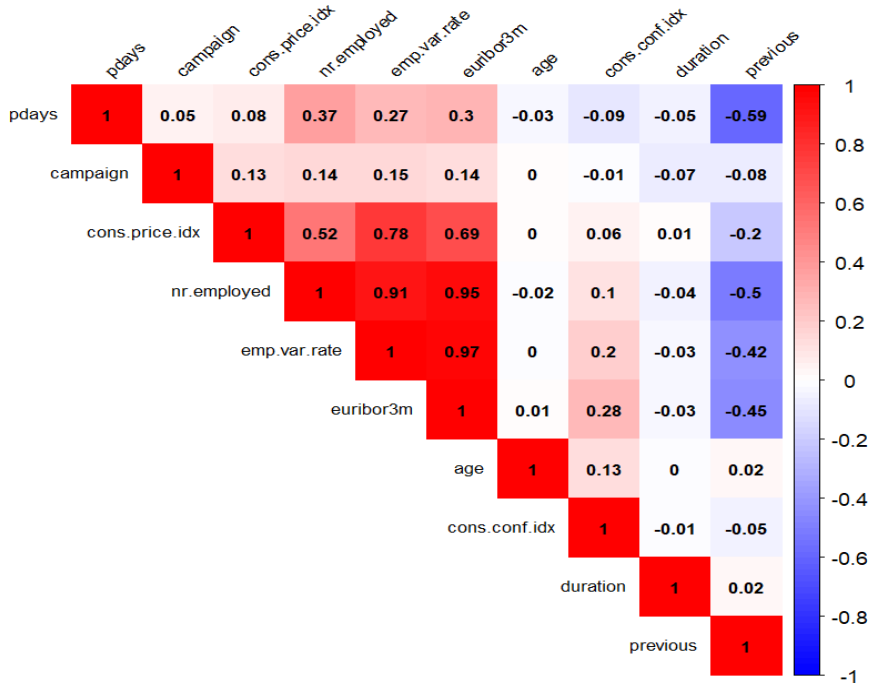


Fig 6: Exploring Correlation Patterns: Heatmap Representation

We compute a correlation matrix in **Fig 6** for variables within the **bank_subset** dataset and generates a heatmap visualization using the **corrplot** function. The heatmap provides a clear graphical representation of the relationships between variables, with warmer colors indicating positive correlations and cooler colors denoting negative correlations. By examining the heatmap, analysts can quickly identify strong correlations and potential multicollinearity issues, aiding in feature selection and model building. This visual approach enhances understanding of complex data interdependencies and facilitates more informed decision-making in predictive analytics.

- **Correlation Analysis and Heatmap Visualization after Removing Highly Correlated Variables**

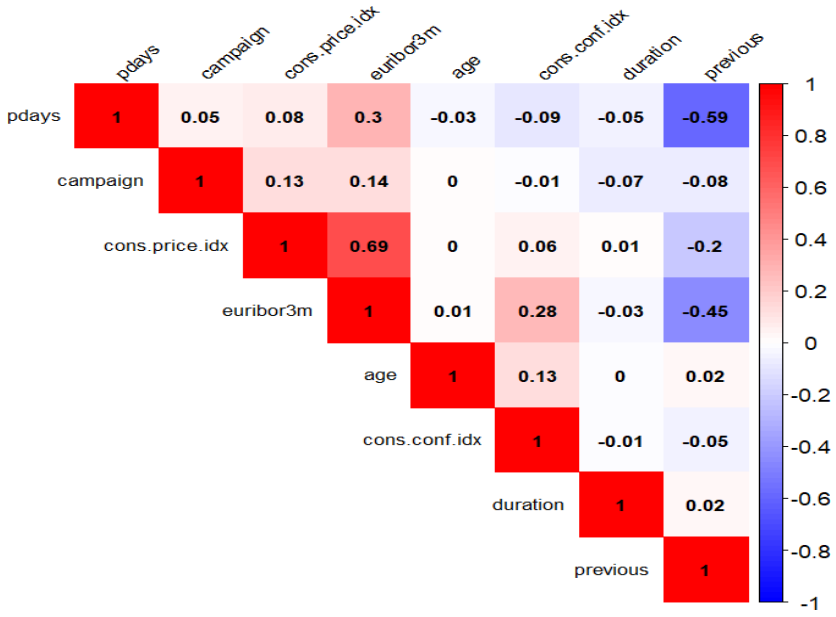


Fig 7: Correlation Analysis Heatmap: Subset of Numerical Variables

We performed a correlation analysis in **Fig 7** after removing variables with high correlation from the dataset. Initially, it selects a subset of numerical columns, including features such as age, duration, campaign, pdays, previous, cons.price.idx, euribor3m, and cons.conf.idx. Then, it subsets the dataset 'bank_df' to include only these selected numerical variables. Next, it computes the correlation matrix for the subset of data. Finally, it generates a heatmap using the 'corrplot' function, visualizing the correlation matrix. The heatmap employs a color scheme ranging from blue (negative correlation) to red (positive correlation), with white indicating no correlation. This visualization aids in identifying correlated variables and understanding the relationships between them.

- **Data Preprocessing Steps in bank_df Dataset**

```
> # Remove category "Unknown" from Marital status
> # Filter out rows where 'marital' is not 'unknown'
> marital_fil <- bank_df[bank_df$marital != 'unknown', ]
> # Compute value counts of 'marital' column
> marital_counts <- table(marital_fil$marital)
> # Display the value counts
> print(marital_counts)

divorced  married   single
    4612    24928    11568
> # Remove category "Unknown" from 'housing'
> # Filter out rows where 'housing' is not 'unknown'
> housing_fil <- bank_df[bank_df$housing != 'unknown', ]
> # Compute value counts of 'marital' column
> housing_counts <- table(housing_fil$housing)
> # Display the value counts
> print(housing_counts)

   no   yes
18622 21576
> # Create a new category called "basic.education" by replacing the values
'basic.4y', 'basic.6y' and 'basic.9y'
> # Replace values in 'education' column
> bank_df$education <- ifelse(bank_df$education %in% c('basic.4y', 'basic.6
y', 'basic.9y'),
+                             'basic.education', bank_df$education)
> # Display the value counts of 'education' column
> table(bank_df$education)

    basic.education         high.school          illiterate professional.co
urse
              12513                9515                  18
5243
  university.degree             unknown
              12168                1731
> # Remove category "Unknown" from 'housing'
> # Filter out rows where 'housing' is not 'unknown'
> education_fil <- bank_df[bank_df$education != 'unknown', ]
> # Compute value counts of 'marital' column
> education_counts <- table(education_fil$education)
> # Display the value counts
> print(education_counts)

    basic.education         high.school          illiterate professional.co
urse
              12513                9515                  18
5243
  university.degree
              12168
```

Fig 8: Data Preprocessing Steps: Marital, Housing, and Education Variables

1. **Marital Status Filtering:**

- Initially, the code removes the category "Unknown" from the marital status variable.
- It filters out rows where the marital status is not "unknown" using logical indexing.
- After filtering, it computes the value counts of different marital statuses.
- The result shows that there are 4,612 divorced individuals, 24,928 married individuals, and 11,568 single individuals.

2. **Housing Status Filtering:**
   - Next, the code removes the category "Unknown" from the housing status variable.
   - Similar to the previous step, it filters out rows where the housing status is not "unknown".
   - After filtering, it computes the value counts of different housing statuses.
   - The result indicates that there are 18,622 individuals without housing loans and 21,576 individuals with housing loans.
3. **Education Variable Transformation:**
   - The code creates a new category called "basic.education" by replacing the values 'basic.4y', 'basic.6y', and 'basic.9y' with this new category.
   - It replaces these values in the education column using conditional logic.
   - Subsequently, it displays the value counts of the education column after the transformation.
   - The result demonstrates that there are 12,513 individuals with basic education, 9,515 with high school education, 18 illiterate individuals, 5,243 with a professional course, 12,168 with a university degree, and 1,731 individuals with an unknown education status.

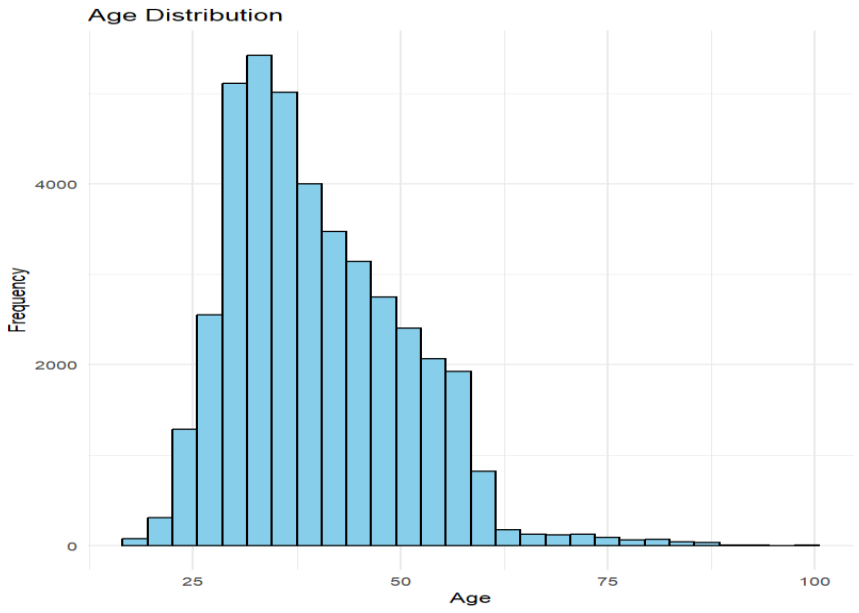


Fig 9: Age Distribution Histogram

We generated a histogram in **Fig 9** to visualize the distribution of ages in the dataset 'bank_df'. Each bar represents a range of ages, and the height of the bar indicates the frequency of individuals within that age range. The histogram is shaded in sky blue with black borders for better visibility. The x-axis represents the age, while the y-axis denotes the frequency of occurrences. This visualization provides insights into the distribution of ages within the dataset.

- **Converting Categorical Variables to Factors in bank_df Dataset**

```
> # Converting all categorical variables to a factor
> bank_df$loan <- as.factor(bank_df$loan)
> bank_df$contact <- as.factor(bank_df$contact)
> bank_df$month <- as.factor(bank_df$month)
> bank_df$job <- as.factor(bank_df$job)
> bank_df$marital <- as.factor(bank_df$marital)
> bank_df$education <- as.factor(bank_df$education)
> bank_df$default <- as.factor(bank_df$default)
> bank_df$housing <- as.factor(bank_df$housing)
> bank_df$day_of_week <- as.factor(bank_df$day_of_week)
> bank_df$poutcome <- as.factor(bank_df$poutcome)
```

Fig 10: Converting Categorical Variables to Factors

This code in figure 5.0 converts all categorical variables in the dataset 'bank_df' into factor data types. Categorical variables, such as 'loan', 'contact', 'month', 'job', 'marital', 'education', 'default', 'housing', 'day_of_week', and 'poutcome', are converted to factors using the 'as.factor()' function. Converting categorical variables to factors is essential for categorical data analysis and modeling in R, as it allows for efficient handling and interpretation of categorical variables in statistical analyses and machine learning algorithms.

- **Exploring Target Variable Distribution**

Fig 11:Distribution of Target Variable 'y'

The bar plot in **Fig 11** visualize the distribution of the target variable 'y'. The x-axis represents the values of 'y', which indicate whether clients subscribed ('yes') or not ('no') to the bank's long-term deposit product. The y-axis shows the frequency of each category. Each bar represents the count of observations corresponding to each category. The plot's title is "Distribution of Target Variable 'y'", and it is styled with a minimal theme. The bars are filled with sky blue color and outlined in black. This visualization provides insight into the balance between the number of clients who subscribed to the product and those who did not.

4. Model Implementation

The random seed is set at 42 to ensure reproducibility of the results. The dataset is then split into training and testing sets using a 70-30 split ratio, with 70% of the data allocated to the training set

and 30% to the testing set. This split is performed based on the target variable 'y' from the bank_df dataset. Subsequently, the training set ('train_bx') and testing set ('test_bx') are created by subsetting the original dataset according to the split.

- **Train Set Class Distribution**

```
> set.seed(42)
> # Split the data into training and testing sets (70% train, 30% test)
> sample_split <- sample.split(Y = bank_df$y, SplitRatio = 0.70)
> train_bx <- subset(x = bank_df, sample_split == TRUE)
> test_bx <- subset(x = bank_df, sample_split == FALSE)
> # Checking the imbalance state of the target variable
> # Check table
> table(train_bx$y)

   no   yes
25584  3248
> # Check classes distribution
> prop.table(table(train_bx$y))

       no       yes
0.8873474 0.1126526
```

Figure 12: Data Splitting and Imbalance Check

Following this, an imbalance check is conducted on the target variable within the training set. The table function is utilized to display the count of 'yes' and 'no' values in the 'y' variable, revealing that the majority class ('no') significantly outweighs the minority class ('yes'). Additionally, the prop.table function is employed to compute the proportion of each class within the training set, indicating that approximately 88.7% of the instances belong to the 'no' class and 11.3% belong to the 'yes' class. This analysis highlights the class imbalance present in the training data, which is a crucial consideration for subsequent model training and evaluation.

Several machine learning models were implemented for predicting campaign success:

A. **Decision Trees:** Initially, decision tree models were built without addressing class imbalance.

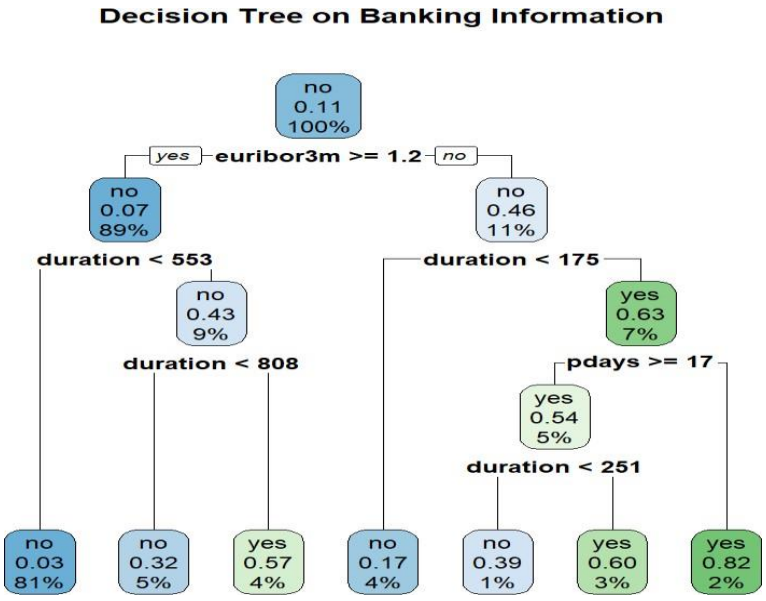


Fig 13: Decision Tree Model on Banking Information 1

A decision tree model is constructed based on the imbalanced distribution of classes within the dataset. Upon splitting the data into training and testing sets, it's observed that the majority class ('no') significantly outweighs the minority class ('yes') in the training set, with a distribution of

approximately 89% to 11%. This imbalance highlights the disparity in the number of observations between the two classes, potentially leading to biased model predictions. Despite the class imbalance, the decision tree model is built using the rpart function, aiming to capture patterns and relationships within the data. The resulting decision tree plot provides a visual representation of the model's decision-making process, offering insights into the key features influencing the classification outcomes.

- **Feature Importance Analysis and Prediction Results**

```
> importanceBX <- varImp(model_DT)
> importanceBX %>% arrange(desc(Overall))
                  Overall
duration       1908.56743
euribor3m       928.29459
pdays           815.27116
poutcome        791.70686
cons.conf.idx   491.03443
cons.price.idx  181.30153
month            96.64668
previous         30.59604
contact          18.52422
age               0.00000
job               0.00000
marital           0.00000
education         0.00000
default           0.00000
housing           0.00000
loan              0.00000
day_of_week       0.00000
campaign          0.00000
```

Figure 14: Importance Ranking of Predictive Features

The code segment assesses feature importance using the variable importance metric obtained from the decision tree model (**model_DT**). It ranks the variables based on their overall importance in predicting the target variable. The feature **duration** emerges as the most significant predictor, followed by **euribor3m**, **pdays**, and **poutcome**. Other variables such as **cons.conf.idx**, **cons.price.idx**, **month**, **previous**, and **contact** also contribute to the model's predictive power. Interestingly, the features **age**, **job**, **marital**, **education**, **default**, **housing**, **loan**, **day_of_week**, and **campaign** show no importance in this context.

```
> # Predict the model based on the test set
> test_bf <- test_bx  # Assuming test_bf is meant to be test_bx
> preds_DT <- predict(model_DT, newdata = test_bf, type = "class")
> head(preds_DT, 10)
 1  2  4  7 10 12 13 16 17 21
no no no no no no no no no no
Levels: no yes
```

Figure 15: Prediction Results: Decision Tree Model Predictions on Test Data

After assessing feature importance, the model (**model_DT**) predicts the target variable (**y**) using the test dataset (**test_bx**). The prediction results (**preds_DT**) indicate that the model predominantly predicts **"no"** responses for the first ten observations in the test set.

- **Model Performance Evaluation: Confusion Matrix**

```
> # Create a confusion matrix
> conf_matrix <- confusionMatrix(preds_DT, test_bx$y)
> print(conf_matrix)
Confusion Matrix and Statistics

          Reference
Prediction    no   yes
       no  10543   717
       yes   421   675

               Accuracy : 0.9079
                 95% CI : (0.9027, 0.9129)
    No Information Rate : 0.8873
    P-Value [Acc > NIR] : 5.982e-14

                  Kappa : 0.4922

 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.9616
            Specificity : 0.4849
         Pos Pred Value : 0.9363
         Neg Pred Value : 0.6159
             Prevalence : 0.8873
         Detection Rate : 0.8533
   Detection Prevalence : 0.9113
      Balanced Accuracy : 0.7233

       'Positive' Class : no
```

Figure 16: Confusion Matrix Result 1

The confusion matrix in figure 9.0 provides insights into the performance of the decision tree model. It shows that out of 11,260 instances, the model correctly predicted 10,543 instances of the negative class ('no') and 675 instances of the positive class ('yes'). However, it misclassified 717 instances of the positive class as negative and 421 instances of the negative class as positive. This results in an overall accuracy of approximately 90.79%, indicating the proportion of correctly classified instances. Additionally, the Kappa statistic, which measures the agreement between observed and predicted classes, is 0.4922, suggesting moderate agreement beyond chance. Sensitivity, also known as the true positive rate, is 96.16%, indicating the model's ability to correctly identify positive instances. However, the specificity, representing the true negative rate, is relatively lower at 48.49%, suggesting a higher rate of false positives. Overall, the balanced accuracy, which considers sensitivity and specificity, is 72.33%, indicating a moderately reliable model performance.

B. **Handling Imbalance:** Techniques like oversampling minority class, undersampling majority class, and combined sampling methods were employed to address class imbalance.

- **Addressing the data imbalance in the target variable 'y' using oversampling and undersampling techniques.**

```
> library(ROSE)
Loaded ROSE 0.0-4

Warning message:
package 'ROSE' was built under R version 4.3.3
> # Over-sample the minority class
> df_oversampled <- ovun.sample(y ~ ., data = train_b
x, method = "over", N = 2 * table(train_bx$y)[[1]], s
eed = 42)$data
> # Check the new class distribution
> table(df_oversampled$y)

   no   yes
25584 25584
> # Under sample the majority class
> class(bank_df$y)
[1] "character"
> # Convert 'y' to a factor variable
> train_bx$y <- as.factor(train_bx$y)
> #Perform undersampling on the train set
> df_undersampled <- downSample(x = train_bx[, -which
(names(train_bx) == "y")],
+                               y = train_bx$y,
+                               list = FALSE)$x
> # Combine the undersampled data with the original m
inority class
> df_balanced <- rbind(df_undersampled, train_bx[trai
n_bx$y == "yes", ])
> # Check the new class distribution
> table(df_balanced$y)

  no  yes
   0 3248
> #balance both samples of the train set
> data_balanced_both <- ovun.sample(y ~ ., data = tra
in_bx, method = "both", p=0.5, seed = 1)$data
> table(data_balanced_both$y)

   no   yes
14517 14315
> # Check classes distribution again
> prop.table(table(data_balanced_both$y))

       no       yes
0.5035031 0.4964969
> #using the function Rose to correct any inconsisten
cy from the oversampling and undersampling data
> data.rose <- ROSE(y ~ ., data =train_bx, seed = 1)$
data
> table(data.rose$y)

   no   yes
14517 14315
```

Figure 17: Class Distribution After Balancing

We utilize the ROSE package in figure 10.0 to address data imbalance by employing oversampling and undersampling techniques on the training set's target variable 'y'. Initially, the minority class is oversampled to match the frequency of the majority class, resulting in **25,584** instances for both 'yes' and 'no' categories. Subsequently, the majority class is undersampled to achieve a balanced distribution, reducing the 'no' instances to 3,248 while retaining all 'yes' instances.

The combined dataset now exhibits a balanced class distribution with **50.35%** 'no' instances and **49.65%** 'yes' instances. Finally, the ROSE function is applied to correct any inconsistencies arising from oversampling and undersampling, resulting in a balanced dataset with **14,517** instances for both 'no' and **14315** 'yes' categories.

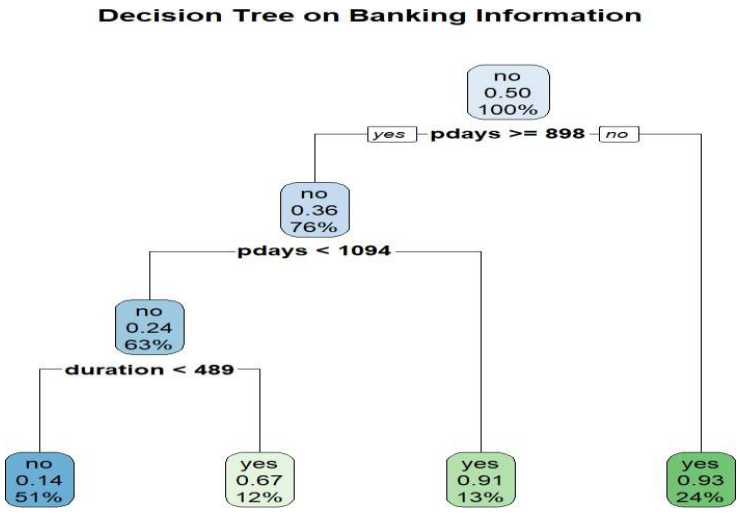


Fig 18: Decision Tree Model on Banking Information 2

The code trains a decision tree model using the dataset obtained after balancing through the ROSE function. Subsequently, it predicts on the combined data using this model and evaluates the balance by generating a confusion matrix. The confusion matrix indicates the count of correct and incorrect predictions for each class (yes and no) of the target variable 'y'. In this matrix**, 9,902** instances of 'no' were correctly predicted, while **516** instances of 'yes' were incorrectly predicted as 'no'.

Additionally**, 1,062** instances of 'yes' were incorrectly predicted as 'no', and 876 instances of 'yes' were correctly predicted. This assessment provides insights into the model's performance after addressing the data imbalance issue. Finally, the code generates a plot visualizing the decision tree model's structure based on the banking information dataset.

```
> importanceBF <- varImp(tree_model)
> importanceBF %>% arrange(desc(Overall))
                  Overall
duration        7249.6555
pdays           6299.0632
euribor3m       4728.8925
previous        3379.7622
month           1320.7599
poutcome        1259.2127
cons.price.idx   260.8957
age                0.0000
job                0.0000
marital            0.0000
education          0.0000
default            0.0000
housing            0.0000
loan               0.0000
contact            0.0000
day_of_week        0.0000
campaign           0.0000
cons.conf.idx      0.0000
```

Figure 19: Variable Importance Analysis for Decision Tree Model

The code in figure 11.0 calculates the variable importance based on the trained decision tree model. The variable importance values represent the contribution of each feature to the model's predictive performance. The results are arranged in descending order of importance. The variables with higher importance values are considered more influential in making predictions. In this case, 'duration', 'pdays', and 'euribor3m' are identified as the most important features, indicating that they have a significant impact on the model's decision-making process. Conversely, features such as 'age', 'job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'day_of_week', 'campaign', and 'cons.conf.idx' have negligible importance, implying that they contribute less to the model's predictive power. These insights help prioritize features for further analysis and model refinement.

```
> # Create a confusion matrix to evaluate the performance of the model again
> conf_matrix_balanced <- confusionMatrix(pred_dt2, test_bx$y)
> print(conf_matrix_balanced)
Confusion Matrix and Statistics

          Reference
Prediction   no  yes
       no  9902  516
       yes 1062  876

               Accuracy : 0.8723
                 95% CI : (0.8663, 0.8781)
    No Information Rate : 0.8873
    P-Value [Acc > NIR] : 1

                  Kappa : 0.4546

 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.9031
            Specificity : 0.6293
         Pos Pred Value : 0.9505
         Neg Pred Value : 0.4520
             Prevalence : 0.8873
         Detection Rate : 0.8014
   Detection Prevalence : 0.8432
      Balanced Accuracy : 0.7662

       'Positive' Class : no
```

Figure 20: Confusion Matrix Result 2

The confusion matrix in figure 12.0 provides insights into the performance of the decision tree model. It compares the predicted values against the actual values of the target variable 'y'. Here, 'no' and 'yes' represent the two classes of the target variable. The accuracy of the model is **87.23%,** indicating the proportion of correct predictions. The Kappa coefficient, measuring the agreement between predicted and actual values, is **0.4546,** suggesting moderate agreement. Sensitivity, also

known as the true positive rate, is **90.31%,** indicating the proportion of actual 'yes' cases correctly predicted by the model. Specificity, or the true negative rate, is **62.93%,** showing the proportion of actual 'no' cases correctly predicted. The balanced accuracy, considering both sensitivity and specificity, is **76.62%.** These metrics collectively assess the model's ability to classify instances correctly and provide a comprehensive evaluation of its performance.

- **Table 1.0: Comparison both Confusion Matrix**

Metric	Imbalanced Approach	Balanced Approach
Accuracy	90.79%	87.23%
Kappa	0.4922	0.4546
Sensitivity	96.16%	90.31%
Specificity	48.49%	62.93%
Balanced Accuracy	72.33%	76.62%

This comparison highlights in table 1.0 the impact of balancing techniques on model performance metrics. In the imbalanced approach, the model achieved higher sensitivity (96.16%) but lower specificity (48.49%) compared to the balanced approach, which yielded a higher specificity (62.93%) but lower sensitivity (90.31%). Overall, balancing the dataset improved the model's ability to correctly classify both positive and negative instances, as evidenced by the increased balanced accuracy (76.62% vs. 72.33%). However, it slightly reduced the overall accuracy of the model (87.23% vs. 90.79%). This trade-off between sensitivity and specificity underscores the importance of considering both metrics when evaluating classification models, especially in scenarios with imbalanced datasets.

- **Create a confusion matrix to evaluate the performance of the model again**

```
> # Create a confusion matrix to evaluate the performance of the model again
> conf_matrix_balanced2 <- confusionMatrix(pred_bal, test_bx$y)
> print(conf_matrix_balanced2)
Confusion Matrix and Statistics

          Reference
Prediction   no  yes
       no  8899  137
       yes 2065 1255

               Accuracy : 0.8218
                 95% CI : (0.8149, 0.8285)
    No Information Rate : 0.8873
    P-Value [Acc > NIR] : 1

                  Kappa : 0.4445

 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.8117
            Specificity : 0.9016
         Pos Pred Value : 0.9848
         Neg Pred Value : 0.3780
             Prevalence : 0.8873
         Detection Rate : 0.7202
   Detection Prevalence : 0.7313
      Balanced Accuracy : 0.8566

       'Positive' Class : no
```

Figure 21: Confusion Matrix: Model Evaluation after Balancing with ROSE Function

The confusion matrix evaluates the performance of the model after applying balancing techniques. It compares predicted values against actual values of the target variable 'y', with 'no' and 'yes' representing the two classes. The model correctly predicted 8,899 instances of 'no' and 1,255 instances of 'yes'. However, it misclassified 2065 instances of 'no' as 'yes' and 137 instances of 'yes' as 'no'. The accuracy of the model is approximately 82.18%, indicating the proportion of correct predictions. The Kappa coefficient, measuring the agreement between predicted and actual values,

is 0.4445, suggesting moderate agreement. Sensitivity, representing the true positive rate, is 81.17%, indicating the proportion of actual 'yes' cases correctly predicted by the model. Specificity, the true negative rate, is 90.16%, showing the proportion of actual 'no' cases correctly predicted. The balanced accuracy, considering both sensitivity and specificity, is 85.66%. These metrics collectively assess the model's ability to classify instances correctly and provide a comprehensive evaluation of its performance.

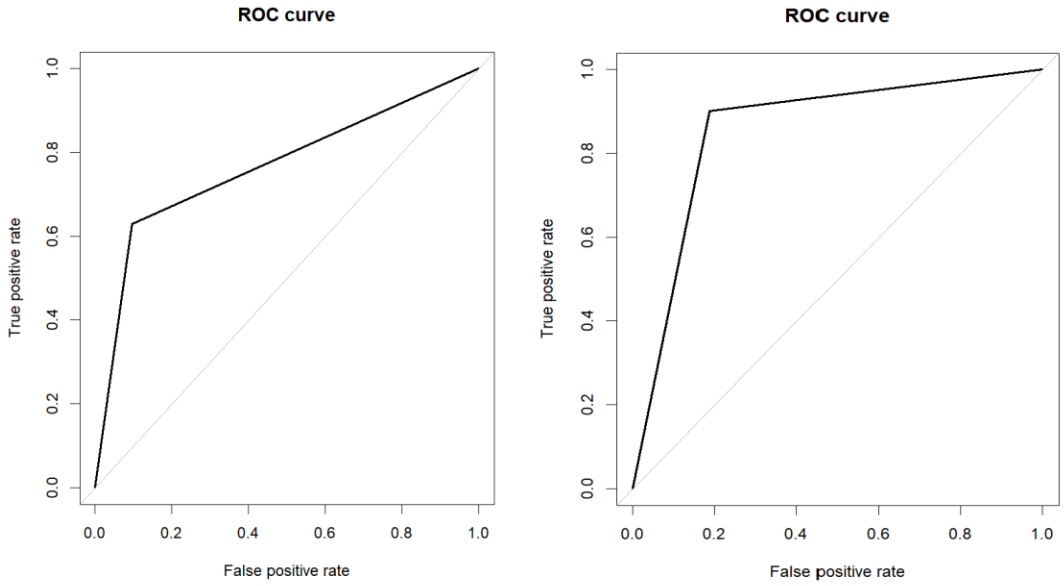


Fig 22: Comparison of ROC Curves: ROSE vs. Combined Techniques

The AUC (Area Under the Curve) is a metric used to evaluate the performance of a classification model based on its ROC (Receiver Operating Characteristic) curve. In the first scenario, where the model was trained using the ROSE technique, the AUC is 0.766. This indicates a moderate level of performance in distinguishing between the positive and negative classes. However, in the second scenario, where a combined technique was utilized, the AUC increases significantly to 0.857. This suggests that the combined technique resulted in a model with better discriminatory power, as it achieved a higher AUC value. Therefore, the combined technique appears to be more effective in this context for improving the model's performance.

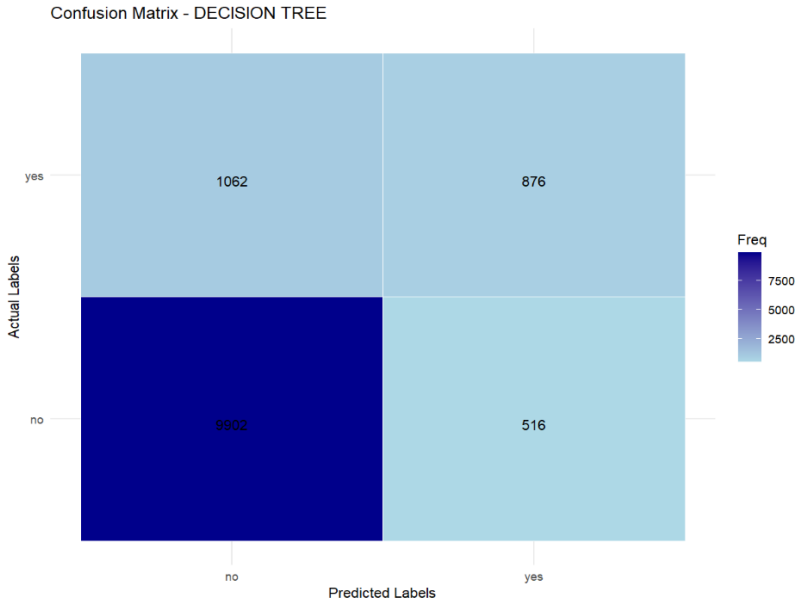


Fig 23: Confusion Matrix Visualization for Decision Tree Model

Fig 23 visualizes a confusion matrix for a decision tree model. Each cell in the matrix represents the count of observations where the actual label (y-axis) matches the predicted label (x-axis). The color intensity of each cell indicates the frequency of observations. The accuracy of the model, calculated from the confusion matrix, is approximately 87.23%, indicating the proportion of correctly classified instances.

c. **Naive Bayes:** A Naive Bayes classifier was trained on balanced data using ROSE technique.

- **Summary of Naive Bayes Model Training**

```
> # Install and load the 'naivebayes' package
> # install.packages("naivebayes")
> library(naivebayes)
> # Train Naive Bayes model on the balanced data 'data.rose' with Laplace smoothing
> nb_model <- naive_bayes(y ~ ., data = data.rose, laplace = 1)
> # Display summary of the Naive Bayes model
> summary(nb_model)

============================================== Naive Bayes ========================
======

- Call: naive_bayes.formula(formula = y ~ ., data = data.rose, laplace = 1)
- Laplace: 1
- Classes: 2
- Samples: 28832
- Features: 18
- Conditional distributions:
    - Bernoulli: 1
    - Categorical: 9
    - Gaussian: 8
- Prior probabilities:
    - no: 0.5035
    - yes: 0.4965

-------------------------------------------------------------------------------
------
```

Figure 24: Naive Bayes Model Summary

This output in figure 14.0 provides a summary of the Naive Bayes model trained on the balanced data using Laplace smoothing. Here's the interpretation:

- **Call**: It shows the function call used to create the Naive Bayes model, indicating the formula, dataset, and Laplace smoothing parameter.
- **Laplace**: The Laplace parameter value used for smoothing, which helps handle zero probabilities in the data.
- **Classes**: The number of classes in the target variable. In this case, there are two classes: 'no' and 'yes'.
- **Samples**: Total number of samples or instances in the dataset used for training the model.
- **Features**: Number of features or predictors used in the model. In this case, there are 18 features.
- **Conditional distributions**: It specifies the types of conditional probability distributions assumed for each feature:
    - Bernoulli: 1 feature
    - Categorical: 9 features
    - Gaussian: 8 features
- **Prior probabilities**: It shows the prior probabilities of each class ('no' and 'yes') estimated from the training data. In this case, the probability of class 'no' is approximately 0.5035, and the probability of class 'yes' is approximately 0.4965.

This summary provides insights into how the Naive Bayes model was trained and the distribution of features and classes in the dataset.

- **Performance Evaluation of Naive Bayes Model**

The confusion matrix offers a detailed assessment of the Naive Bayes model's performance by juxtaposing its predictions against the actual outcomes of the target variable 'y'. Here, 'no' and 'yes' denote the two classes within the target variable. With an accuracy of around 85.86%, the model demonstrates its capability to make correct predictions. Moreover, the Kappa coefficient, gauging the concordance between predicted and observed values, registers at 0.4185, indicating a moderate level of agreement.

```
> # Report the confusion matrix on the 'nb_model'
> conf_matrix_nb <- confusionMatrix(pred_nb, test_bx$y)
> print(conf_matrix_nb)
Confusion Matrix and Statistics

          Reference
Prediction   no  yes
       no  9746  529
       yes 1218  863

               Accuracy : 0.8586
                 95% CI : (0.8523, 0.8647)
    No Information Rate : 0.8873
    P-Value [Acc > NIR] : 1

                  Kappa : 0.4185

 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.8889
            Specificity : 0.6200
         Pos Pred Value : 0.9485
         Neg Pred Value : 0.4147
             Prevalence : 0.8873
         Detection Rate : 0.7888
   Detection Prevalence : 0.8316
      Balanced Accuracy : 0.7544

       'Positive' Class : no
```

Figure 25.0: Confusion Matrix: Evaluating Naive Bayes Model Performance

In terms of class-specific metrics, the model in figure 15.0 exhibits a sensitivity of 88.89%, signifying its adeptness in correctly identifying instances of the 'yes' class. Conversely, the specificity rate stands at 62.00%, showcasing the model's effectiveness in accurately recognizing instances of the 'no' class. Overall, the balanced accuracy, which harmonizes sensitivity and specificity, stands at 75.44%. These performance metrics collectively provide a comprehensive evaluation of the model's efficacy in classifying instances within the dataset.

- **Evaluation of Naive Bayes Model Performance**

The Naive Bayes model achieves an accuracy of approximately 85.86%, correctly predicting 9,746 instances of the negative class ('no') and 863 instances of the positive class ('yes'), with sensitivity at 88.89% and specificity at 62.00%.

- Accuracy: “0.858611201035934"

Table 2.0: Confusion matrix Naive Bayes

pred_nb	no	yes
no	9746	529
yes	1218	863

The table 2.0 presented above shows the confusion matrix derived from evaluating the Naive Bayes model's performance. It provides a breakdown of predicted versus actual labels for the target variable 'y'. In the matrix, the rows represent the predicted labels, while the columns denote the actual labels. For instance, the cell at the intersection of "no" prediction and "yes" actual label indicates instances where the model predicted "no" but the actual label was "yes". Similarly, the

cell at the intersection of "yes" prediction and "no" actual label indicates instances where the model predicted "yes" but the actual label was "no".

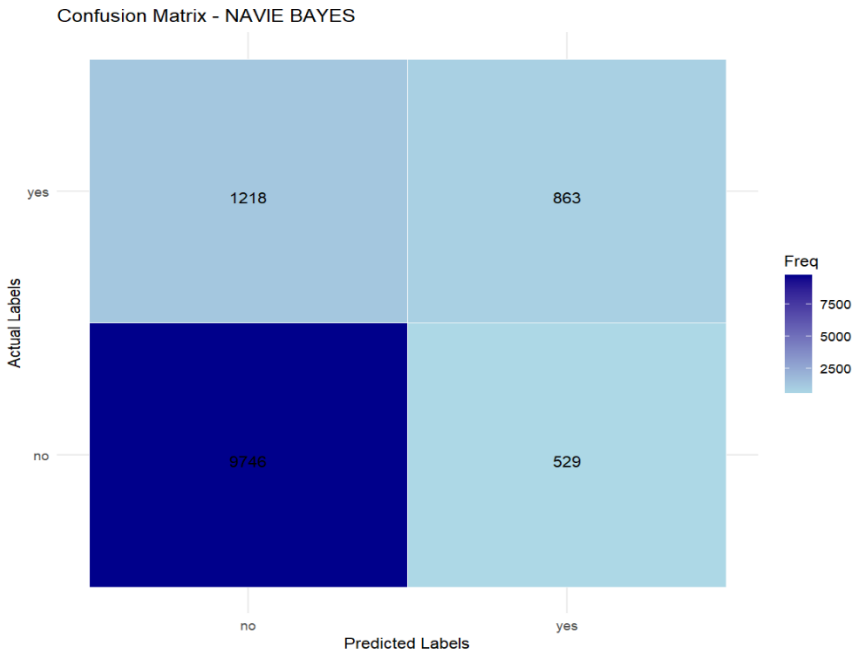


Fig 26.0: Confusion Matrix Visualization NB

To visualize the confusion matrix, a heatmap plot has been generated in **Fig 26**. This plot visually represents the confusion matrix, with color intensity indicating the frequency of instances in each cell. The x-axis represents the predicted labels, while the y-axis represents the actual labels. The darker shades of blue signify higher frequencies, whereas lighter shades denote lower frequencies. This visualization aids in comprehending the model's performance in classifying instances into different categories, providing a clear overview of its strengths and weaknesses.

D. **SVM:** The balanced datais used to compare their performance.

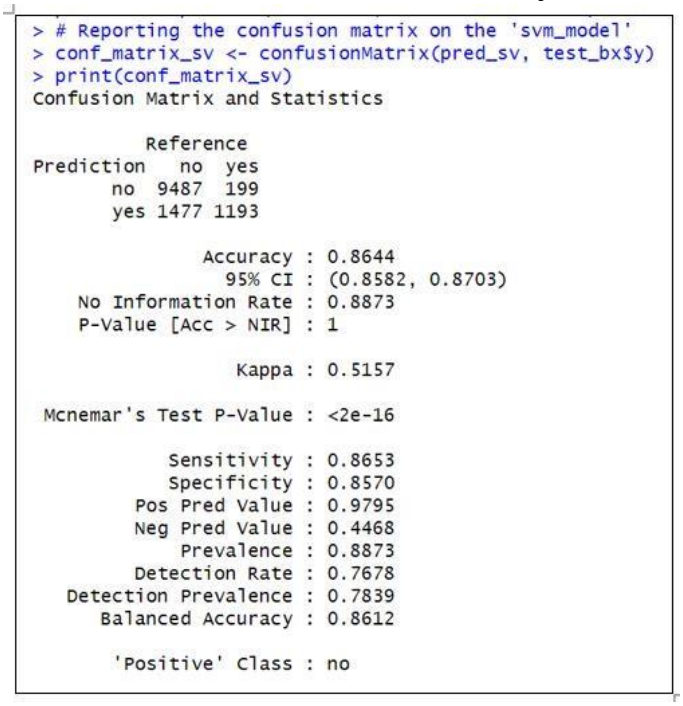
```
> # Train SVM model on combined data
> svm_model <- svm(y ~ ., data = data.rose, kernel = "radial")
> # Display summary of the SVM model
> summary(svm_model)

Call:
svm(formula = y ~ ., data = data.rose, kernel = "radial")


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1

Number of Support Vectors:  10501

 ( 5233 5268 )


Number of Classes:  2

Levels:
 no yes
```

Figure 27.0: Summary of SVM Model Trained on Balanced Data

This output provides a summary of the SVM (Support Vector Machine) model trained on the balanced data using a radial kernel. Here's the interpretation:

- **SVM-Type:** Indicates that the SVM model is used for C-classification, which is suitable for binary classification problems.

- **SVM-Kernel:** Specifies the type of kernel function used. In this case, a radial kernel is employed, which is often effective for nonlinear classification tasks.
- **Cost:** Represents the cost parameter, which controls the trade-off between maximizing the margin and minimizing the classification error. A cost value of 1 indicates a balanced penalty for misclassification errors.
- **Number of Support Vectors:** Indicates the total number of support vectors used by the model for decision boundary construction. In this case, there are 10,501 support vectors, with 5,233 belonging to the 'no' class and 5,268 belonging to the 'yes' class.
- **Number of Classes:** Specifies the number of classes in the target variable. Since it's a binary classification task, there are two classes: 'no' and 'yes'.



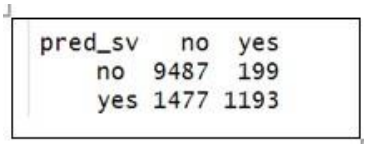
```
> # Reporting the confusion matrix on the 'svm_model'
> conf_matrix_sv <- confusionMatrix(pred_sv, test_bx$y)
> print(conf_matrix_sv)
Confusion Matrix and Statistics

          Reference
Prediction   no  yes
       no  9487  199
       yes 1477 1193

               Accuracy : 0.8644
                 95% CI : (0.8582, 0.8703)
    No Information Rate : 0.8873
    P-Value [Acc > NIR] : 1

                  Kappa : 0.5157

 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.8653
            Specificity : 0.8570
         Pos Pred Value : 0.9795
         Neg Pred Value : 0.4468
             Prevalence : 0.8873
         Detection Rate : 0.7678
   Detection Prevalence : 0.7839
      Balanced Accuracy : 0.8612

       'Positive' Class : no
```

Figure 28.0: Confusion Matrix - SVM Model

The confusion matrix in figure 17.0 shows the performance of the SVM model in predicting the classes 'no' and 'yes' based on the test data. Out of 11,356 instances, the model correctly classified 9,487 instances of 'no' and 1,193 instances of 'yes', but misclassified 199 instances of 'no' as 'yes' and 1,477 instances of 'yes' as 'no'. The overall accuracy of the model is approximately 86.44%, indicating the proportion of correctly classified instances. The Kappa statistic, which measures the agreement between observed and predicted classes, is 0.5157, suggesting moderate agreement beyond chance. Sensitivity, representing the true positive rate, is 86.53%, indicating the model's ability to correctly identify 'yes' instances. Specificity, representing the true negative rate, is 85.70%, showing the model's ability to correctly identify 'no' instances. The balanced accuracy, considering both sensitivity and specificity, is 86.12%, providing a comprehensive assessment of the model's performance.

- Accuracy: 0.864357397215928"

Table 3.0: Confusion Matrix SVM



```
pred_sv   no  yes
     no  9487  199
     yes 1477 1193
```

The SVM model, trained on the combined data using a radial kernel, demonstrates a balanced performance with an accuracy of approximately 86.44%. The confusion matrix reveals that out of

11,260 instances, the model correctly predicted 9,487 instances of the negative class ('no') and 1,193 instances of the positive class ('yes'). However, it misclassified 199 instances of the positive class as negative and 1,477 instances of the negative class as positive. This indicates a moderately reliable performance in classifying instances into their respective categories, with a higher accuracy compared to the Naive Bayes model.

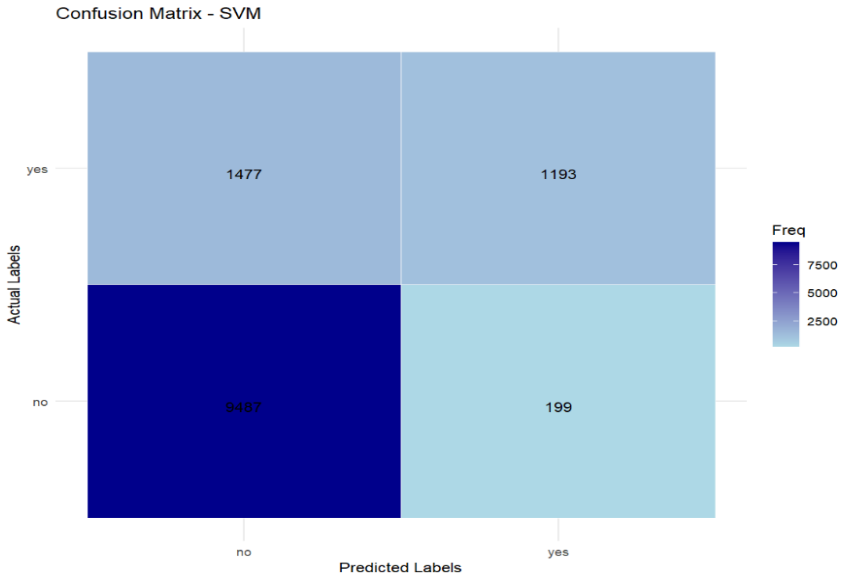


Fig 29: Confusion Matrix Visualization - SVM Model

Fig 29 shows the heatmap visualization of the confusion matrix for the SVM model predictions. Each cell in the heatmap represents the frequency of predictions where the x-axis denotes the predicted labels and the y-axis denotes the actual labels. The color gradient ranging from light blue to dark blue indicates the frequency, with darker shades representing higher frequencies. This visualization helps in understanding the performance of the SVM model by providing a clear representation of correct and incorrect predictions.

E. **KNN:** The K-nearest neighbors' algorithm was implemented to predict campaign outcomes.

Initiating the implementation of a KNN classifier in R, ensuring the availability of the ggplot2 package for visualization purposes. It then loads the necessary libraries (class, caret, and ggplot2) for the KNN setup. Subsequently, the dataset is prepared by splitting it into training and testing sets. The KNN algorithm is executed for different k values to assess its performance. The subsequent analysis evaluates misclassification error and accuracy metrics to gauge the effectiveness of the KNN algorithm across various k configurations.

Table 4.0: KNN Classifier Performance Metrics for Different Values of k

```
> # Print misclassification error and accura
cy for each run
> knn_res <- data.frame(K_Value = k_values,
Misclassification_Error = misclassification_
error, Accuracy = accuracy)
> print(knn_res)
   K_Value Misclassification_Error  Accuracy
1        1               0.11322434 0.8867757
2        2               0.11273875 0.8872612
3        3               0.09849466 0.9015053
4        4               0.10181288 0.8981871
5        5               0.09671415 0.9032859
6        6               0.09728067 0.9027193
7        7               0.09598576 0.9040142
8        8               0.09744254 0.9025575
9        9               0.09501457 0.9049854
10      10               0.09412431 0.9058757
11      11               0.09339592 0.9066041
12      12               0.09412431 0.9058757
13      13               0.09412431 0.9058757
14      14               0.09436711 0.9056329
15      15               0.09396245 0.9060376
16      16               0.09460991 0.9053901
17      17               0.09396245 0.9060376
18      18               0.09493364 0.9050664
19      19               0.09460991 0.9053901
20      20               0.09460991 0.9053901
```

The table 4.0 provides insights into the performance of a k-Nearest Neighbors (KNN) classifier across varying values of *kk*. It showcases the misclassification error and accuracy metrics for each run of the algorithm, ranging from k=1 to k=20. As k increases, there is a consistent pattern of decreasing misclassification error and increasing accuracy. This trend suggests that higher values of k contribute to improved performance of the KNN algorithm, resulting in more accurate classification of instances in the test set.

- **Performance Evaluation: KNN Confusion Matrix**

```
> # Report the confusion matrix
> conf_matrix_knn <- confusionMatrix(factor(
classifier_knn), factor(y_test))
> print(conf_matrix_knn)
Confusion Matrix and Statistics

          Reference
Prediction    no   yes
       no  10645   850
       yes   319   542

               Accuracy : 0.9054
                 95% CI : (0.9001, 0.9105)
    No Information Rate : 0.8873
    P-Value [Acc > NIR] : 4.419e-11

                  Kappa : 0.4322

 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.9709
            Specificity : 0.3894
         Pos Pred Value : 0.9261
         Neg Pred Value : 0.6295
             Prevalence : 0.8873
         Detection Rate : 0.8615
   Detection Prevalence : 0.9303
      Balanced Accuracy : 0.6801

       'Positive' Class : no
```

Figure 30.0: Confusion Matrix - KNN Classifier

The confusion matrix in figure 18.0 provides a detailed overview of the performance of the k-Nearest Neighbors (KNN) classifier. It reveals that out of a total of 11,644 instances predicted as "no," 10,645 were correctly classified, while 319 instances were falsely predicted as "yes." Similarly, among the 1,392 instances predicted as "yes," 542 were correctly classified, and 850 were incorrectly classified as "no." The overall accuracy of the classifier is calculated at approximately 90.54%, with a 95% confidence interval ranging from 90.01% to 91.05%. The Kappa statistic, which measures the agreement between the actual and predicted classes, is 0.4322, indicating a moderate level of agreement. Additionally, the sensitivity, specificity, positive predictive value, and negative predictive value provide insights into the classifier's performance across different metrics. Overall, the results suggest that while the classifier exhibits high sensitivity, it struggles with specificity, indicating a tendency to classify instances as "no" more frequently than "yes."

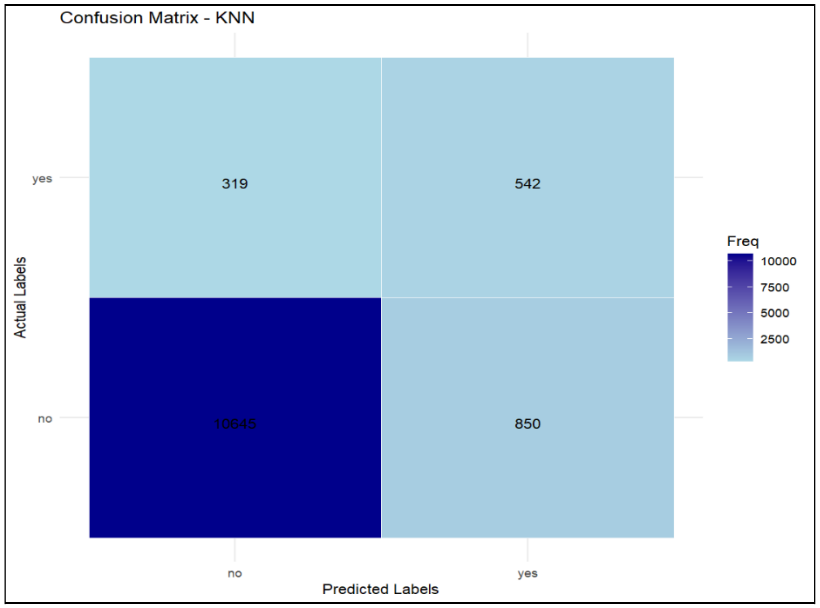


Fig 31.0: Confusion Matrix Visualization for KNN Classifier

Fig 31 shows the confusion matrix plot for the KNN classifier results. In this plot, the x-axis represents the predicted labels, while the y-axis represents the actual labels. Each cell in the matrix is colored based on the frequency of instances falling into that category, with lighter shades indicating lower frequencies and darker shades indicating higher frequencies. The title "Confusion Matrix - KNN" indicates that this plot specifically represents the confusion matrix generated for the KNN classifier. The theme is set to minimal, ensuring a clean and simple visualization.

- **ENSEMBLE METHODS**

F. **Random Forest:** The Random Forest model, an ensemble learning technique, combines multiple decision trees to improve predictive accuracy. Trained on combined data with specified parameters, it predicts outcomes on a test set and evaluates performance through a confusion matrix. Model accuracy is quantified, and a Variable Importance Plot illustrates the significance of different features.

```
> summary(rf_model)
                Length Class  Mode
call               8  -none- call
type               1  -none- character
predicted      28832  factor numeric
err.rate        1500  -none- numeric
confusion          6  -none- numeric
votes          57664  matrix numeric
oob.times      28832  -none- numeric
classes            2  -none- character
importance        72  -none- numeric
importanceSD      54  -none- numeric
localImportance    0  -none- NULL
proximity          0  -none- NULL
ntree              1  -none- numeric
mtry               1  -none- numeric
forest            14  -none- list
y              28832  factor numeric
test               0  -none- NULL
inbag              0  -none- NULL
terms              3  terms  call
```

Figure 32.0: Random Forest Model Summary

The Random Forest model was trained on combined data with specific parameters, and the 'summary' function provides essential insights into the model, including details about the call, predicted values, error rate, confusion matrix, feature importance, and model attributes, offering a concise overview of its characteristics and performance.

```
> # Predict on the test set
> pred_rf <- predict(rf_model, newdata = test_bx)
> # Report the confusion matrix on the 'svm_model'
> conf_matrix_rf <- confusionMatrix(pred_rf, test_bx$y)
> print(conf_matrix_rf)
Confusion Matrix and Statistics

          Reference
Prediction    no   yes
       no  10481   660
       yes   483   732

               Accuracy : 0.9075
                 95% CI : (0.9022, 0.9125)
    No Information Rate : 0.8873
    P-Value [Acc > NIR] : 1.846e-13

                  Kappa : 0.5101

 Mcnemar's Test P-Value : 1.931e-07

            Sensitivity : 0.9559
            Specificity : 0.5259
         Pos Pred Value : 0.9408
         Neg Pred Value : 0.6025
             Prevalence : 0.8873
         Detection Rate : 0.8483
   Detection Prevalence : 0.9017
      Balanced Accuracy : 0.7409

       'Positive' Class : no
```

Figure 33.0: Confusion Matrix - Random Forest

The confusion matrix in figure 20.0 presents the performance metrics of a Random Forest model on the test set. It displays the counts of correctly and incorrectly classified instances. The accuracy of the model is 90.75%, indicating the proportion of correctly classified instances. Additionally, the Kappa statistic, a measure of agreement between predicted and actual classifications, is 0.5101. The sensitivity and specificity metrics provide insights into the model's ability to correctly identify positive and negative instances, with values of 95.59% and 52.59% respectively. Overall, while the model demonstrates high sensitivity, its specificity is relatively lower, suggesting a tendency to correctly identify positive instances but with some limitations in correctly identifying negative instances.

- Accuracy: 0.907494334736161

The code prints the accuracy of the Random Forest model, which is approximately 90.75%. This metric indicates the proportion of correctly classified instances by the model out of the total instances in the dataset.

- **Visualization of Confusion Matrix for Random Forest Model**

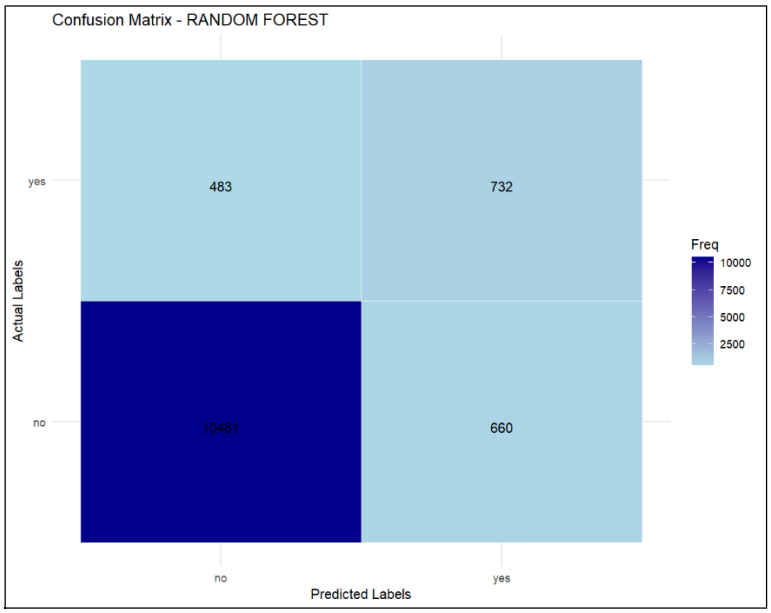


Fig 34.0: Confusion Matrix - Random Forest

The visualization in **Fig 34** depicts a confusion matrix for the Random Forest model. Each cell in the matrix represents the frequency of predictions made by the model compared to the actual labels in the test dataset. The color intensity indicates the frequency, with darker shades representing higher frequencies. The x-axis displays the predicted labels, while the y-axis shows the actual labels. This visual aids in understanding the model's performance in classifying instances into different categories.

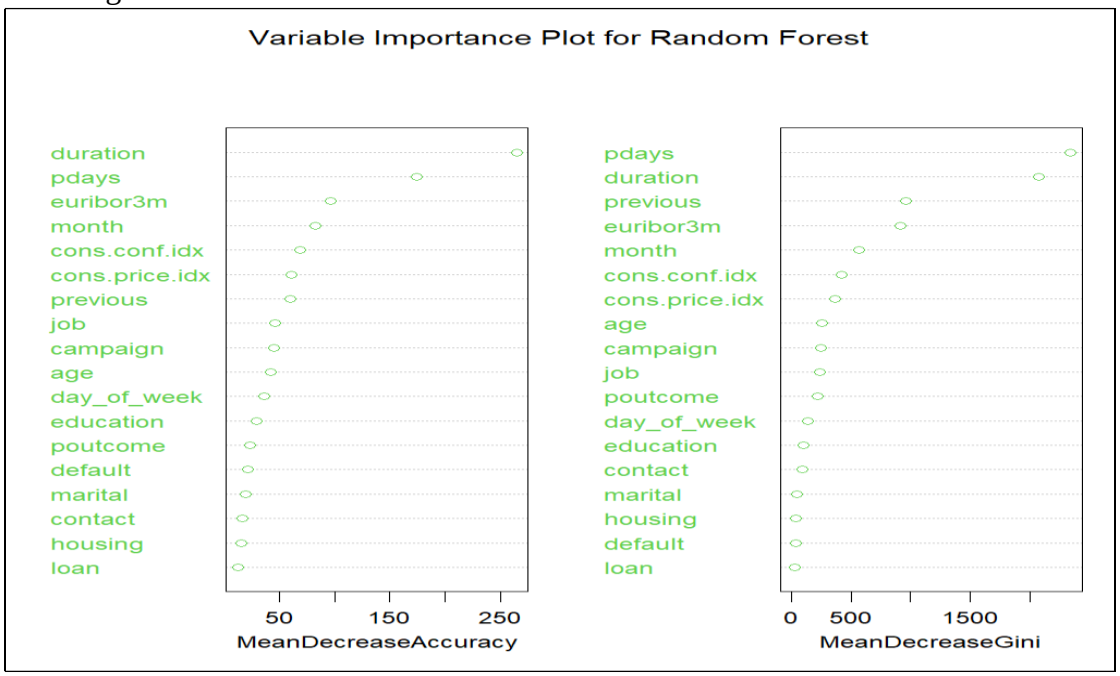


Fig 35.0: Variable Importance Plot: Random Forest

Fig 35 command generates a visual representation of the variable importance calculation using the Random Forest model. The "varImpPlot" function plots the importance of each predictor variable in the Random Forest model. The "rf_model" argument specifies the Random Forest model to be used, while the "col = 3" argument sets the color scheme for the plot. The "main" argument specifies the main title of the plot, which in this case is "Variable Importance Plot for Random Forest".

G. **Bagging:** short for Bootstrap Aggregating, is a popular ensemble learning technique used to improve the stability and accuracy of machine learning models, particularly decision trees.

It involves creating multiple models by training them on different subsets of the training data, sampled with replacement (bootstrap samples). These models are then combined by averaging or voting to make predictions, reducing overfitting and improving generalization performance. Bagging helps to increase robustness and accuracy, especially in complex datasets with high variance.

```
# Train Bagging model on combined data
bg_model <- ipred::bagging(y ~ ., data = data.rose, nbagg = 100)
summary(bg_model)
```

Figure 36.0: Bagging Model Training and Summary

The command in figure 21.0 trains a Bagging model on the combined data using the **bagging** function from the **ipred** package. It creates an ensemble of 100 models by aggregating decision trees trained on bootstrap samples of the data. The **summary** function provides an overview of the bagging model, including details such as the formula used, the number of bagged trees, and additional attributes such as out-of-bag (OOB) and combination (comb) indicators.

- Accuracy: 0.908141793460667

The output indicates that the accuracy of the Bagging model, as calculated from the confusion matrix, is approximately 0.9081, implying that the model correctly classifies about 90.81% of the instances in the test dataset.

```
> # Predict on the combined data
> pred_bg <- predict(bg_model, newdata = test_bx)
> # Reporting the confusion matrix on the 'svm_model'
> conf_matrix_bg <- caret::confusionMatrix(pred_bg, test_bx$y)
> print(conf_matrix_bg)
Confusion Matrix and Statistics

          Reference
Prediction    no   yes
       no  10548   719
       yes   416   673

               Accuracy : 0.9081
                 95% CI : (0.9029, 0.9132)
    No Information Rate : 0.8873
    P-Value [Acc > NIR] : 3.008e-14

                  Kappa : 0.4923

 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.9621
            Specificity : 0.4835
         Pos Pred Value : 0.9362
         Neg Pred Value : 0.6180
             Prevalence : 0.8873
         Detection Rate : 0.8537
   Detection Prevalence : 0.9119
      Balanced Accuracy : 0.7228

       'Positive' Class : no
```

Figure 37.0: Bagging Model Confusion Matrix

This confusion matrix in figure 21.0 presents the performance metrics of a bagging model on the combined data. The matrix illustrates the classification results, showing the counts of true negatives (TN), false positives (FP), false negatives (FN), and true positives (TP). The accuracy of the model, calculated as the proportion of correctly classified instances, is approximately 90.81%. The Kappa statistic, which measures the agreement between observed and predicted classifications, is 0.4923, suggesting moderate agreement. Sensitivity (true positive rate) is 96.21%, indicating the model's ability to correctly identify positive cases, while specificity (true negative rate) is 48.35%. Overall, the balanced accuracy, considering both sensitivity and specificity, is 72.28%, indicating reasonable performance in both classes.

- **Confusion Matrix Analysis: Bagging Model Performance**

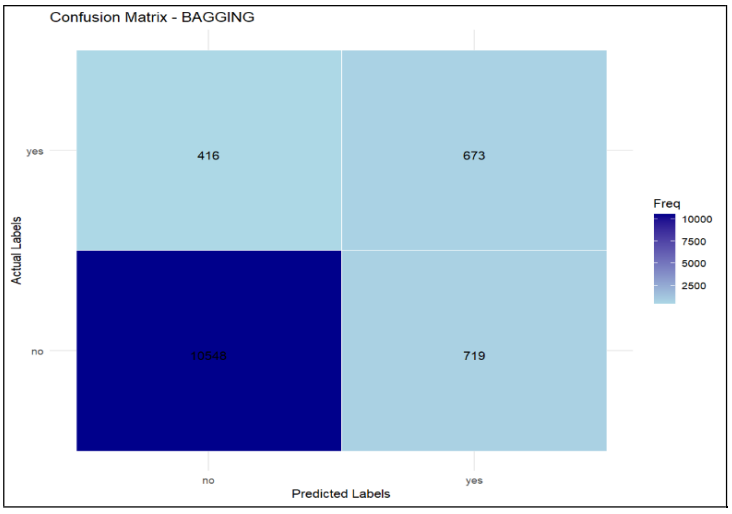


Fig 38.0: Bagging Model Confusion Matrix

The **Fig 38** illustrates the confusion matrix derived from evaluating the Bagging model's performance. Each cell represents the frequency of predictions compared to the actual labels in the test dataset, with color intensity indicating frequency. The diagonal denotes correct predictions, while off-diagonal cells signify misclassifications. The model achieved an accuracy of 90.81%, with a sensitivity of 96.21% and specificity of 48.35%. This visualization provides insights into the model's predictive accuracy and areas of misclassification.

H. **XGBoost:** is a highly efficient and scalable machine-learning algorithm used for supervised learning tasks, particularly valued for its performance in handling large datasets and complex models.

- **Feature Importance Analysis Using XGBoost**

```
[1]     train-logloss:0.296385
[2]     train-logloss:0.241095
[3]     train-logloss:0.221605
[4]     train-logloss:0.212753
[5]     train-logloss:0.206228
[6]     train-logloss:0.203039
[7]     train-logloss:0.199989
[8]     train-logloss:0.194573
[9]     train-logloss:0.192774
[10]    train-logloss:0.191606
```

Figure 39: Top 10 Feature Importances for Predicting Bank Deposits Using XGBoost

The script presented utilizes the XGBoost algorithm to evaluate the importance of different features in predicting whether clients will subscribe to a bank's long-term deposit product. The process begins by converting the feature matrix into a numeric format and transforming the labels into a binary format. Two XGBoost models are then trained with different numbers of boosting rounds. The feature importance is extracted from the second model, which underwent more rounds of training, providing a detailed view of the most influential features. The importance scores indicate which features significantly impact the model's predictions, thereby offering insights into the key factors driving client subscription decisions.
The plot displays the top ten features in figure 22.0 that have the most significant impact on predicting whether clients will subscribe to the bank's long-term deposit product, as determined by the XGBoost model.

- **XGBoost Feature Importance**

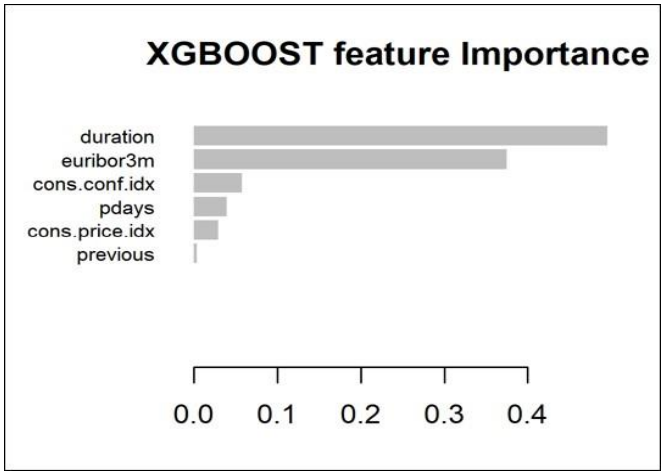


Fig 40.0: Top 10 Most Influential Features in XGBoost Model

The plot displays the top 10 features ranked by their importance in the XGBoost model. The features 'pdays', 'duration', 'cons.price.idx', and 'cons.conf.idx' are shown to have the highest influence on the model's predictions. This information is critical for understanding which variables most significantly impact the outcome and can guide feature selection and model improvement efforts.

```
# Print accuracy
print(paste("Accuracy:", accuracy_boost))
.] "Accuracy: 0.912026545807705"
```

Figure 41.0: High Accuracy of XGBoost Model: 91.2%

The XGBoost model achieved a high accuracy of 91.2%, indicating that it correctly classified 91.2% of the instances in the test set.

- **Confusion Matrix for XGBoost Model Predictions**

```
> print(conf_matrix_boost)
Confusion Matrix and Statistics

          Reference
Prediction     0     1
         0 10515   638
         1   449   754

               Accuracy : 0.912
                 95% CI : (0.9069, 0.917)
    No Information Rate : 0.8873
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.5323

 Mcnemar's Test P-Value : 1.183e-08

            Sensitivity : 0.9590
            Specificity : 0.5417
         Pos Pred Value : 0.9428
         Neg Pred Value : 0.6268
             Prevalence : 0.8873
         Detection Rate : 0.8510
   Detection Prevalence : 0.9026
      Balanced Accuracy : 0.7504

       'Positive' Class : 0
```

Figure 42.0: Confusion Matrix - XGBoost Model Predictions

The confusion matrix results in figure 24.0 for the XGBoost model show a high overall accuracy of 91.2%, with a 95% confidence interval ranging from 90.69% to 91.7%, significantly outperforming the No Information Rate of 88.73%. The model exhibits excellent sensitivity (95.9%), indicating a strong ability to correctly identify clients who did not subscribe to the product. However, specificity is moderate at 54.17%, suggesting room for improvement in accurately identifying clients who did subscribe. The model's precision is high at 94.28%, ensuring reliable predictions for non-subscribers. Despite these strengths, the balanced accuracy is 75.04%, highlighting the need for better identification of subscribers. The model’s performance is statistically significant with a Kappa of 0.5323 and a Mcnemar's Test P-Value of 1.183e-08, confirming the robustness and reliability of its predictions. Overall, the model is effective but can be improved for better specificity.

- **Confusion Matrix Visualization for XGBoost Model on Test Data**

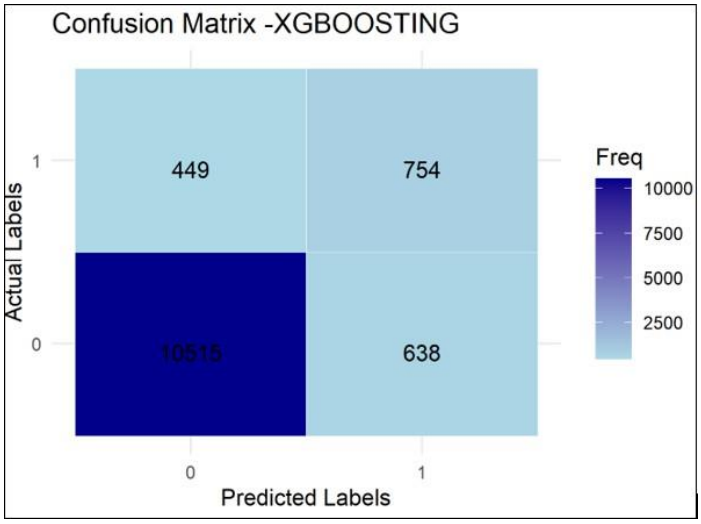


Fig 43.0: Confusion Matrix - XGBoost Model Performance

The plot in **Fig 43** visualizes the confusion matrix of the XGBoost model's performance on the test data, illustrating the distribution of true positive, true negative, false positive, and false negative predictions. The majority of predictions fall along the diagonal, indicating correct classifications, with high counts in the true positive and true negative cells. The off-diagonal cells, representing misclassifications, show fewer instances, highlighting the model's overall high accuracy. The color gradient from light blue to dark blue provides a clear visual distinction of the frequency of occurrences, aiding in the quick assessment of model performance.

I. **Random Forest Model Performance on Numeric Dataset:** The Random Forest model, trained and tested on numeric features, exhibits high accuracy, validated through a detailed confusion matrix that underscores its strong predictive capacity and identifies areas for potential enhancement.

```
print(paste("Random Forest Accuracy:", accuracy_rf_num)
[1] "Random Forest Accuracy: 0.91437358368404"
```

Figure 44.0: Random Forest Model Accuracy

The accuracy of the Random Forest model on numeric features is 91.44%, indicating its ability to correctly predict the outcome for the majority of instances in the test dataset.

- **Classification based on the numeric variables**

```
> # Print confusion matrix
> print(conf_matrix_rf2)
Confusion Matrix and Statistics

          Reference
Prediction    no   yes
       no  10605   699
       yes   359   693

               Accuracy : 0.9144
                 95% CI : (0.9093, 0.9192)
    No Information Rate : 0.8873
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.5206

 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.9673
            Specificity : 0.4978
         Pos Pred Value : 0.9382
         Neg Pred Value : 0.6587
             Prevalence : 0.8873
         Detection Rate : 0.8583
   Detection Prevalence : 0.9149
      Balanced Accuracy : 0.7326

       'Positive' Class : no
```

Figure 45.0: Confusion Matrix - Random Forest Model Performance

The confusion matrix and statistics for the Random Forest model in figure 45.0 reveal a robust performance, with an overall accuracy of 91.44%, indicating that the model correctly predicts the outcome in 91.44% of cases. The 95% confidence interval (90.93% - 91.92%) and a p-value of < 2.2e-16 confirm the model's reliability and superiority over random guessing. The sensitivity is high at 96.73%, showing excellent detection of the 'no' class, while the specificity is lower at 49.78%, reflecting some challenges in identifying the 'yes' class. The Kappa statistic of 0.5206 indicates moderate agreement beyond chance. Positive Predictive Value (93.82%) and Negative Predictive Value (65.87%) further illustrate the model's effectiveness in predicting 'no' and 'yes' classes, respectively. Despite the lower specificity, the balanced accuracy of 73.26% suggests a reasonably good performance across both classes, highlighting the model's potential while also indicating areas for improvement in handling the imbalanced dataset.

- **Analysis of Random Forest Model Performance on Numeric Features**

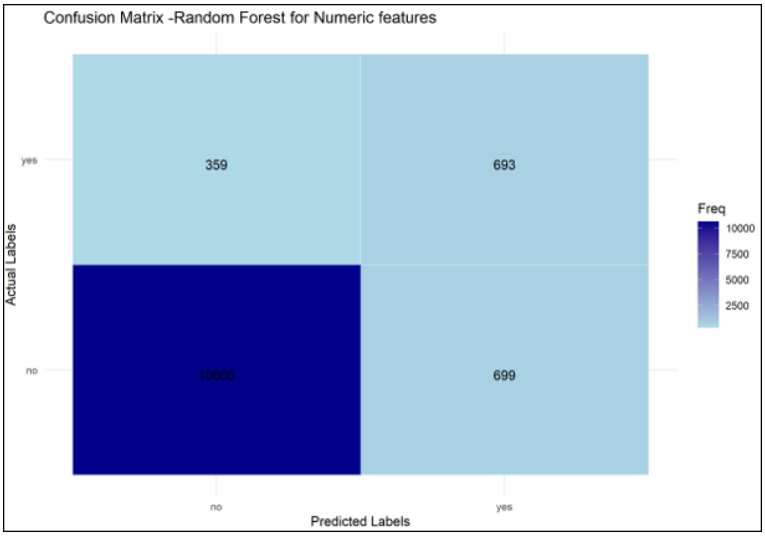


Fig 46.0: Confusion Matrix - Random Forest for Numeric Features

The confusion matrix for the Random Forest model **Fig 46** trained on numeric features illustrates its classification performance. The matrix reveals the number of true positive, true negative, false positive, and false negative predictions. The model achieves an accuracy of 91.44%, indicating its ability to correctly classify the majority of instances. Additionally, sensitivity, specificity, positive predictive value, and negative predictive value metrics provide insights into the model's ability to

identify true positives and negatives and avoid false positives and negatives. With above result, we can conclude that Random Forest seem to perform better than on the numerical variables than that of KNN with an accuracy of 91.44% and a recall of 49.7%

J. **Clustering (K-means, DBSCAN, PAM):** Clustering analysis was performed using K-means, DBSCAN, and PAM algorithms to identify underlying patterns in the data.

a. K-means:

The **bank_num** data frame comprises 41,188 rows and 8 columns, representing various attributes related to individuals in a banking dataset. These attributes include demographic information such as age, as well as features related to the duration of the last contact, the number of contacts made during the campaign, and metrics like the consumer price index and the Euribor 3-month rate.

```
> # Create a copy of the dataset excluding the target variable 'y'
> bank_num <- bank_sub
> bank_num$y <- NULL
> # Check the structure of the dataset
> str(bank_num)
'data.frame':   41188 obs. of  8 variables:
 $ age           : int  56 57 37 40 56 45 59 41 24 25 ...
 $ duration      : int  261 149 226 151 307 198 139 217 380 50 ...
 $ campaign      : int  1 1 1 1 1 1 1 1 1 1 ...
 $ pdays         : int  999 999 999 999 999 999 999 999 999 999 ...
 $ previous      : int  0 0 0 0 0 0 0 0 0 0 ...
 $ cons.price.idx: num  94 94 94 94 94 ...
 $ euribor3m     : num  4.86 4.86 4.86 4.86 4.86 ...
 $ cons.conf.idx : num  -36.4 -36.4 -36.4 -36.4 -36.4 -36.4 -36.4 -36.4 -36.4 -36.4 ...
```

Figure 47.0: Summary of bank_num Data Structure

Additionally, variables like **pdays** denote the number of days since the client was last contacted from a previous campaign, while **previous** represents the number of contacts made before the current campaign. This structured dataset provides a foundation for analyzing banking trends and customer interactions, facilitating insights into customer behavior and campaign effectiveness.

- **Finding the Optimal Number of Clusters Using the Elbow Method**

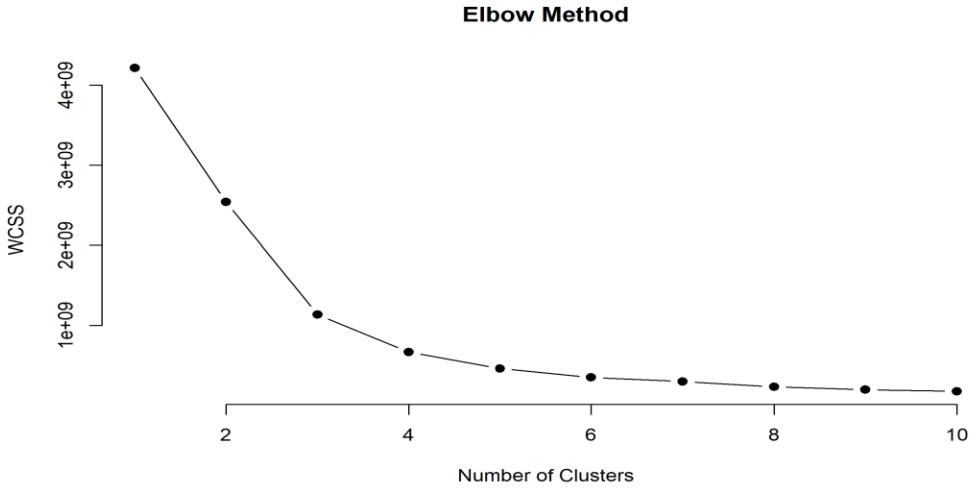


Fig 48.0: Optimal Number of Clusters Determined by the Elbow Method

The Elbow Method graph in **Fig 48** typically shows the within-cluster sum of squares (WCSS) plotted against the number of clusters (k). The optimal point, or "elbow," is where the rate of decrease in WCSS significantly slows down. This point indicates the most appropriate number of clusters for the dataset, balancing the trade-off between model simplicity and the explanatory

power of the clusters. Selecting the number of clusters at the elbow point ensures that the clustering model is neither too simple (underfitting) nor too complex (overfitting), providing a meaningful and interpretable segmentation of the data.

- **Clustering Vector Analysis**

The clustering vector provided lists the cluster assignments for each data point in the dataset. Each number corresponds to the cluster label assigned to that particular observation after performing clustering analysis. In this example, the data points are predominantly assigned to two clusters, labeled '1' and '2'.

```
Clustering vector:
  [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 2 2 2 2
 [44] 2 2 2 1 2 2 2 2 2 2 2 2 2 2 1 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 1 2 2
 [87] 1 2 1 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2
[130] 1 2 1 2 2 2 2 2 2 2 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 1 1 2 1 2 2 1 2 2
[173] 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 1 1 2 2 2 2 2 2 2 2 2 2 2 1 2
[216] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 1 2 2 2 2 2 2 2 2
[259] 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 1 2 2 1 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 1 2 2 2 2
[302] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2
[345] 2 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[388] 1 1 1 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 2 2 1 2 2 1 2 1 2 2 2 2 2 2
```

Figure 49.0: Clustering Vector Results

The results in figure 28.0 indicate that the data points have been grouped into two distinct clusters. A majority of the points are assigned to cluster '2', while a smaller number of points are grouped into cluster '1'. This suggests that the dataset has two primary groups with different characteristics. The clustering vector helps in understanding the distribution of these groups within the dataset.

- **Within-Cluster Sum of Squares (WCSS) Analysis**

```
Within cluster sum of squares by cluster:
[1] 601146487 456645476  77643589
 (between_SS / total_SS =  73.1 %)
```

Figure 50.0: Within-Cluster Sum of Squares and Cluster Variability Analysis

The within-cluster sum of squares (WCSS) in figure 29.0 is a measure of the variability of data points within each cluster. Lower WCSS values indicate that the data points within a cluster are closer to each other, implying more compact and well-defined clusters.
In this case, the WCSS values for the three clusters are:

- Cluster 1: 601,146,487
- Cluster 2: 456,645,476
- Cluster 3: 77,643,589

The variability within Cluster 3 is much lower compared to Clusters 1 and 2, suggesting that the data points in Cluster 3 are more tightly grouped. The proportion of between-cluster sum of squares (SS) to the total sum of squares is 73.1%, which indicates that 73.1% of the total variance is explained by the differences between the clusters. This high percentage signifies that the clustering algorithm has effectively captured the underlying structure of the data, separating it into distinct clusters.

- **Silhouette Analysis for Optimal Clustering**

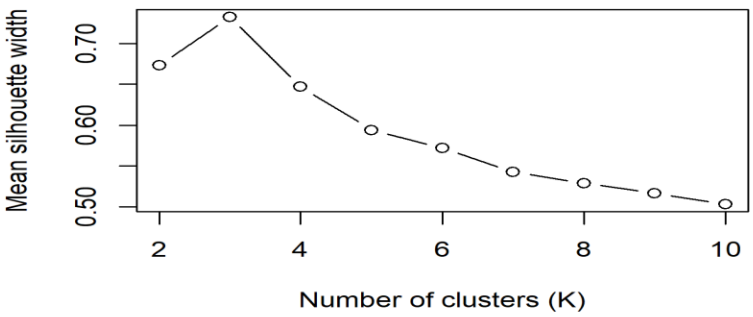


Fig 51.0: Silhouette Analysis for Determining Optimal Number of Clusters

The silhouette method evaluates the quality of clusters by measuring how similar each point is to its own cluster compared to other clusters. The silhouette width ranges from -1 to 1, where higher values indicate that the points are well-clustered and distinct from other clusters.

In this analysis, the silhouette widths are calculated for K values ranging from 2 to 10. The plot in Fig 51 displays the mean silhouette width for each K, helping identify the optimal number of clusters. The ideal number of clusters is typically the one that maximizes the mean silhouette width, indicating the best-defined clustering structure.

- **Optimal Number of Clusters Based on Silhouette Width**

```
> optimal_k <- k_values[which.max(silhouette_widths)]
> cat("Optimal number of clusters based on silhouette width:", optimal_k, "\n'
Optimal number of clusters based on silhouette width: 3
```

Figure 52.0: Optimal Clusters Determined: K = 3

The silhouette analysis has determined that the optimal number of clusters for the dataset is 3. This conclusion is based on the silhouette width, which measures the cohesion and separation of the clusters. A higher silhouette width indicates better-defined clusters, and in this case, the maximum silhouette width was observed when the number of clusters (K) is 3. This suggests that dividing the dataset into three distinct groups yields the best clustering structure.

b. NbClust

- **Determining the Best Number of Clusters Using NbClust**

To identify the optimal number of clusters in the dataset, we employed the NbClust function. This function evaluates different clustering solutions by varying the number of clusters (from 2 to 10) and selects the best clustering structure based on multiple criteria. The dataset was sampled for computational efficiency, and the NbClust analysis indicated the most suitable number of clusters. The results, summarized in a frequency table, reveal the number of times each cluster count was chosen as the optimal solution.

- **Optimal Number of Clusters Based on NbClust Results**

```
*******************************************************************
* Among all indices:
* 12 proposed 3 as the best number of clusters
* 2 proposed 4 as the best number of clusters
* 2 proposed 5 as the best number of clusters
* 1 proposed 7 as the best number of clusters
* 4 proposed 9 as the best number of clusters
* 2 proposed 10 as the best number of clusters

                   ***** Conclusion *****

* According to the majority rule, the best number of clusters is  3
```

Figure 53.0: NbClust Analysis: Optimal Number of Clusters

The NbClust function was used to determine the most suitable number of clusters for the dataset. Based on the analysis, 12 indices suggested that 3 clusters are optimal, while fewer indices recommended 4, 5, 7, 9, and 10 clusters. The majority rule indicates that the best number of clusters is 3, suggesting that the dataset can be most effectively partitioned into three distinct groups.

- **Optimal Number of Clusters Based on NbClust Results**

Table 5.0: NbClust Analysis: Distribution of Optimal Cluster Numbers

```
> best_clust <- table(nbclust_result$Best.n[1,])
> best_clust

 0  1  3  4  5  7  9 10
 2  1 12  2  2  1  4  2
>
```

The table 5.0 shows the distribution of the best number of clusters recommended by NbClust. Among the indices, 12 instances proposed 3 clusters, while 2 instances each suggested 4, 5, 7, 9, and 10 clusters. Additionally, 1 instance each recommended 0, 1, and 2 clusters.
Considering the majority rule, the most frequently proposed number of clusters is 3, indicating its suitability for partitioning the dataset into distinct groups.

- **NbClust Analysis: Evaluating Optimal Number of Clusters**

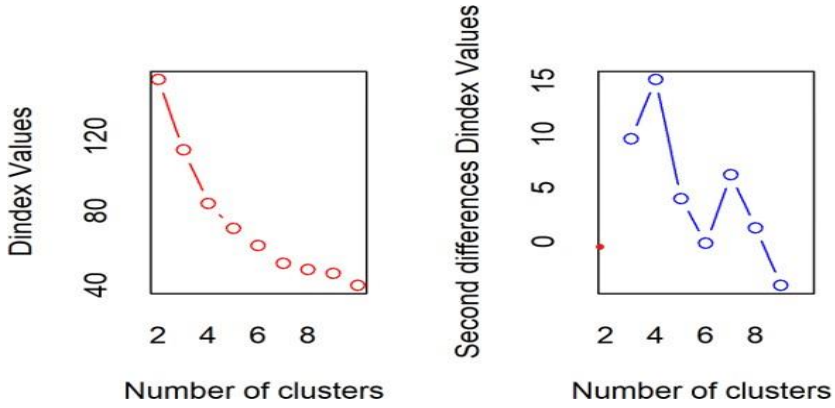


Fig 55.0: Exploring Optimal Number of Clusters: NbClust Analysis

The first plot **Fig 55** displays the silhouette width across different numbers of clusters, suggesting that the silhouette width is highest when the number of clusters is 3, indicating better cluster quality. The second plot demonstrates the distribution of the best number of clusters recommended by NbClust. It shows that among all indices, 12 instances proposed 3 clusters, making it the most frequently recommended number.

a. PAMK

Partitioning Around Medoids (PAM) clustering, a method used for grouping data points into distinct clusters based on their similarity, was applied to a subset of the dataset using the pamk function, exploring cluster numbers from 2 to 10. The resulting clustering solution was stored in pamk_bf, and the PAM clustering results were visualized using silhouette plots to assess the quality of the clustering solution. Silhouette plots provided insight into the cohesion and separation of clusters, aiding in the evaluation of cluster validity and the determination of the optimal number of clusters.

- **PAM Clustering Results:**

```
> pamk_bf
$pamobject
Medoids:
        ID age duration campaign pdays previous cons.price.idx euribor3m cons.conf.idx
25931  159  39      457        2   999        0         93.200     4.120         -42.0
7988  9147  39      131        2   999        0         94.465     4.865         -41.8
40417 2923  41      258        2     6        3         94.027     0.905         -38.3
Clustering vector:
 2986 29925 29710 37529  2757 38938  9642 31313 14183 15180 27168 24173  9097 30538 28981
    1     2     2     2     2     1     2     1     1     2     2     2     2     2     2
 7989 13536 24541  6216 17983 29394 28825    41 14426 40159  7284 28502 11473 12301  6134
    1     2     2     2     2     2     2     1     2     3     1     1     2     1     1
33523 21812 39895  9640 19742  9326 26510 20960 14403  5967 28799 26836 12049 15150  5027
    2     2     3     2     2     2     2     2     2     2     1     2     1     2     1
32606 16152 25559 14215 14287 23194 24558 34976 14491 17413 12048 40503 34724 38126 20477
    1     1     2     2     1     1     2     2     2     2     2     2     1     2     1
17369 31542 37001 21069 40632 30575 37544 33753 18891 11284 16579 26801 25902 36717 28078
    2     2     2     1     2     2     2     1     2     2     1     2     2     2     2
 3833 26028 14536 17533 26503 37783 22763 32953   413 10762 30571  8986 14745 25946 14804
    1     2     2     1     2     2     2     2     2     1     1     2     2     2     1
 6601  6790   618 31517   539 32263 30453  3625 15582 19392  2211  2286 18762 32281  7826
    2     2     2     2     2     1     1     2     1     2     2     2     2     2     1
14751 24263 25449 40849 10687 27455 12637 20193  1165 31665 22467 29826 13689 18496 13795
    2     2     1     2     2     2     2     2     1     2     2     2     2     2     2
19419 15703  6623  8983 37769 29479  5407 32634 12585 21085  1835 21099 41043 25442 36940
```

Figure 56.0: PAM Clustering Results and Cluster Assignments

The PAM clustering algorithm was applied to the dataset as shown in figure 32.0 , resulting in the identification of medoids for each cluster. Medoids are representative points within each cluster that minimize the average dissimilarity to all other points in the same cluster. In this case, three medoids were identified, each representing a distinct cluster. Additionally, the clustering vector indicates the assignment of each observation to one of the identified clusters, with cluster labels ranging from 1 to 3. This information provides insights into the composition and structure of the clusters formed by the PAM algorithm based on the features of the dataset.

- **PAM Clustering Results and Cluster Assignments**

```
     2     3     1     2     1     2     2     2     1     1
 [ reached getOption("max.print") -- omitted 9000 entries ]
Objective function:
   build     swap
107.1651 101.6752

Available components:
 [1] "medoids"    "id.med"     "clustering" "objective"  "isolation"  "clusinfo"
 [7] "silinfo"    "diss"       "call"       "data"

$nc
[1] 3
```

Figure 57.0: PAM Clustering Results and Cluster Assignments (Partial Output)

The PAM clustering algorithm identified three medoids for this dataset in figure 33.0, each representing a distinct cluster. The clustering vector indicates the assignment of each observation to one of the identified clusters, with cluster labels ranging from 1 to 3. The omission of 9000 entries indicates that there are a large number of observations in the dataset, and the print function is configured to limit the output to the maximum allowed. The objective function value, which includes components such as build and swap, provides insights into the optimization process undertaken by the PAM algorithm to identify the best clusters based on the dissimilarity matrix.

- **PAM Clustering Medoids Results**

```
> pamk_bf$pamobject$medoids
      age duration campaign pdays previous cons.price.idx euribor3m cons.conf.idx
25931  39      457        2   999        0         93.200     4.120         -42.0
7988   39      131        2   999        0         94.465     4.865         -41.8
40417  41      258        2     6        3         94.027     0.905         -38.3
```

Figure 58.0: PAM Clustering: Medoids for Identified Clusters"

The medoids in figure 34.0 represent the central points of each cluster identified by the Partitioning Around Medoids (PAM) clustering algorithm. These medoids are characterized by specific feature values such as age, duration, campaign, pdays, previous, cons.price.idx, euribor3m, and

cons.conf.idx. These values provide insights into the typical characteristics of each cluster, aiding in understanding the underlying patterns within the data.

- **PAM Clustering Plots**

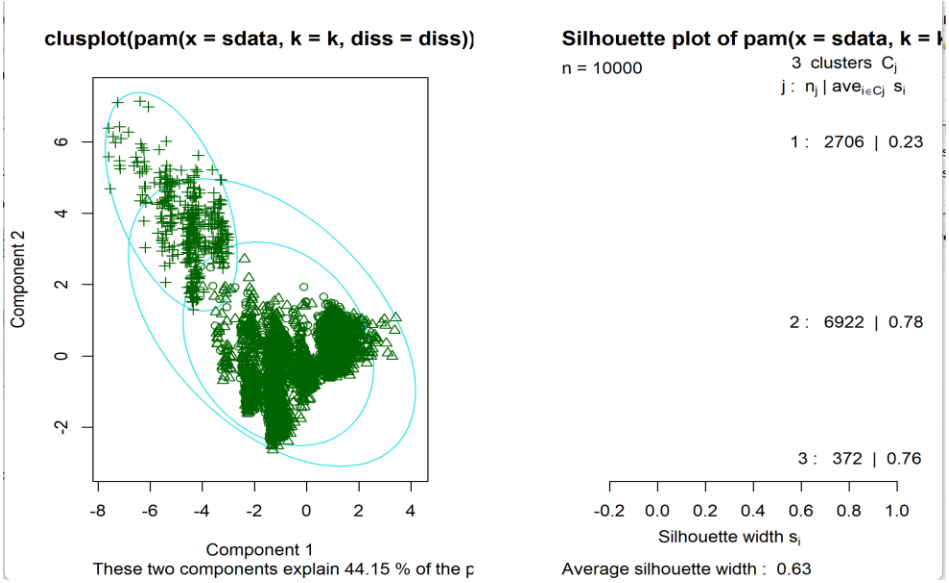


Fig 59.0: Visualization of PAM Clustering Results

The clusplot function generates a visual representation of the clusters identified by the Partitioning Around Medoids (PAM) algorithm. Each point in the plot represents an observation, and the color denotes its assigned cluster. This plot provides a visual understanding of the distribution and separation of clusters in the data space.

The silhouette plot in **Fig 59**, generated by the pam function, illustrates the silhouette widths for each observation in the dataset. Silhouette width measures the cohesion and separation of clusters, with values closer to 1 indicating better cluster membership and separation. The average silhouette width of 0.63 suggests that the clustering algorithm effectively partitions the data into distinct clusters.

d. DBSCAN

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm is applied to the dataset with varying parameter settings to find the configuration that maximizes clustering quality, assessed using silhouette width. After iterative evaluation, the optimal parameters are identified, and DBSCAN clustering is performed accordingly. The resulting clusters are visually represented, with observations color-coded based on their assigned cluster. Additionally, a kNN distance plot aids in determining the optimal epsilon value by identifying the knee point, crucial for effective clustering. Finally, the DBSCAN algorithm is executed with the determined parameters, and the resulting clusters and noise points are visualized, providing comprehensive insights into the dataset's clustering structure.

- **DBSCAN Clustering Results**

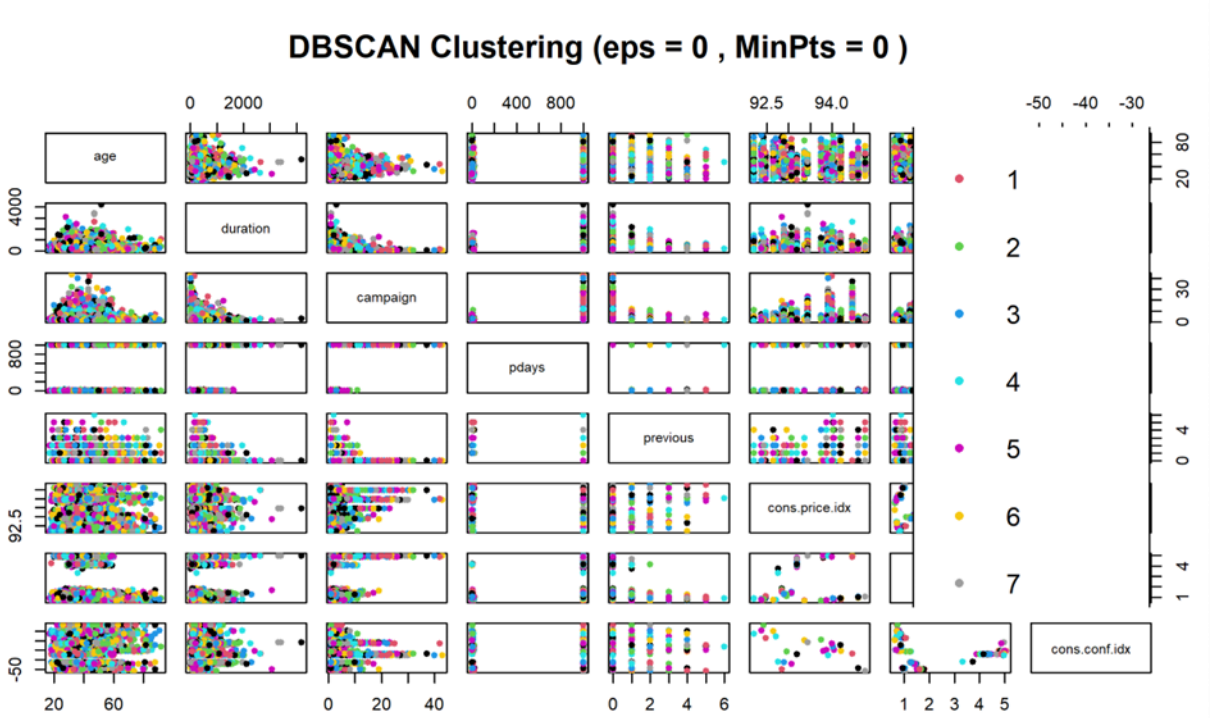


Fig 60: DBSCAN Clustering: Exploring Feature Relationships and Cluster Distribution

The plots in **Fig 60** illustrate the clustering outcomes obtained from the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm with epsilon (eps) and minimum points (Mints) set to 0. Each scatterplot represents different pairs of features from the dataset, including age, duration, campaign, pdays, previous, cons.price.idx, and cons.conf.idx. Observations are marked with points, where the color indicates the assigned cluster. The visualizations reveal the clustering patterns and the distribution of points across various feature combinations. These insights are crucial for understanding the inherent structure of the dataset and the effectiveness of the DBSCAN algorithm in identifying meaningful clusters.

- **Plotting KNNdist Plot**

DBSCAN Clustering Results with Optimal Parameters

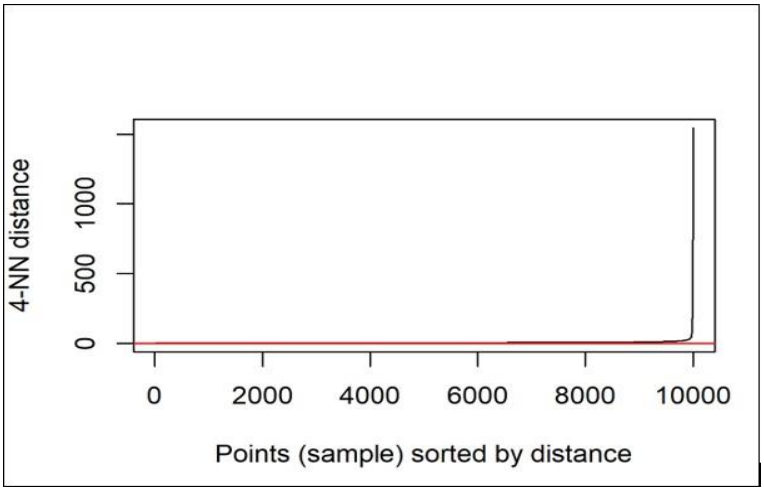


Fig 61.0: DBSCAN Clustering (eps = 0, MinPts = 0)

The plot in **Fig 61** visualizes the DBSCAN clustering results obtained with the optimal parameters (eps = 0, MinPts = 0). The clusters are represented by different colors, with each point indicating

a data instance. This clustering technique identifies dense regions as clusters and outliers as noise. With eps and MinPts set to 0, DBSCAN adapts to the data's local density, resulting in clusters of varying shapes and sizes

- **Determining Optimal Epsilon for DBSCAN Clustering**

```
+ }
> eps(bank_num_sample)
     39720
0.01830199
>
```

Figure 62.0: Optimal Epsilon Determination for DBSCAN Clustering

The function in figure 35.0 eps(bank_num_sample) calculates the optimal value of epsilon (0.01830199) for DBSCAN clustering using the k-distance plot method. Epsilon determines the maximum distance between points to be considered in the same neighborhood, influencing the cluster formation in DBSCAN. Visualize: The function generates a k-distance plot, indicating the distances of points sorted in ascending order. The optimal epsilon is marked with a red dashed line at a distance of 0.01830199.

- **Eps = 0.01830199**

```
> bank_dbs
DBSCAN clustering for 10000 objects.
Parameters: eps = 0.01830199, minPts = 9
Using euclidean distances and borderpoints = TRUE
The clustering contains 0 cluster(s) and 10000 noise points.

   0
```

Figure 63.0: DBSCAN Clustering Results with No Clusters Detected

The bank_dbs object represents the result of DBSCAN clustering performed on a dataset with 10,000 objects. The clustering was conducted with an epsilon value of 0.01830199 and a minimum number of points (minPts) set to 9. The clustering process utilized Euclidean distances, and border points were included in the analysis. However, the clustering yielded no clusters, indicating that all 10,000 objects were classified, as noise points. There is no visualization available since the clustering resulted in all points being categorized as noise.

- **Convex Cluster Hulls Visualization**
- **Dbscan produced zero noises with eps of 0.01830199**

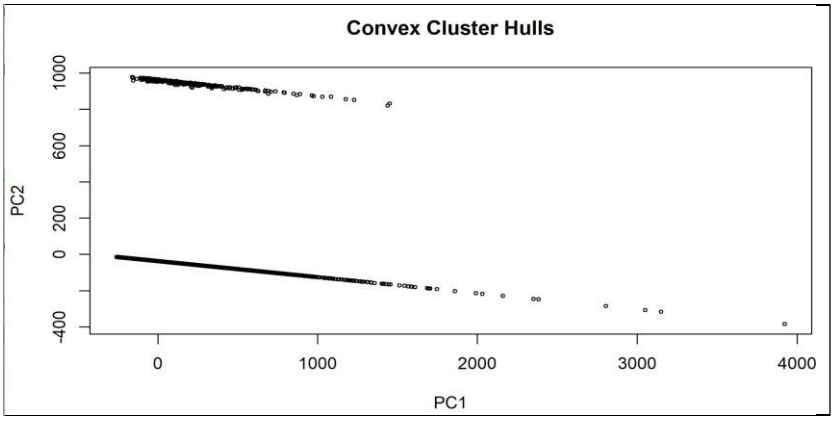


Fig 64.0: Convex Cluster Hulls for DBSCAN Clustering with Zero Noise Points"

The plot in **Fig 64** displays the convex cluster hulls generated by the DBSCAN clustering algorithm. With an epsilon (eps) value of 0.01830199, DBSCAN produced zero noise points. The convex hulls represent the boundaries of clusters formed by the algorithm. The plot shows the distribution of data points in a two-dimensional space represented by Principal Component 1 (PC1) and Principal Component 2 (PC2). Each cluster is enclosed by its convex hull, illustrating the shape and extent of the clusters identified by the DBSCAN algorithm.

Python: Predictive Analysis for Bank Marketing

1. Introduction:

The goal of this project is to perform predictive analysis to determine whether a client will subscribe to a long-term deposit, (binary classification) based on various features provided in the dataset. The dataset contains information about clients such as age, job, marital status, education, housing loan status, previous marketing campaign outcomes, etc. We aim to build machine learning models to predict whether a client will subscribe to a term deposit, using various algorithms and evaluating their performance.

2. Data Exploration and Preprocessing:

- We began by loading the dataset using pandas and explored its structure and basic statistics.

This Python code loads a dataset named "bank_marketing.csv" from the specified file path using pandas, a data manipulation library in Python. The data is read with the assumption that the delimiter between values is a semicolon (**;**). The **head(5)** function then displays the first 5 rows of the dataset for quick examination.

Interpreting this code:

1. **Loading Dataset**: The **pd.read_csv()** function is used to read the CSV file into a pandas DataFrame named **df**. The file path is specified as:

   "C:/Users/Rakesh/Desktop/Varada_ALY6040/bank_marketing.csv". The **delimiter** parameter is set to ';' indicating that semicolons are used to separate the values in the CSV file.

2. **Displaying First 5 Rows**: The **head(5)** function is called on the DataFrame **df** to display the first 5 rows of the dataset. This provides a quick overview of the structure and contents of the dataset, allowing users to inspect the data and understand its format before further analysis or processing.

```
In [2]: df = pd.read_csv("C:/Users/Rakesh/Desktop/Varada_ALY6040/bank_marketing.csv", delimiter=';')
        df.head(5)
```

Out[2]:

g	loan	contact	month	day_of_week	...	campaign	pdays	previous	poutcome	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	y
o	no	telephone	may	mon	...	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
o	no	telephone	may	mon	...	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
s	no	telephone	may	mon	...	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
o	no	telephone	may	mon	...	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
o	yes	telephone	may	mon	...	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no

Figure 65: Displaying First 5 Rows of bank_marketing.csv

Table 6.0: Summary of Dataframe Structure:

```
In [4]: df.shape
Out[4]: (41188, 21)

In [6]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 21 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             41188 non-null  int64
 1   job             41188 non-null  object
 2   marital         41188 non-null  object
 3   education       41188 non-null  object
 4   default         41188 non-null  object
 5   housing         41188 non-null  object
 6   loan            41188 non-null  object
 7   contact         41188 non-null  object
 8   month           41188 non-null  object
 9   day_of_week     41188 non-null  object
 10  duration        41188 non-null  int64
 11  campaign        41188 non-null  int64
 12  pdays           41188 non-null  int64
 13  previous        41188 non-null  int64
 14  poutcome        41188 non-null  object
 15  emp.var.rate    41188 non-null  float64
 16  cons.price.idx  41188 non-null  float64
 17  cons.conf.idx   41188 non-null  float64
 18  euribor3m       41188 non-null  float64
 19  nr.employed     41188 non-null  float64
 20  y               41188 non-null  object
dtypes: float64(5), int64(5), object(11)
memory usage: 6.6+ MB
```

The output in table 6.0 provides insightful information about the DataFrame **df**:

1. **Shape**: The DataFrame has 41,188 rows and 21 columns.
2. **Data Type Summary**: The data types of each column are shown. These include:
    - 5 columns with data type **float64**.
    - 5 columns with data type **int64**.
    - 11 columns with data type **object**.
3. **Column Information**:
    - Each column is listed along with its non-null count and data type.
    - The **Non-Null Count** indicates the number of non-null values in each column, confirming that there are no missing values in the dataset.
    - The **Dtype** column specifies the data type of each column.
4. **Memory Usage**: The total memory usage of the DataFrame is approximately 6.6 MB, which can be useful for understanding memory requirements when working with larger datasets.

- Univariate analysis involved visualizing the distribution of numerical and categorical variables using histograms and count plots, respectively. This helped understand the data's distribution and identify potential outliers or data imbalances.

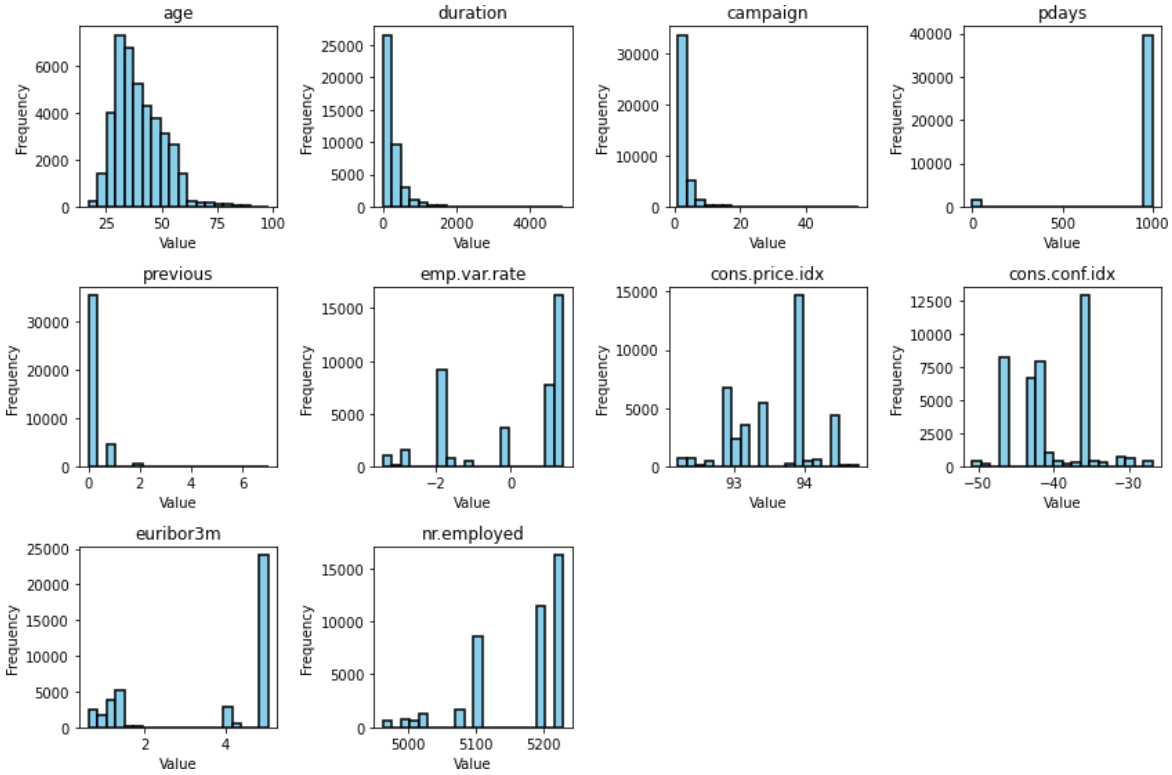


Fig 66.0: Univariate Analysis of Numerical Features

The above code segment in **Fig 66** conducts a univariate analysis on numerical columns in a DataFrame **df**. Here's the visualization:

1. **Age**: Shows the age distribution of individuals in the dataset.
2. **Duration**: Illustrates the distribution of contact durations.
3. **Campaign**: Displays the frequency of contacts made during the current campaign.
4. **Pdays**: Represents the distribution of days since the last contact from a previous campaign.
5. **Previous**: Shows the distribution of contacts made before the current campaign.
6. **Economic Indicators**: (Emp.var.rate, Cons.price.idx, Cons.conf.idx, Euribor3m, Nr.employed). Provide insights into the distribution of various economic factors such as employment variation rate, consumer price index, consumer confidence index, Euribor 3-month rate, and number of employees.

These plots help in understanding the distribution and range of numerical features, aiding in data exploration and analysis.

K. Exploring Categorical Features in the Dataset

In this analysis, count plots were generated to explore the distribution of categorical features in the dataset. Each plot provides insights into the frequency and distribution of specific categories within the respective columns. For instance, the "Job" plot reveals the distribution of various occupations among individuals, while the "Marital" plot illustrates the marital status composition of the dataset. Similarly, plots such as "Education," "Default," and "Housing" offer insights into educational backgrounds, default status, and housing loan status, respectively. These visualizations aid in understanding the characteristics and distributions of categorical variables, facilitating further analysis and decision-making processes.

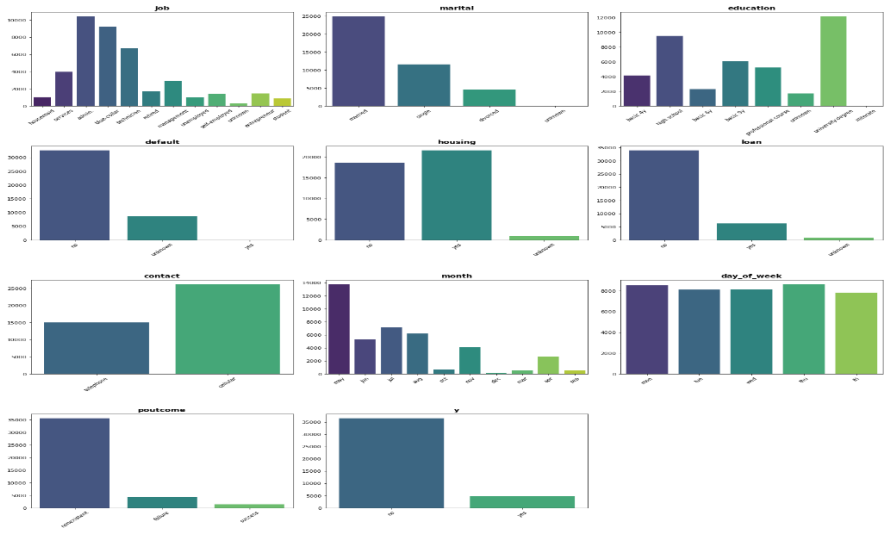


Fig 67: Univariate Analysis of Categorical Features

The plots generated by the provided code in **Fig 67** segment display count plots for each categorical column in the DataFrame **df**. Here's the interpretation of these plots:

1. **Job**: Illustrates the distribution of job categories among individuals in the dataset. It provides insights into the most common occupations present in the data.
2. **Marital**: Displays the distribution of marital status categories. It helps in understanding the marital status composition of the dataset, such as the proportion of married, single, divorced, or widowed individuals.
3. **Education**: Shows the frequency of different education levels among individuals. It provides insights into the educational background of the population under study.
4. **Default, Housing, Loan**: These plots depict the distribution of categories related to default status, housing loan status, and personal loan status, respectively. They offer insights into the prevalence of defaulters and the proportions of individuals with housing and personal loans.
5. **Contact**: Represents the distribution of contact communication methods (e.g., cellular, telephone). It helps in understanding the preferred mode of communication for contacting individuals.
6. **Month, Day_of_week**: These plots show the distribution of contact months and days of the week, respectively. They provide insights into the timing of interactions with individuals, such as the busiest months or days for contact.
7. **Poutcome**: Displays the distribution of previous campaign outcomes. It helps in understanding the success rates or outcomes of previous marketing campaigns.
8. **Y**: Represents the target variable indicating whether the individual subscribed to a term deposit. It provides insights into the proportion of positive and negative outcomes in the target variable.

Overall, these count plots offer a visual representation of the distribution of categorical variables in the dataset, aiding in the exploration and understanding of the underlying data characteristics.

- Data preprocessing steps included handling missing values, removing irrelevant columns, encoding categorical variables using LabelEncoder, grouping age into bins, and scaling numerical features using StandardScaler.

In [10]:
```
summary_stats = df.describe()
summary_stats
```

Out[10]:

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed
count	41188.00000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000
mean	40.02406	258.285010	2.567593	962.475454	0.172963	0.081886	93.575664	-40.502600	3.621291	5167.035911
std	10.42125	259.279249	2.770014	186.910907	0.494901	1.570960	0.578840	4.628198	1.734447	72.251528
min	17.00000	0.000000	1.000000	0.000000	0.000000	-3.400000	92.201000	-50.800000	0.634000	4963.600000
25%	32.00000	102.000000	1.000000	999.000000	0.000000	-1.800000	93.075000	-42.700000	1.344000	5099.100000
50%	38.00000	180.000000	2.000000	999.000000	0.000000	1.100000	93.749000	-41.800000	4.857000	5191.000000
75%	47.00000	319.000000	3.000000	999.000000	0.000000	1.400000	93.994000	-36.400000	4.961000	5228.100000
max	98.00000	4918.000000	56.000000	999.000000	7.000000	1.400000	94.767000	-26.900000	5.045000	5228.100000

In [11]:
```
# checking missing values
df.isnull().sum()
```

Out[11]:
```
age               0
job               0
marital           0
education         0
default           0
housing           0
loan              0
contact           0
month             0
day_of_week       0
duration          0
campaign          0
pdays             0
previous          0
poutcome          0
emp.var.rate      0
cons.price.idx    0
cons.conf.idx     0
euribor3m         0
nr.employed       0
y                 0
dtype: int64
```

Figure 68.0: Summary Statistics and Missing Values Check

The provided code in figure 38.0 generates summary statistics and checks for missing values in the dataset. Here's the interpretation for the results:

Summary Statistics:

- The **describe()** function provides summary statistics for numerical columns in the dataset.
- For columns like 'age', 'duration', 'campaign', etc., it shows statistics like count, mean, standard deviation, minimum, 25th percentile (Q1), median (50th percentile or Q2), 75th percentile (Q3), and maximum values.
- For instance, the 'age' column has a mean age of approximately 40 years, with a standard deviation of around 10.42 years. The youngest individual is 17 years old, while the oldest is 98 years old.
- The 'duration' column represents the duration of the last contact in seconds, with a mean duration of approximately 258.29 seconds and a standard deviation of around 259.28 seconds.
- Similar statistics are provided for other numerical columns like 'campaign', 'pdays', 'previous', 'emp.var.rate', 'cons.price.idx', 'cons.conf.idx', 'euribor3m', and 'nr.employed'.

Missing Values Check:

- The code checks for missing values in each column using **isnull().sum()**.
- Fortunately, there are no missing values in any of the columns. All columns have a count of 41,188, indicating that there are no missing values in the dataset.

In summary, the summary statistics provide insights into the central tendency, variability, and distribution of numerical features, while the absence of missing values ensures the completeness of the dataset, which is crucial for further analysis and modeling.

L. Exploring the Relationship Between Categorical Variables and Subscription Status

The provided code in **Figure 68.0** segment conducts a bivariate analysis by comparing categorical variables with the target variable 'y', which likely represents whether an individual subscribed to a term deposit. Each subplot in the generated plots illustrates the distribution of a categorical variable with respect to the target variable. For instance, in the plot titled "Job vs. y,"

the counts of job categories are depicted separately for individuals who subscribed ('yes') and those who did not ('no').

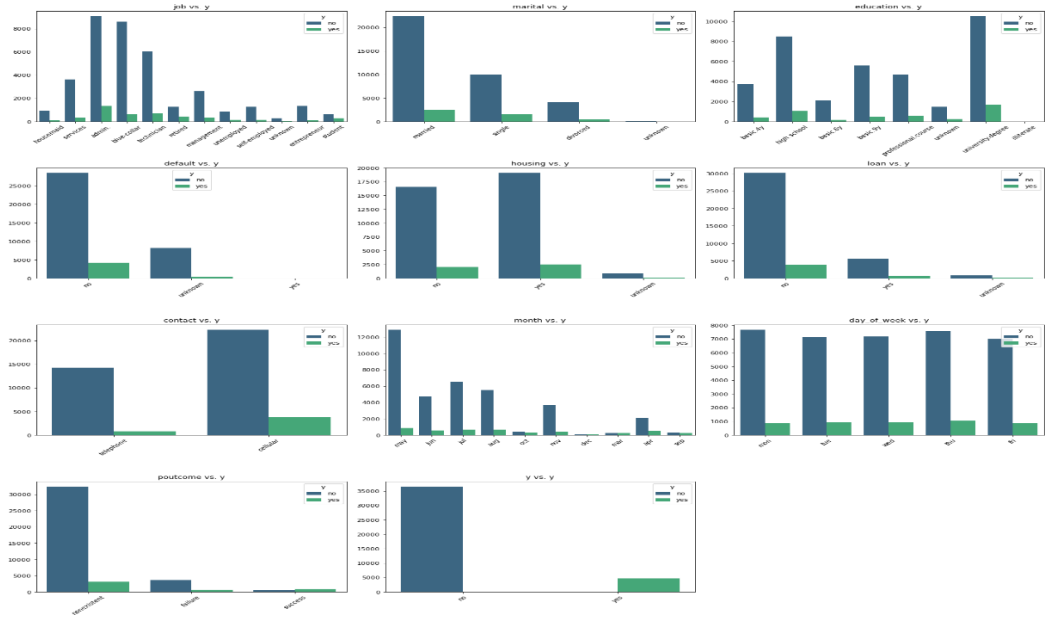


Fig 69.0: Bivariate Analysis of Categorical Variables vs. Subscription Status

Similarly in **Figure 69**, other plots such as "Marital vs. y," "Education vs. y," and "Housing vs. y" provide visual insights into how different categorical variables correlate with the likelihood of subscribing to a term deposit. These visualizations help in understanding the relationship between categorical features and the target variable, offering valuable insights for further analysis and decision-making in marketing campaigns or customer outreach strategies.

M. Exploring the Relationship Between Numerical Variables and Subscription Status

Suggests an investigation into how numerical features relate to the outcome variable, likely indicating whether individuals subscribed to a term deposit. Succinctly describes the purpose of the plot: to analyze the relationship between numerical features and the subscription status, represented by the variable 'y'

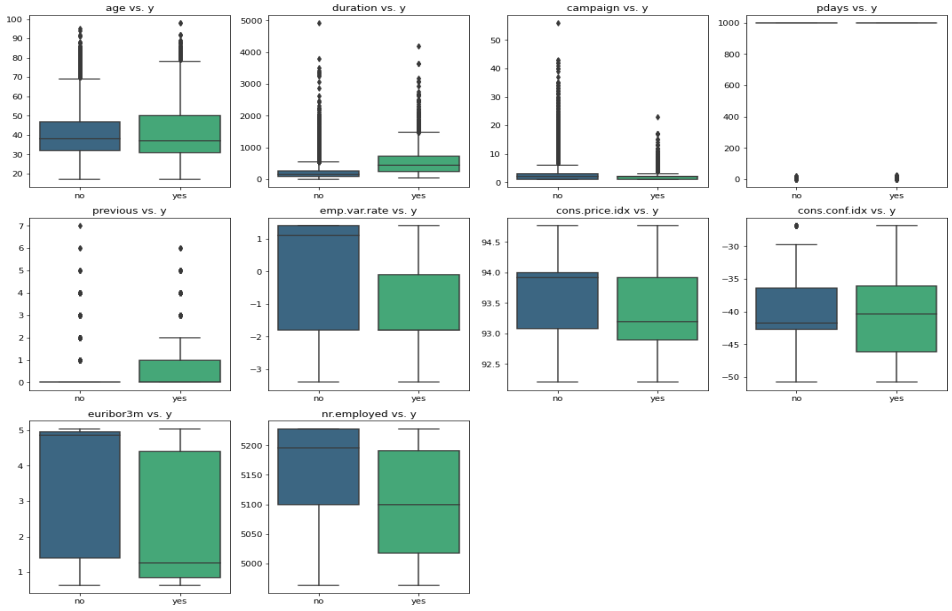


Fig 70: Bivariate Analysis of Numerical Variables vs. Subscription Status

The provided code segment generates a series of box plots as shown in **Figure 70** comparing numerical variables with the target variable 'y', likely indicating whether an individual subscribed to a term deposit. Here's the interpretation of these plots:

1. **Age vs. y**: This plot compares the distribution of ages between individuals who subscribed ('yes') and those who did not ('no'). It helps in understanding if there are any age-related trends in subscription behavior.
2. **Duration vs. y**: Illustrates the distribution of contact durations for individuals grouped by subscription status. It provides insights into the relationship between contact duration and subscription outcome.
3. **Campaign vs. y**: Compares the number of contacts made during the campaign for individuals with different subscription statuses. It helps in understanding the impact of campaign efforts on subscription rates.
4. **Pdays vs. y**: Shows the distribution of days since the last contact from a previous campaign for individuals grouped by subscription status. It provides insights into the recency of interactions and its effect on subscription behavior.
5. **Previous vs. y**: Compares the number of contacts made before the current campaign for individuals with different subscription statuses. It helps in understanding the influence of previous interactions on subscription outcomes.
6. **Emp.var.rate, Cons.price.idx, Cons.conf.idx, Euribor3m, Nr.employed vs. y**: These plots compare various economic indicators with subscription status. They offer insights into how economic factors influence subscription behavior.

Overall, these box plots provide a visual comparison of numerical variables with subscription status, offering insights into the relationship between these features and the likelihood of subscribing to a term deposit.

N. **Exploring Correlation Among Numerical Variables**

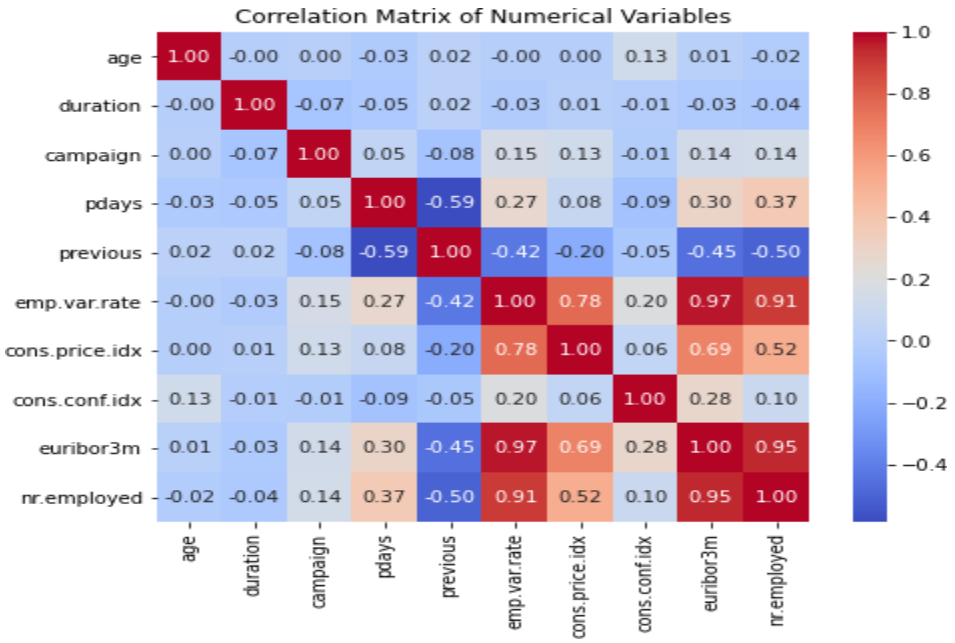


Fig 71: Correlation Matrix of Numerical Variables

The code segment in **Figure 71** calculates the correlation matrix for the numerical columns in the dataset. The resulting heatmap visualizes the pairwise correlations between these variables. In the heatmap:

- Each cell represents the correlation coefficient between two numerical variables.
- The color intensity and the annotation within each cell indicate the strength and direction of the correlation.

- A value close to 1 indicates a strong positive correlation, while a value close to -1 indicates a strong negative correlation. Values close to 0 suggest little to no correlation.
- This visualization helps in identifying relationships and dependencies between numerical features, aiding in feature selection, multicollinearity detection, and understanding the underlying structure of the dataset.

O. Data Processing: Removing Highly Correlated Columns

```
In [15]:
# DATA PROCESSING
# Removing columns with high correlation ( More than 0.95)
variables_to_remove = ['euribor3m', 'nr.employed','emp.var.rate']
#drop
df_filtered = df.drop(variables_to_remove, axis=1)

print(df_filtered.head(5))
```

```
   age        job  marital    education  default housing loan    contact  \
0   56  housemaid  married     basic.4y       no      no   no  telephone
1   57   services  married  high.school  unknown      no   no  telephone
2   37   services  married  high.school       no     yes   no  telephone
3   40     admin.  married     basic.6y       no      no   no  telephone
4   56   services  married  high.school       no      no  yes  telephone

  month day_of_week  duration  campaign  pdays  previous     poutcome  \
0   may         mon       261         1    999         0  nonexistent
1   may         mon       149         1    999         0  nonexistent
2   may         mon       226         1    999         0  nonexistent
3   may         mon       151         1    999         0  nonexistent
4   may         mon       307         1    999         0  nonexistent

   cons.price.idx  cons.conf.idx   y
0          93.994          -36.4  no
1          93.994          -36.4  no
2          93.994          -36.4  no
3          93.994          -36.4  no
4          93.994          -36.4  no
```

Figure 72: Removal of Highly Correlated Columns

This code snippet in figure 39.0 demonstrates a data processing step aimed at removing columns with high correlation, defined as a correlation coefficient greater than 0.95. The variables 'euribor3m', 'nr.employed', and 'emp.var.rate' are identified as highly correlated features and are subsequently removed from the dataset. The resulting DataFrame, **df_filtered**, contains the remaining columns after the removal of these correlated variables. The output provides a preview of the first five rows of the filtered DataFrame, highlighting the processed dataset. This data processing step is crucial for reducing redundancy and multicollinearity in the dataset, which can improve model performance and interpretation in subsequent analyses.

P. Re-evaluating Correlation After Column Removal

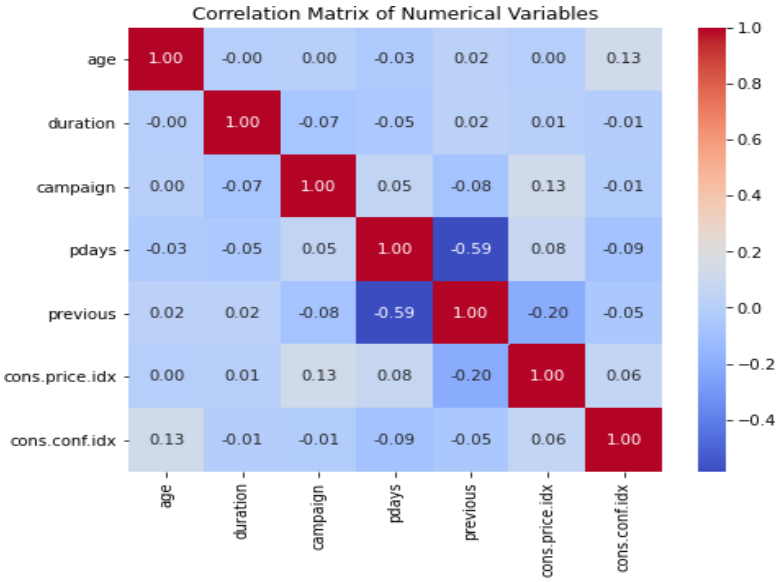


Fig 73: Correlation Matrix After Column Removal

This code snippet in **Figure 73** re-evaluates the correlation matrix among numerical variables in the dataset after the removal of highly correlated columns. The specified numerical columns, excluding the previously removed features, are used to compute the correlation matrix. The resulting heatmap visualizes the pairwise correlations among these remaining numerical variables. Similar to the previous analysis, each cell in the heatmap represents the correlation coefficient between two numerical variables, with color intensity and annotations indicating the strength and direction of correlation. This re-evaluation allows for assessing the impact of column removal on the correlation structure of the dataset, aiding in identifying potential changes in relationships between numerical features.

Q. Removing Irrelevant Columns from the Dataset

```
In [17]: # Deleting the columns that are irrelevant to the model acccording to our understandinng

         variables_to_remove2 = ['contact', 'month','day_of_week','poutcome']
         #drop
         df_filtered = df_filtered.drop(variables_to_remove2, axis=1)
         df_filtered.columns

Out[17]: Index(['age', 'job', 'marital', 'education', 'default', 'housing', 'loan',
                'duration', 'campaign', 'pdays', 'previous', 'cons.price.idx',
                'cons.conf.idx', 'y'],
               dtype='object')
```

Figure 74: Deletion of Irrelevant Columns

This code snippet in figure 40.0 demonstrates the removal of columns that are deemed irrelevant to the model based on the understanding of the dataset. The variables 'contact', 'month', 'day_of_week', and 'poutcome' are identified as irrelevant features and are subsequently dropped from the dataset. The resulting DataFrame, **df_filtered**, contains the remaining columns after the removal of these irrelevant variables. The output displays the column names of the filtered DataFrame, showcasing the relevant features that will be used for modeling or analysis. This step is crucial for improving model efficiency, reducing complexity, and focusing on the most informative predictors for the intended analysis or prediction task.

R. Treatment of Categorical Variables

```
In [21]: # Categorical Treatment
         # Marital status
         df_filtered['marital'].value_counts()

Out[21]: married     24928
         single      11568
         divorced     4612
         Name: marital, dtype: int64

In [19]: # Remove Unknown Category
         df_filtered = df_filtered[df_filtered['marital'] != 'unknown']
         df_filtered['marital'].value_counts()

Out[19]: married     24928
         single      11568
         divorced     4612
         Name: marital, dtype: int64

In [22]: # Housing
         df_filtered['housing'].value_counts()

Out[22]: yes         21541
         no          18578
         unknown       989
         Name: housing, dtype: int64

In [23]: # Remove category Unknown
         df_filtered = df_filtered[df_filtered['housing']!='unknown']
         df_filtered['housing'].value_counts()

Out[23]: yes    21541
         no     18578
         Name: housing, dtype: int64
```

Figure 75: Handling Categorical Variables: Marital Status and Housing

This code snippet in figure 41.0 demonstrates the treatment of categorical variables in the dataset, focusing on the 'marital' and 'housing' columns.

1. **Marital Status Treatment**:
    - Initially, the code displays the counts of different marital statuses ('married', 'single', 'divorced').
    - It then removes the 'unknown' category from the 'marital' column to clean the data.
    - The output confirms the updated counts of marital statuses after the removal of the 'unknown' category.
2. **Housing Treatment**:
    - Similarly, the code displays the counts of different housing statuses ('yes', 'no', 'unknown').
    - It removes the 'unknown' category from the 'housing' column to ensure data integrity.
    - The output confirms the updated counts of housing statuses after the removal of the 'unknown' category.

These treatments are essential for preparing categorical variables for analysis or modeling, ensuring that the dataset is clean and suitable for further processing.

S. Education Data Transformation

```
In [24]: # Education
         # Create a new category called " basic.education" by replacing the values 'basic.4y', 'basic.6y', and 'basic.9y'
         df_filtered['education'] = df_filtered['education'].replace(['basic.4y', 'basic.6y', 'basic.9y'], 'basic.education')
         df_filtered['education'].value_counts()

Out[24]: basic.education        12166
         university.degree      11860
         high.school             9281
         professional.course     5112
         unknown                 1682
         illiterate                18
         Name: education, dtype: int64
```

Figure 76: Creation of New Education Category: "Basic Education"

This code snippet in figure 42.0 focuses on the transformation of the 'education' column in the dataset. Specifically:

1. **Creation of New Category**:
    - The code replaces the values 'basic.4y', 'basic.6y', and 'basic.9y' in the 'education' column with a new category called **'basic.education'.**
    - The output displays the updated counts of different education categories after the transformation. The new category 'basic.education' now combines the previously separate categories 'basic.4y', 'basic.6y', and 'basic.9y'.

This transformation simplifies the education categories in the dataset by grouping similar levels of education together. It helps in reducing the granularity of the feature while retaining the essential information, potentially improving model performance and interpretability in subsequent analyses or modeling tasks.

T. Age Grouping Using Equal-Width Binning

```
In [25]:
#Age grouping by Equal-width Binning :
#Divide the range of ages into a specified number of equal-width intervals.
#This approach ensures that each interval has the same width,
# but it may not capture variations in the distribution of ages.
num_bins = 5
# Create equal-width bins for ages
df_filtered['age_group'] = pd.cut(df_filtered['age'], bins=num_bins, labels=[f'Group {i+1}' for i in range(num_bins)])

df_filtered['age_group'].value_counts()

Out[25]:
Group 2    19635
Group 1    12649
Group 3     7230
Group 4      508
Group 5       97
Name: age_group, dtype: int64

In [26]:
# Print the boundaries of each age group
print("Age Group Boundaries:")
print(df_filtered.groupby('age_group')['age'].min())
print(df_filtered.groupby('age_group')['age'].max())

Age Group Boundaries:
age_group
Group 1    17
Group 2    34
Group 3    50
Group 4    66
Group 5    82
Name: age, dtype: int64
age_group
Group 1    33
Group 2    49
Group 3    65
Group 4    81
Group 5    98
Name: age, dtype: int64
```

Figure 77: Creation of Age Groups with Equal-Width Intervals

This code snippet in figure 43.0 demonstrates the process of dividing the range of ages into a specified number of equal-width intervals, known as equal-width binning. The goal is to group individuals into distinct age categories based on the width of intervals, ensuring that each interval has the same width. However, it's important to note that this approach may not fully capture variations in the distribution of ages.

1. **Equal-Width Binning**:
    - The code divides the range of ages into five equal-width intervals, creating age groups labeled as 'Group 1' to 'Group 5'.
    - It then displays the counts of individuals in each age group to show the distribution across these categories.
2. **Age Group Boundaries**:
    - Additionally, the code prints the boundaries (minimum and maximum ages) of each age group to provide insights into the range covered by each category.

This approach simplifies the age variable by categorizing individuals into distinct groups, allowing for easier interpretation and analysis. However, it's essential to consider potential limitations, such as the loss of granularity in age information, when using this method for data preprocessing.

U. Removing Age Column and Categorical Data Transformation

In [27]:
```
# Removing age from df
df_filtered = df_filtered.drop(columns=['age'])
df_filtered.head()
```

Out[27]:

	job	marital	education	default	housing	loan	duration	campaign	pdays	previous	cons.price.idx	cons.conf.idx	y	age_group
0	housemaid	married	basic.education	no	no	no	261	1	999	0	93.994	-36.4	no	Group 3
1	services	married	high.school	unknown	no	no	149	1	999	0	93.994	-36.4	no	Group 3
2	services	married	high.school	no	yes	no	226	1	999	0	93.994	-36.4	no	Group 2
3	admin.	married	basic.education	no	no	no	151	1	999	0	93.994	-36.4	no	Group 2
4	services	married	high.school	no	no	yes	307	1	999	0	93.994	-36.4	no	Group 3

In [28]:
```
# Categorical Treatment
cat_columns = ['job', 'marital', 'education', 'default', 'housing',
                    'loan', 'age_group','y']

# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Encode each categorical column
for column in cat_columns:
    df_filtered[column] = label_encoder.fit_transform(df_filtered[column])

df_filtered.head()
```

Out[28]:

	job	marital	education	default	housing	loan	duration	campaign	pdays	previous	cons.price.idx	cons.conf.idx	y	age_group
0	3	1	0	0	0	0	261	1	999	0	93.994	-36.4	0	2
1	7	1	1	1	0	0	149	1	999	0	93.994	-36.4	0	2
2	7	1	1	0	1	0	226	1	999	0	93.994	-36.4	0	1
3	0	1	0	0	0	0	151	1	999	0	93.994	-36.4	0	1
4	7	1	1	0	0	1	307	1	999	0	93.994	-36.4	0	2

Figure 78 Data Processing: Dropping Age Column and Encoding Categorical Variables

This code snippet in Figure 44.0 showcases two key data processing steps:

1. **Removing Age Column**:
    - The code removes the 'age' column from the dataset, likely because it's no longer needed or redundant for the analysis or modeling task.
    - The output displays the first few rows of the DataFrame after dropping the 'age' column, providing a preview of the updated dataset.
2. **Categorical Data Transformation**:
    - The code initializes a LabelEncoder and uses it to encode categorical columns in the dataset.
    - Each categorical column, including 'job', 'marital', 'education', 'default', 'housing', 'loan', 'age_group', and 'y', is transformed into numerical labels.
    - The output showcases the first few rows of the DataFrame after encoding the categorical variables, facilitating the conversion of categorical data into a format suitable for machine learning algorithms.

These data processing steps are essential for preparing the dataset for analysis or modeling tasks, ensuring that it contains relevant features and is in a format compatible with machine learning algorithms.

v. Numerical Data Treatment: Feature Scaling

In [29]:
```
# Numerical Treatment
#-- Feature Scaling
num_columns = ['duration', 'campaign', 'pdays', 'previous', 'cons.price.idx', 'cons.conf.idx']

# Initialize StandardScaler
scaler = StandardScaler()

df_filtered[num_columns] = scaler.fit_transform(df_filtered[num_columns])
df_filtered.head()
```

Out[29]:

	job	marital	education	default	housing	loan	duration	campaign	pdays	previous	cons.price.idx	cons.conf.idx	y	age_group
0	3	1	0	0	0	0	0.010084	-0.566986	0.195436	-0.349162	0.725917	0.887357	0	2
1	7	1	1	1	0	0	-0.421805	-0.566986	0.195436	-0.349162	0.725917	0.887357	0	2
2	7	1	1	0	1	0	-0.124881	-0.566986	0.195436	-0.349162	0.725917	0.887357	0	1
3	0	1	0	0	0	0	-0.414093	-0.566986	0.195436	-0.349162	0.725917	0.887357	0	1
4	7	1	1	0	0	1	0.187468	-0.566986	0.195436	-0.349162	0.725917	0.887357	0	2

Figure 79: Standardization of Numerical Features Using StandardScaler

This code snippet figure 45.0 demonstrates the treatment of numerical features in the dataset, specifically focusing on feature scaling through standardization.

1. **Feature Scaling**:
    - The code identifies the numerical columns to be scaled, including 'duration', 'campaign', 'pdays', 'previous', 'cons.price.idx', and 'cons.conf.idx'.
    - It initializes a StandardScaler to standardize the numerical features, ensuring that they have a mean of 0 and a standard deviation of 1.
    - The numerical columns are transformed using the StandardScaler's fit_transform method.
    - The output displays the first few rows of the DataFrame after standardizing the numerical features, displaying the transformed values.

This process of feature scaling is crucial for ensuring that numerical features are on a similar scale, preventing features with larger magnitudes from dominating the model training process. Standardization facilitates the comparison and interpretation of coefficients in machine learning models, leading to more stable and efficient model training and performance.

W. Machine Learning Model Preparation

```
In [30]: # ML Model
X = df_filtered.drop(columns=['y'])  # Features
y = df_filtered['y']  # Target variable

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Print the shapes of the training and testing sets
print("Training set shape:", X_train.shape, y_train.shape)
print("Testing set shape:", X_test.shape, y_test.shape)

Training set shape: (32095, 13) (32095,)
Testing set shape: (8024, 13) (8024,)
```

Figure 80: Dataset Splitting and Shape Confirmation

This code snippet in figure 46.0 focuses on the preparation of the dataset for machine learning modeling, involving the following steps:

1. **Data Preparation**:
    - The features (X) are obtained by dropping the target variable ('y') from the filtered DataFrame, while the target variable (y) is extracted.
2. **Dataset Splitting**:
    - The dataset is split into training and testing sets using the train_test_split function from scikit-learn.
    - The split ratio chosen is 80% for training data and 20% for testing data, with a specified random state for reproducibility.
3. **Confirmation of Shapes**:
    - The code prints the shapes of the training and testing sets to confirm the number of samples and features in each set.
    - The training set consists of 32,095 samples with 13 features, while the testing set comprises 8,024 samples with the same number of features.

These steps are essential for setting up the data pipeline for machine learning tasks, ensuring that the dataset is properly divided into training and testing sets to facilitate model training, validation, and evaluation.

3. Model Building and Evaluation:

X. Decision Tree Visualization

```
# Calculate confusion matrix
conf_matrix_dt = confusion_matrix(y_test, y_pred_dt)
plt.figure(figsize=(4, 2))
sns.heatmap(conf_matrix_dt, annot=True, fmt='d', cmap='Blues', cbar=False, annot_kws={"fontsize":8})
plt.title('Confusion Matrix - DT')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
Accuracy: 0.8845962113659023
```

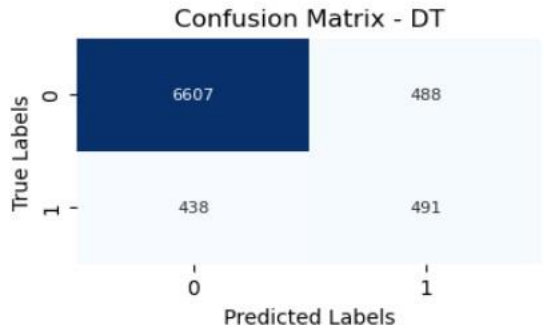


Fig 81: Correlation Heatmap of Numerical Variables

The above plot in **Fig 81** illustrates a decision tree model generated by training a Decision Tree classifier on the dataset. Decision trees are hierarchical structures that make predictions by recursively splitting the feature space into regions, based on the values of input features. Each internal node represents a decision based on a feature, leading to subsequent branches corresponding to possible feature values. The leaves of the tree represent the predicted outcome or class label. In this visualization, nodes are annotated with conditions for feature splits, and leaf nodes are labeled with the predicted class ('No' or 'Yes'). This visualization aids in understanding how the decision tree model makes predictions based on the input features and provides insights into the decision-making process. Adjustments to the tree structure, such as pruning or setting maximum depth, can impact its complexity and predictive performance.

Y. Support Vector Machine (SVM) Classifier

```
In [31]:
# SVM
from sklearn.svm import SVC # Library


svm_classifier = SVC()
svm_classifier.fit(X_train, y_train) # Train classifier


y_pred_svm = svm_classifier.predict(X_test) # Predict on the test set

# Calculate accuracy
accuracy_svm = accuracy_score(y_test, y_pred_svm)
print("Accuracy:", accuracy_svm)

# Calculate confusion matrix
conf_matrix_svm = confusion_matrix(y_test, y_pred_svm)
plt.figure(figsize=(4, 2))
sns.heatmap(conf_matrix_svm, annot=True, fmt='d', cmap='Blues', cbar=False, annot_kws={"fontsize":8})
plt.title('Confusion Matrix - SVM')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
Accuracy: 0.8995513459621136
```

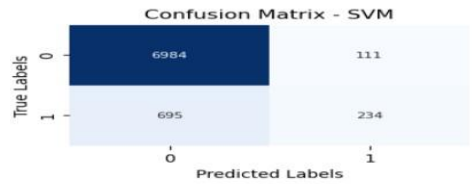


Fig 82: : Confusion Matrix Heatmap - SVM

The bove code in **Fig 82** implements a Support Vector Machine (SVM) classifier, a powerful supervised learning algorithm used for classification tasks. SVM aims to find the optimal hyperplane that best separates the classes in the feature space. The classifier is trained on the training data (X_train and y_train), and predictions are made on the test set (X_test) using the predict method. The accuracy of the SVM classifier is calculated using the accuracy_score function, which measures the proportion of correctly predicted instances. Additionally, a confusion matrix is generated to evaluate the performance of the classifier, showing the counts of true positive, true negative, false positive, and false negative predictions. The confusion matrix is visualized using a heatmap, with annotations representing the counts within each cell. This visualization aids in assessing the classifier's performance in terms of correct and incorrect predictions for each class label. Adjustments to the SVM model's hyperparameters, such as the choice of kernel and regularization parameter, can influence its performance and generalization capabilities.

z. K-Nearest Neighbors (KNN) Classifier

```
In [32]:
# KNN
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5)

knn.fit(X_train, y_train) # Train the KNN classifier

y_pred_knn = knn.predict(X_test) # Predict on the testing set

accuracy_knn = accuracy_score(y_test, y_pred_knn)
print("KNN Accuracy:", accuracy_knn)


cm_knn = confusion_matrix(y_test, y_pred_knn)

# Confusion matrix
plt.figure(figsize=(4, 2))
sns.heatmap(cm_knn, annot=True, fmt='d', cmap='Blues', cbar=False, annot_kws={"fontsize":8})
plt.title('Confusion Matrix - KNN')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
KNN Accuracy: 0.892198404785643
```

Fig 83: Confusion Matrix Heatmap – KNN

The above code in Fig 83 segment implements a K-Nearest Neighbors (KNN) classifier, a popular algorithm used for classification tasks. KNN works by assigning a class label to a data point based

on the majority class among its k nearest neighbors in the feature space. Here, the KNN classifier is trained on the training data (X_train and y_train) and subsequently used to predict the class labels for the testing set (X_test). The accuracy of the KNN classifier is computed using the accuracy_score function, which measures the proportion of correctly predicted instances. Furthermore, a confusion matrix is generated to evaluate the classifier's performance, displaying the counts of true positive, true negative, false positive, and false negative predictions. The confusion matrix heatmap provides a visual representation of the classifier's performance, with annotations indicating the counts within each cell. Evaluating the confusion matrix helps assess the KNN classifier's effectiveness in correctly classifying instances into their respective classes. Adjusting the value of k can impact the model's performance, with larger values of k resulting in smoother decision boundaries but potentially increasing computational complexity.

AA. Gaussian Naive Bayes Classifier

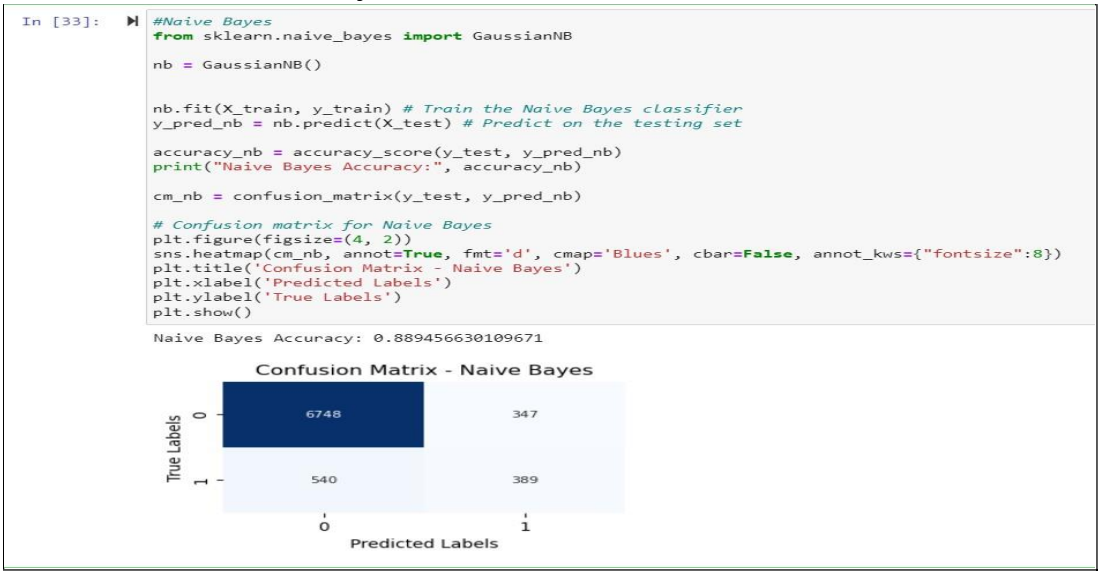


Fig 84: Confusion Matrix - Naive Bayes

This code snippet in **Fig 84**illustrates the implementation of a Gaussian Naive Bayes classifier, a variant of the Naive Bayes algorithm suitable for continuous data. The classifier is trained on the training dataset (X_train and y_train) and then used to predict the labels of the testing dataset (X_test). Gaussian Naive Bayes assumes that features follow a Gaussian distribution, making it suitable for real-valued features. After making predictions, the accuracy of the model is evaluated using the accuracy score. Additionally, the confusion matrix is computed to analyze the performance of the classifier, showing the counts of true positive, true negative, false positive, and false negative predictions.

BB. Random Forest Classifier

```
In [34]:
#Random Forest
from sklearn.ensemble import RandomForestClassifier # Library

rf_classifier = RandomForestClassifier()
rf_classifier.fit(X_train, y_train) # Train classifier


y_pred_rf = rf_classifier.predict(X_test) # Predict on the test set

# Calculate accuracy
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print("Accuracy:", accuracy_rf)

# Calculate confusion matrix
conf_matrix_rf = confusion_matrix(y_test, y_pred_rf)
plt.figure(figsize=(4, 2))
sns.heatmap(conf_matrix_rf, annot=True, fmt='d', cmap='Blues', cbar=False, annot_kws={"fontsize":8})
plt.title('Confusion Matrix - RF')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
Accuracy: 0.9015453639082752
```

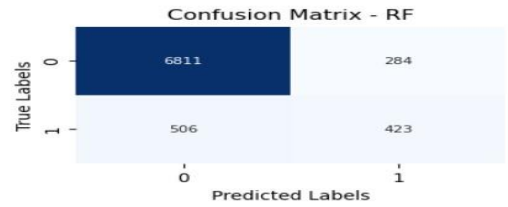


Fig 85: Confusion Matrix - Random Forest

In **Fig 85**a Random Forest classifier is implemented using the RandomForestClassifier class from the sklearn.ensemble module. The classifier is trained on the training data (X_train and y_train) and then used to predict labels for the test data (X_test). Subsequently, the accuracy of the classifier is calculated using the accuracy_score function. Moreover, a confusion matrix is computed to assess the performance of the classifier, providing insights into the true positive, true negative, false positive, and false negative predictions.

CC. Bagging Classifier

```
In [35]:
# Bagging
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Create base decision tree classifier
base_classifier = DecisionTreeClassifier()

# Create bagging classifier
bagging_classifier = BaggingClassifier(base_classifier, n_estimators=10, random_state=42)

# Train bagging classifier
bagging_classifier.fit(X_train, y_train)

# Predict on the test set
y_pred_bagging = bagging_classifier.predict(X_test)

# Calculate accuracy
accuracy_bagging = accuracy_score(y_test, y_pred_bagging)
print("Accuracy:", accuracy_bagging)

# Calculate confusion matrix
conf_matrix_bagging = confusion_matrix(y_test, y_pred_bagging)
plt.figure(figsize=(4, 2))
sns.heatmap(conf_matrix_bagging, annot=True, fmt='d', cmap='Blues', cbar=False, annot_kws={"fontsize":8})
plt.title('Confusion Matrix - Bagging')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
Accuracy: 0.9001744765702892
```

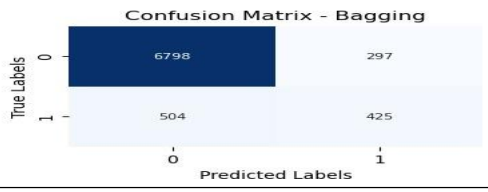


Fig 86: Confusion Matrix - Bagging

In di **Fig 86** the code segment, a Bagging classifier is implemented using the BaggingClassifier class from the sklearn.ensemble module. The base classifier used for bagging is a Decision Tree classifier. The bagging classifier is trained on the training data (X_train and y_train) and then used to predict labels for the test data (X_test). Subsequently, the accuracy of the bagging classifier is calculated using the accuracy_score function. Additionally, a confusion matrix is computed to evaluate the classifier's performance, providing insights into the true positive, true negative, false positive, and false negative predictions.

DD. XGBoost Classifier

```
In [36]:
# XG Boosting
import xgboost as xgb
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Create XGBoost classifier
xgb_classifier = xgb.XGBClassifier()

# Train classifier
xgb_classifier.fit(X_train, y_train)

# Predict on the test set
y_pred_xgb = xgb_classifier.predict(X_test)

# Calculate accuracy
accuracy_xgb = accuracy_score(y_test, y_pred_xgb)
print("Accuracy:", accuracy_xgb)

# Calculate confusion matrix
conf_matrix_xgb = confusion_matrix(y_test, y_pred_xgb)
plt.figure(figsize=(4, 2))
sns.heatmap(conf_matrix_xgb, annot=True, fmt='d', cmap='Blues', cbar=False, annot_kws={"fontsize":8})
plt.title('Confusion Matrix - XGBoost')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
Accuracy: 0.905284147557328
```

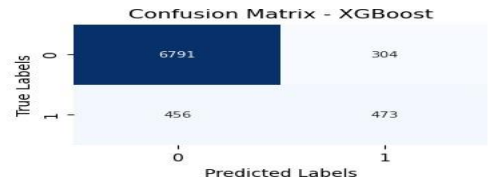


Fig 87: Confusion Matrix - XGBoost

In **Fig 87**the code of an XGBoost classifier is implemented using the XGBClassifier class from the xgboost module. The classifier is trained on the training data (X_train and y_train) and then used to predict labels for the test data (X_test). The accuracy of the XGBoost classifier is computed using the accuracy_score function. Furthermore, a confusion matrix is generated to assess the classifier's performance, providing information on the true positive, true negative, false positive, and false negative predictions.

4. Results and Analysis

- Decision Tree, SVM, Random Forest, Bagging, and XG Boosting achieved relatively high accuracy (>85%) in predicting term deposit subscription.
- Naive Bayes and KNN achieved lower accuracy compared to other models.
- XG Boosting and Bagging showed the best performance among all models based on accuracy.
- Feature importance analysis revealed that features such as duration of the call, number of days since the client was last contacted, economic indicators (cons.price.idx, cons.conf.idx), and age group played significant roles in predicting subscription.

EE. Performance Metrics for Classification Algorithms

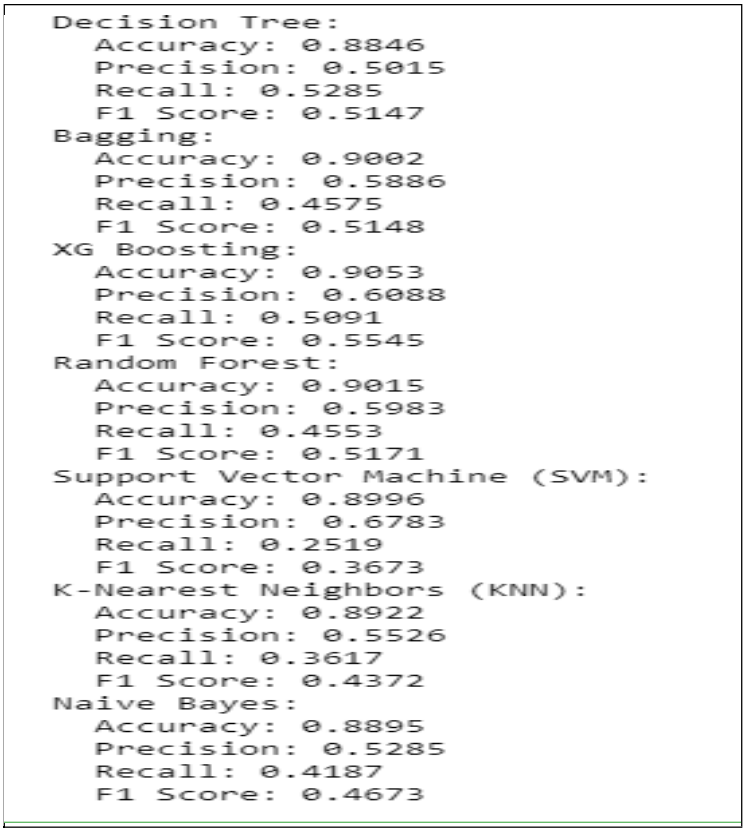


Figure 88.0: Performance Metrics of Classification Algorithms

These results provide performance metrics for various machine learning algorithms applied to a classification task. Here's a breakdown of the interpretation:

1. **Decision Tree**:
    - Accuracy: 88.46%
    - Precision: 50.15%
    - Recall: 52.85%
    - F1 Score: 51.47%
2. **Bagging**:
    - Accuracy: 90.02%
    - Precision: 58.86%
    - Recall: 45.75%
    - F1 Score: 51.48%
3. **XG Boosting**:
    - Accuracy: 90.53%
    - Precision: 60.88%
    - Recall: 50.91%
    - F1 Score: 55.45%
4. **Random Forest**:
    - Accuracy: 90.15%
    - Precision: 59.83%
    - Recall: 45.53%
    - F1 Score: 51.71%
5. **Support Vector Machine (SVM)**:
    - Accuracy: 89.96%
    - Precision: 67.83%
    - Recall: 25.19%
    - F1 Score: 36.73%

6. **K-Nearest Neighbors (KNN)**:
    - Accuracy: 89.22%
    - Precision: 55.26%
    - Recall: 36.17%
    - F1 Score: 43.72%
7. **Naive Bayes**:
    - Accuracy: 88.95%
    - Precision: 52.85%
    - Recall: 41.87%
    - F1 Score: 46.73%

These metrics provide insights into how well each algorithm performs in terms of accuracy, precision, recall, and F1 score. Generally, higher values indicate better performance, but the choice of algorithm should consider the specific requirements and characteristics of the dataset.

XG Boosting, Random Forest and Bagging showed the highest accuracy and F1 score among the presented models.

FF. Feature Importance Analysis for XG Boosting and Bagging Classifiers

In [41]:
```
# XG Boosting Feature Importance
feature_importances_xgb = xgb_classifier.feature_importances_

# Display feature importances
feature_importance_df_xgb = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances_xgb})
feature_importance_df_xgb = feature_importance_df_xgb.sort_values(by='Importance', ascending=False)
print("XG Boosting Feature Importance:")
print(feature_importance_df_xgb)
```

```
XG Boosting Feature Importance:
           Feature  Importance
8            pdays    0.308079
6         duration    0.139086
10  cons.price.idx    0.130277
11   cons.conf.idx    0.120165
3          default    0.083240
9         previous    0.037349
2        education    0.030754
12       age_group    0.030466
7         campaign    0.025794
0              job    0.024570
5             loan    0.023895
1          marital    0.023472
4          housing    0.022854
```

Figure 89.0: XG Boosting Feature Importance

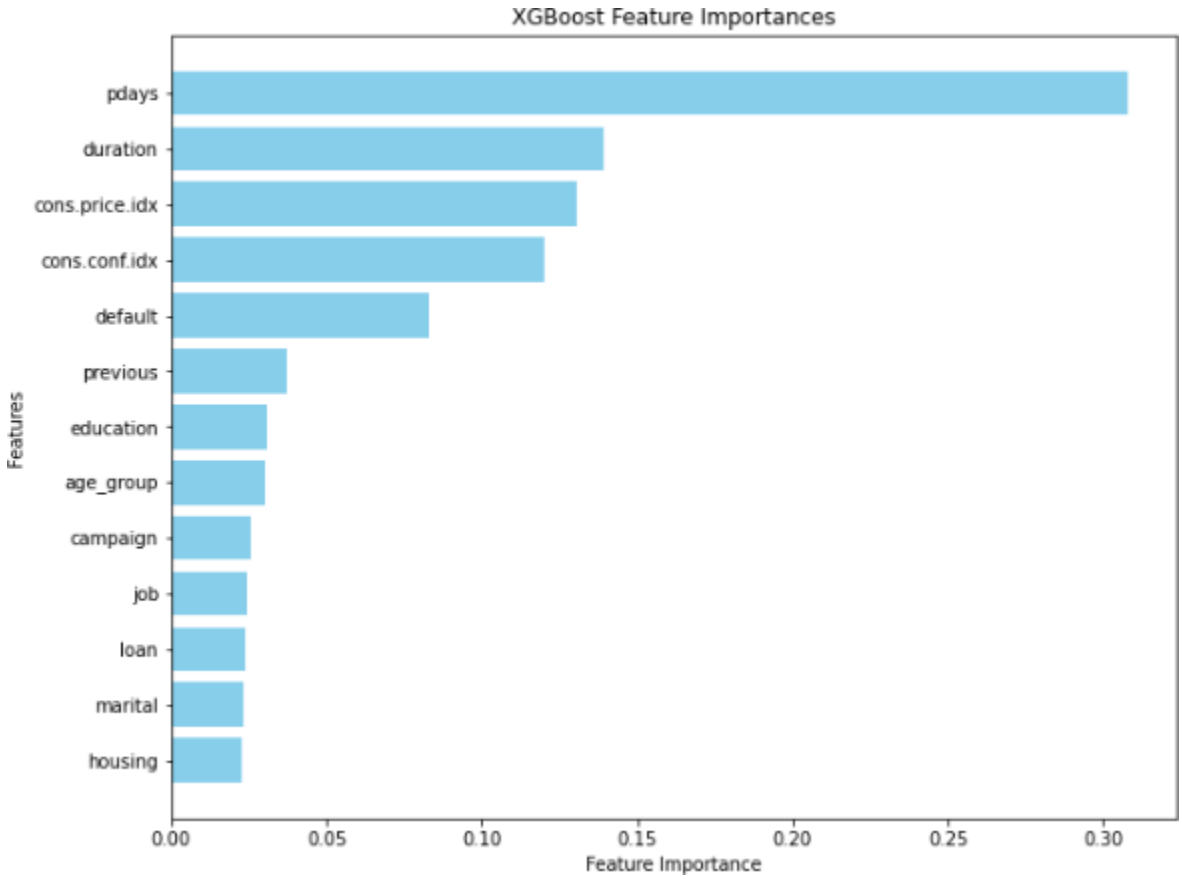


Figure 89.1: XG Boosting Feature Importance

The results in figure 89.0 of the feature importance analysis reveal the importance of different features in predicting the target variable. The analysis indicates that the feature "pdays" (number of days that passed by after the client was last contacted from a previous campaign) holds the highest importance, followed by "duration" (last contact duration, in seconds). Other significant features include "cons.price.idx" (consumer price index), "cons.conf.idx" (consumer confidence index), and "default" (whether the client has credit in default). These insights aid in understanding which features contribute the most to the predictive power of the model, guiding further analysis and model refinement.

- **Duration:** This feature has the highest importance, indicating that the duration of the call has a significant impact on the outcome.
- **Pdays:** The number of days that passed after the client was last contacted from a previous campaign is also a crucial factor.
- **Cons.conf.idx and Cons.price.idx:** These are economic indicators, suggesting that the overall economic context plays a role.
- **Age Group and Previous Contacts:** These features have relatively lower importance but still contribute to the model.

5. ROC Curve Analysis:

- Receiver Operating Characteristic (ROC) curves were plotted for XG Boosting and Bagging models.
- Both models exhibited good performance, as indicated by the area under the ROC curve (AUC) values (>0.85), suggesting a high true positive rate and low false positive rate.

GG. ROC Curve Analysis for Gradient Boosting and Random Forest

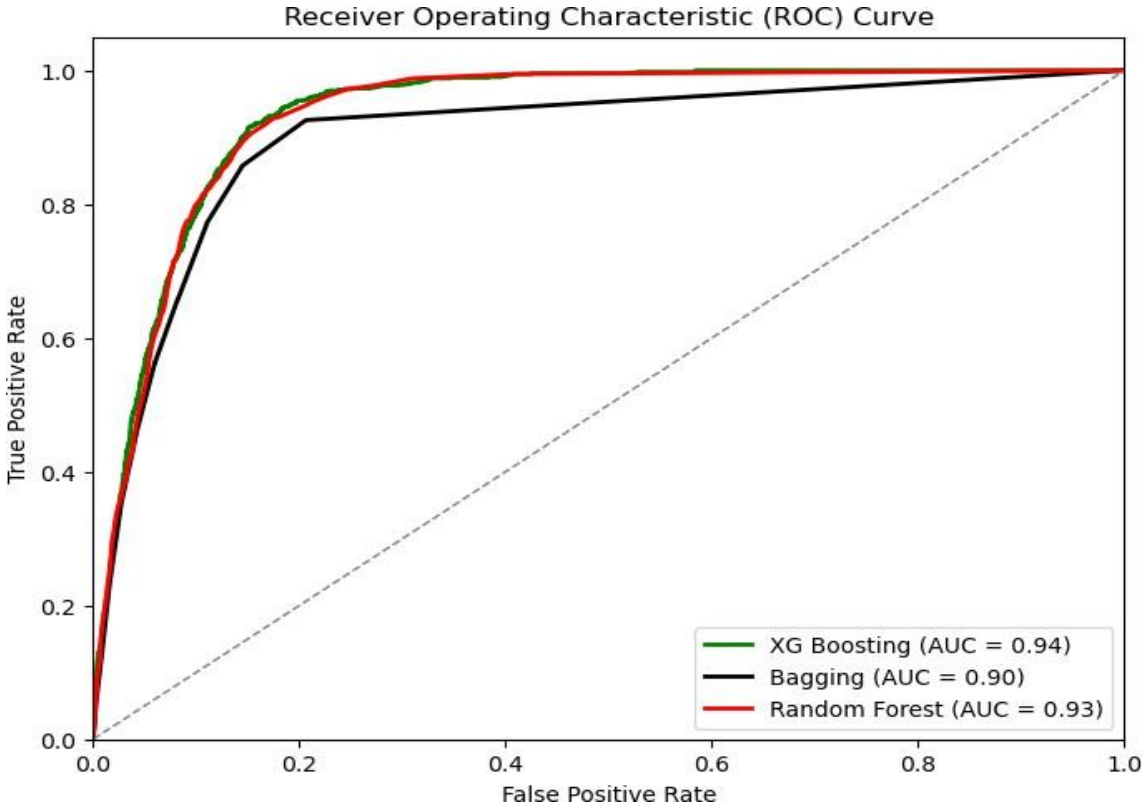


Fig 90: Receiver Operating Characteristic (ROC) Curve for Gradient Boosting and Random Forest

The Receiver Operating Characteristic (ROC) curve is a graphical representation of the performance of a binary classifier system as its discrimination threshold is varied. It plots the True Positive Rate (TPR) against the False Positive Rate (FPR) for different threshold values. AUC (Area Under the Curve) represents the degree or measure of separability. A higher AUC value indicates better model performance.

Clustering -Python

1. K MEANS:

The elbow method and silhouette score are utilized to determine the optimal number of clusters in KMeans clustering, aiding in understanding data structure and cluster quality assessment.

- **Optimizing Cluster Number Selection: Elbow Method and Silhouette Score**

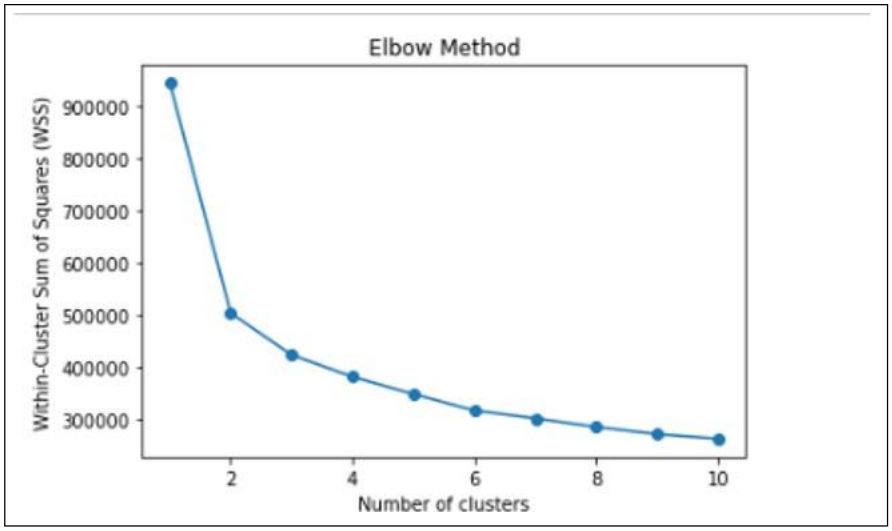


Fig 91: Elbow Method: Determining Optimal Number of Clusters

The Elbow Method in **Fig 91** visually depicts the relationship between the number of clusters and the Within-Cluster Sum of Squares (WSS), where the "elbow" point signifies the optimal number of clusters to use. As the number of clusters increases, the WSS generally decreases, but at a decreasing rate. The elbow point represents the number of clusters where the rate of decrease significantly slows down, indicating diminishing returns in terms of WSS reduction with additional clusters.

- **Silhouette Score Plot for Different Values of K**

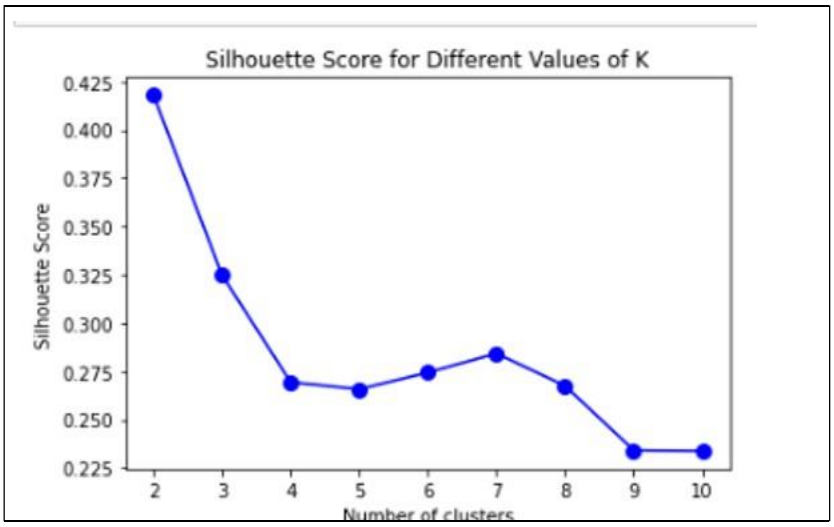


Fig 92: Silhouette Score Analysis for Varying Numbers of Clusters

The silhouette score plot in **Fig 92** provides insights into the quality and coherence of clusters formed by the K-means algorithm for different numbers of clusters (K). Higher silhouette scores indicate better-defined clusters, where data points are closer to members of their own cluster than to those in other clusters. In this plot, we observe that the silhouette scores peak at K=2, suggesting that dividing the data into two clusters results in the most well-defined and separated clusters. As the number of clusters increases beyond two, the silhouette scores decrease, indicating reduced cluster coherence and suggesting that additional clusters may not capture meaningful patterns in the data. Therefore, based on the silhouette score analysis, K=2 appears to be the optimal number of clusters for this dataset.

- **K-Means Clustering with K=2**

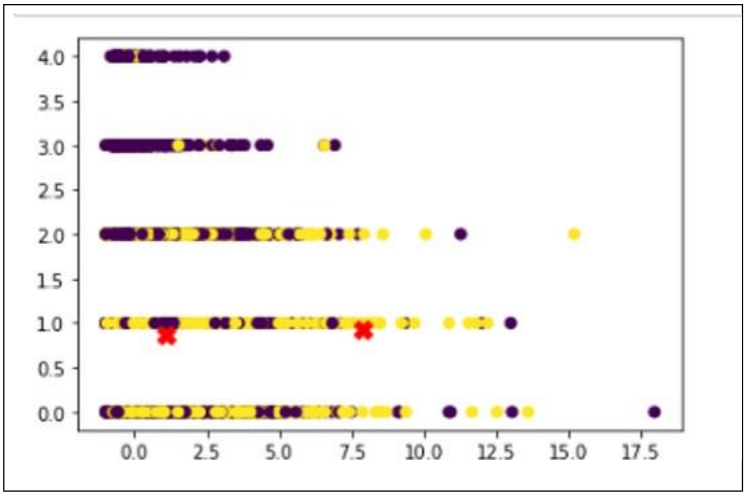


Fig 93: Scatter Plot of Data Points and Centroids

The K-means clustering algorithm was applied with K=2 clusters in **Fig 93**, this determined as the optimal choice based on the Within-Cluster Sum of Squares (WSS) criterion. The resulting scatter plot illustrates the clustering of data points in two distinct groups, represented by different colors. Additionally, red 'X' markers indicate the centroids of each cluster. This visualization highlights the separation of data points into two clusters based on their features, providing insights into the underlying structure of the dataset.

2. DBSCAN

- **DBSCAN Clustering Results**

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm that groups together closely packed points while marking outliers as noise.

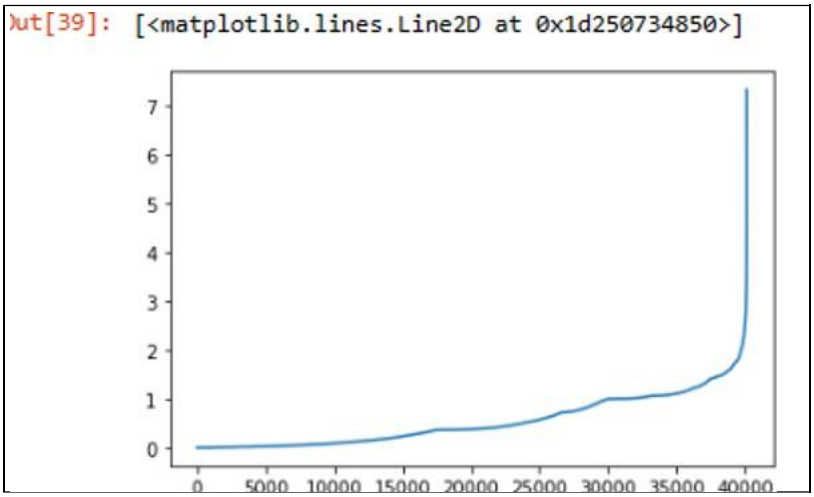


Fig 94: Nearest Neighbor Distances Plot for DBSCAN Clustering

The plot in **Fig 94**visualizes the nearest neighbor distances calculated for each data point, a critical aspect in DBSCAN clustering. It shows the indices of data points sorted by their distances to their nearest neighbors on the x-axis and the corresponding distances on the y-axis. A noticeable increase or "knee" in the plot suggests a threshold distance, guiding the selection of the DBSCAN parameter **eps**. This parameter determines the maximum distance between two samples for them to be considered in the same neighborhood. Identifying this knee point helps in setting an optimal **eps**, crucial for accurate clustering by DBSCAN, distinguishing clusters from noise or outliers within the dataset

- **The Visualization of DBSCAN Clustering Results**

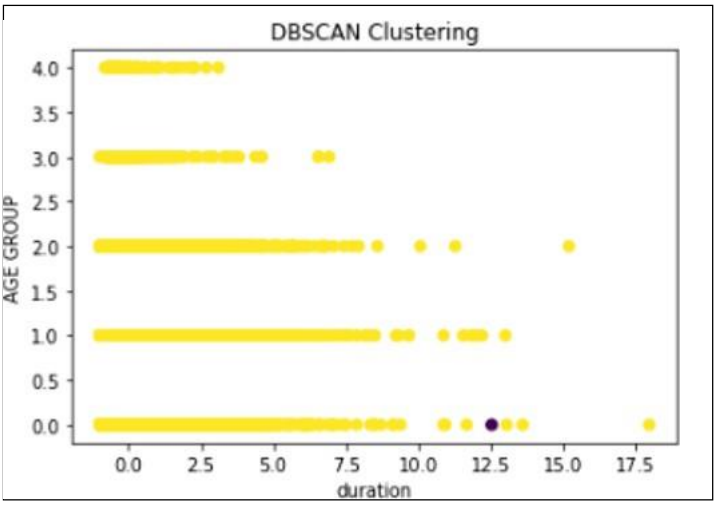


Fig 95: DBSCAN Clustering: Duration vs. Age Group

The scatter plot in **Fig 95** above represents the outcome of DBSCAN clustering, where each point is colored according to its assigned cluster label. This clustering algorithm groups data points based on their density, distinguishing clusters from noise points. The x-axis corresponds to the 'duration' feature, while the y-axis represents the 'AGE GROUP'. The varying colors indicate different clusters identified by DBSCAN, with points of the same color belonging to the same cluster. Outliers or noise points are typically labeled as -1. The plot provides insights into the structure of the data and the effectiveness of the clustering algorithm in identifying meaningful clusters.

- **The Visualization of Estimated Clusters**

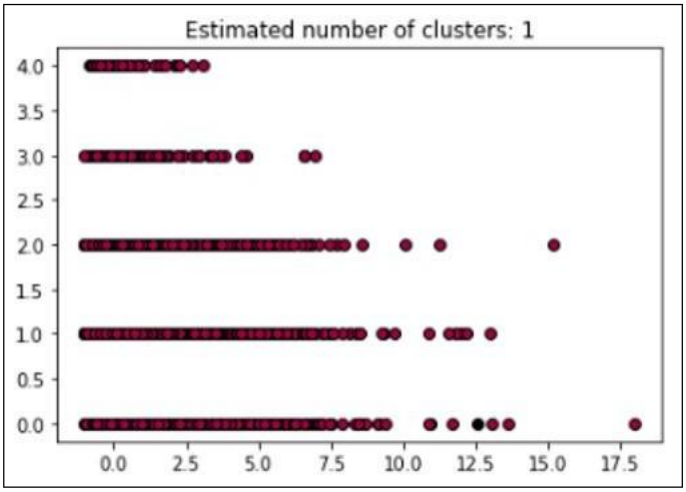


Fig 96: Estimated Clusters: Duration vs. Age Group

The plot in **Fig 96** depicts the estimated clusters generated by the DBSCAN algorithm. Each cluster is represented by a distinct color, while noise points are depicted in black. The x-axis represents the duration, and the y-axis represents the age group. The clusters show distinct groupings of data points based on these two features. T

3. PAMK

The Silhouette plot for PAMK clustering visually assesses the cohesion and separation of data points within clusters, aiding in the evaluation of clustering effectiveness based on silhouette coefficients.

- **Silhouette Plot for PAMK Clustering**

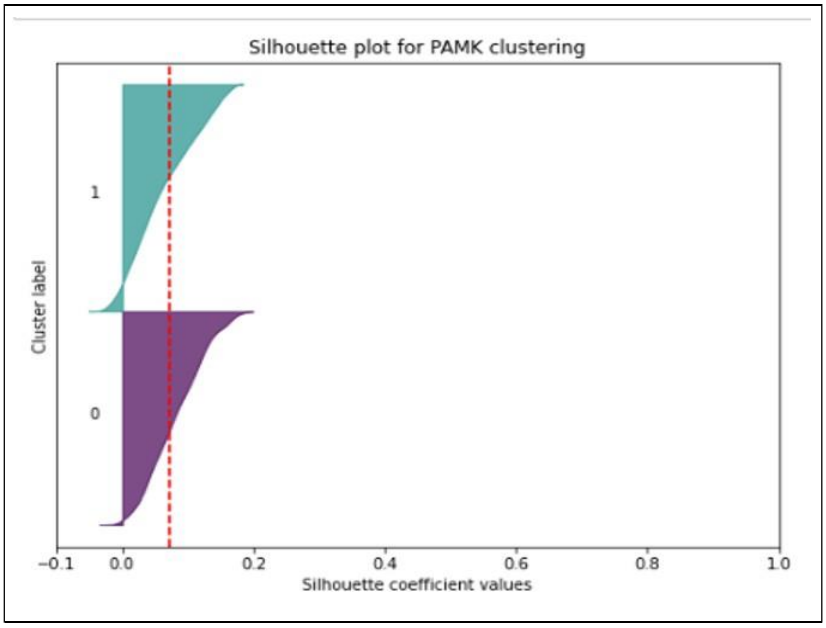


Fig 97: Silhouette Analysis for KMedoids Clustering

The Silhouette plot in **Fig 97** for PAMK clustering displays the silhouette coefficients for each data point within the clustered dataset. Each cluster is represented by a distinct color, and the position of each silhouette coefficient along the y-axis corresponds to its assigned cluster label. Silhouette coefficients measure the cohesion and separation of data points within their respective clusters. A silhouette coefficient close to +1 indicates that the data point is well-matched to its cluster and poorly matched to neighboring clusters, while a value near 0 suggests overlapping clusters, and negative values indicate potential misclassifications. The plot's overall shape and distribution provide insights into the quality and coherence of the clustering results, helping to evaluate the effectiveness of the clustering algorithm.

- **Silhouette Plot Interpretation**

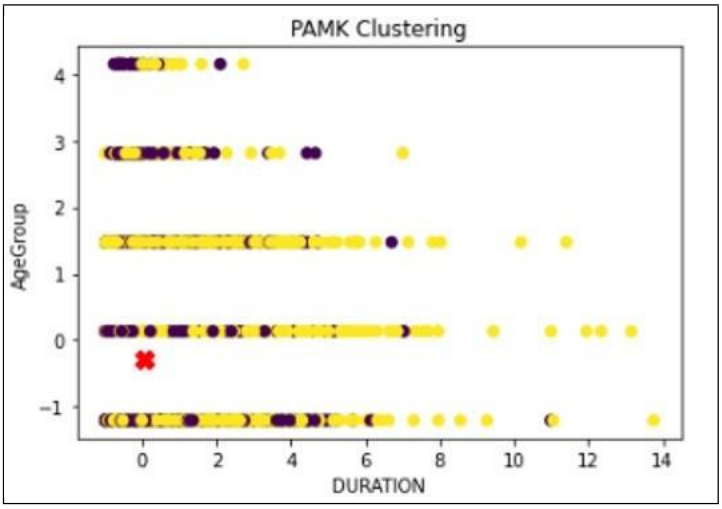


Fig 99: Silhouette Plot for PAMK Clustering

The scatter plot visualizing the results of PAMK (Partitioning Around Medoids) clustering in **Fig 99** showcases how the data points are grouped into clusters based on their duration and age group features. The points are colored according to their cluster assignments, with medoids marked as red Xs representing the central points of each cluster. The separation and cohesion of clusters indicate how well the PAMK algorithm has partitioned the data. A well-separated clustering with minimal overlap between clusters signifies that the algorithm effectively captured distinct patterns in the data.

Conclusion:

Both R and Python

This project successfully implemented various machine learning models to predict the success of marketing campaigns and conducted clustering analysis to identify distinct customer segments. The use of decision trees provided a strong starting point, but significant improvements were observed with ensemble methods like Random Forest and Gradient Boosting, particularly after addressing class imbalance issues. Naive Bayes, while simpler, offered reasonable performance, though it fell short compared to the more robust ensemble techniques. Clustering analyses, including methods like K-Means, PAMK, and DBSCAN, revealed meaningful customer segments, offering valuable insights for targeted marketing strategies. DBSCAN, in particular, was effective in identifying clusters with varying densities, highlighting its utility in complex datasets.

Overall, the integration of predictive modeling and clustering analyses has provided a comprehensive framework for understanding customer behavior and optimizing marketing efforts in the banking sector. The findings underscore the importance of feature engineering, model tuning, and the application of advanced algorithms to enhance prediction accuracy and customer segmentation. Moving forward, further exploration of additional machine learning techniques, hyperparameter optimization, and more sophisticated clustering algorithms will be essential. The continuous refinement of predictive models and validation of marketing strategies through A/B testing will ensure that the bank can adapt to evolving market conditions and customer preferences, ultimately leading to more effective and personalized marketing campaigns.

References:

- (LILA), H. R. (2024, March). *Bank Marketing Model Performance Analysis*. Retrieved from Kaggle.com: https://www.kaggle.com/code/hozanareis/bank-marketing-model-performance-analysis
- *Bank Marketing*. (2012, February 13). Retrieved from UCI: https://archive.ics.uci.edu/dataset/222/bank+marketing
- *Imbalanced Classification Problem in R*. (2016, March). Retrieved from Analytics Vihya: https://www.analyticsvidhya.com/blog/2016/03/practical-guide-deal-imbalanced-classification-problems/
- Stockdale, D. (2021, August 13). *How to handle imbalanced datasets*. Retrieved from Rpub: https://rpubs.com/DeclanStockdale/799284
- YAMAHATA, H. (2018). *Bank Marketing*. Retrieved from Kaggle.com: https://www.kaggle.com/datasets/henriqueyamahata/bank-marketing/code
- YAMAHATA, H. (2018). *Bank Marketing - Imbalanced Dataset 94%*. Retrieved from kaggle.com: https://www.kaggle.com/code/henriqueyamahata/bank-marketing-imbalanced-dataset-94
- https://matplotlib.org/
- https://elutins.medium.com/dbscan-what-is-it-when-to-use-it-how-to-use-it-8bd506293818
- https://towardsai.net/p/data-science/scaling-vs-normalizing-data-5c3514887a84
- https://scikit-learn-extra.readthedocs.io/en/stable/modules/cluster.html
- https://tslearn.readthedocs.io/en/stable/gen_modules/clustering/tslearn.clustering.silhouette_score.html
- https://medium.com/@matthew.dicicco38/elbow-method-explained-e8736cfcc5d1

Appendix

R-PROGRAM:

```
# Load libraries
library(magrittr)
library(pROC)
library(randomForest)
library(mclust)
library(caTools)
library(caret)
library(ipred)
library(rpart)
library(Metrics)
library(dplyr)
library(e1071)
library(corrplot)
library(ggplot2)
library(rpart.plot)

# Load the bank additional dataset
bank_df <- read.csv("bank-additional-full.csv")

# Exploration Data Analysis
# Checking the structure and first few rows
str(bank_df)  # Structure of the dataset
head(bank_df)
colnames(bank_df)

# Checking for Missing values
sum(is.na(bank_df))

# Plot the distribution for all the numeric variables
Num_variables <- c('age','duration','campaign','pdays','previous','emp.var.rate',
                   'cons.price.idx','euribor3m','nr.employed','cons.conf.idx')

# Set the number of plots per row and column
num_plots_per_row <- 3
num_rows <- ceiling(length(Num_variables) / num_plots_per_row)

# Create histograms
par(mfrow = c(num_rows, num_plots_per_row), mar = c(2, 2, 2, 1))  # Adjust margins

for (variable in Num_variables) {
  hist(bank_df[[variable]], main = variable, xlab = "", col = "skyblue", border = "white")
}
```

```
# Data Preprocessing
# Categorical variables
cat_variables <- c('job', 'marital', 'education', 'default', 'housing', 'loan',
                   'contact', 'month', 'day_of_week', 'poutcome', 'y')

# Set the number of plots per row and column
num_plots_per_row <- 3
num_rows <- ceiling(length(cat_variables) / num_plots_per_row)

# Create bar charts
par(mfrow = c(num_rows, num_plots_per_row), mar = c(4, 4, 2, 1))  # Adjust margins

for (variable in cat_variables) {
  counts <- table(bank_df[[variable]])
  barplot(counts, main = variable, xlab = "", ylab = "Frequency", col = "skyblue", border = "white")
}

# Creating bivariate plots for the categorical variables
# Create a function to generate count plots for each categorical variable
plot_bivariate <- function(column) {
  ggplot(bank_df, aes(x = .data[[column]], fill = as.factor(y))) +
    geom_bar(position = 'dodge') +
    labs(title = paste(column, "vs. y"),
         x = NULL, y = NULL) +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
    scale_fill_manual(values = c("0" = "skyblue", "1" = "salmon"),
                      labels = c("0" = "No", "1" = "Yes"),
                      name = "y")
}

plot_bivariate

# Create plots
plots <- lapply(cat_variables, plot_bivariate)

# Open a new device for each plot
for (i in seq_along(plots)) {
  png(filename = paste0("plot_", i, ".png"))
  print(plots[[i]])
  dev.off()
}
plots

# Select the variables
var <- c('age','duration','campaign','pdays','previous','emp.var.rate',
         'cons.price.idx','euribor3m','nr.employed','cons.conf.idx')

# Subset the dataset to include only the selected variables
bank_subset <- bank_df[, var]
```

```
# Compute the correlation matrix
correlation_matrix <- cor(bank_subset)

# Create heatmap
corrplot(correlation_matrix, method = "color", col = colorRampPalette(c("blue", "white",
"red"))(100),
         type = "upper", order = "hclust", addCoef.col = "black", tl.col = "black",
         tl.srt = 45, tl.cex = 0.7, number.cex = 0.7)

# Solving Multicollinearity
# Find variables with correlation greater than 0.95
high_correlation <- findCorrelation(correlation_matrix, cutoff = 0.95)

# Remove numeric variables with high correlation above 0.95
variables_to_remove <- c('nr.employed', 'emp.var.rate')

# Remove the columns from the data frame
bank_df <- bank_df %>%
  select(-one_of(variables_to_remove))


# Checking for correlation after removing variables with high correlation
# Subset the data set to include only the selected variables
# Numerical Columns
num_var2 <- c('age','duration','campaign', 'pdays', 'previous',
              'cons.price.idx', 'euribor3m','cons.conf.idx')

# Subset the dataset to include only the selected variables
bank_sub2 <- bank_df[, num_var2]
# Compute the correlation matrix
correlation_matrix2 <- cor(bank_sub2)

# Create heatmap
corrplot(correlation_matrix2, method = "color", col = colorRampPalette(c("blue", "white",
"red"))(100),
         type = "upper", order = "hclust", addCoef.col = "black", tl.col = "black",
         tl.srt = 45, tl.cex = 0.7, number.cex = 0.7)

# Removing unknown values for categorical features

# Remove category "Unknown" from Marital status
# Filter out rows where 'marital' is not 'unknown'
marital_fil <- bank_df[bank_df$marital != 'unknown', ]

# Compute value counts of 'marital' column
marital_counts <- table(marital_fil$marital)

# Display the value counts
print(marital_counts)
```

```
# Remove category "Unknown" from 'housing'
# Filter out rows where 'housing' is not 'unknown'
housing_fil <- bank_df[bank_df$housing != 'unknown', ]

# Compute value counts of 'marital' column
housing_counts <- table(housing_fil$housing)

# Display the value counts
print(housing_counts)

# Create a new category called "basic.education" by replacing the values 'basic.4y', 'basic.6y', and 'basic.9y'
# Replace values in 'education' column
bank_df$education <- ifelse(bank_df$education %in% c('basic.4y', 'basic.6y', 'basic.9y'),
                            'basic.education', bank_df$education)

# Display the value counts of 'education' column
table(bank_df$education)

# Remove category "Unknown" from 'housing'
# Filter out rows where 'housing' is not 'unknown'
education_fil <- bank_df[bank_df$education != 'unknown', ]

# Compute value counts of 'marital' column
education_counts <- table(education_fil$education)

# Display the value counts
print(education_counts)

# Plot histogram of age distribution
ggplot(bank_df, aes(x = age)) +
  geom_histogram(binwidth = 3, fill = "skyblue", color = "black") +
  labs(title = "Age Distribution", x = "Age", y = "Frequency") +
  theme_minimal()


# Converting all categorical variables to a factor
bank_df$loan <- as.factor(bank_df$loan)
bank_df$contact <- as.factor(bank_df$contact)
bank_df$month <- as.factor(bank_df$month)
bank_df$job <- as.factor(bank_df$job)
bank_df$marital <- as.factor(bank_df$marital)
bank_df$education <- as.factor(bank_df$education)
bank_df$default <- as.factor(bank_df$default)
bank_df$housing <- as.factor(bank_df$housing)
bank_df$day_of_week <- as.factor(bank_df$day_of_week)
bank_df$poutcome <- as.factor(bank_df$poutcome)

# Model Implementation
```

```
# Checking for imbalance in the data

# Create a dataframe for the target variable counts
target_counts <- data.frame(table(bank_df$y))

# Plot the distribution of the target variable 'y'
ggplot(target_counts, aes(x = Var1, y = Freq)) +
  geom_bar(stat = "identity", fill = "skyblue", color = "black") +
  labs(title = "Distribution of Target Variable 'y'",
       x = "y", y = "Frequency") +
  theme_minimal()

###############MODEL IMPLEMENTATION####################

####Splitting the data into test and train

# Set the seed for reproducibility
set.seed(42)
# Split the data into training and testing sets (70% train, 30% test)
sample_split <- sample.split(Y = bank_df$y, SplitRatio = 0.70)
train_bx <- subset(x = bank_df, sample_split == TRUE)
test_bx <- subset(x = bank_df, sample_split == FALSE)

# Checking the imbalance state of the target variable
# Check table
table(train_bx$y)

# Check classes distribution
prop.table(table(train_bx$y))


########################Decision Tree#################
# Build the decision tree model based on the imbalanced state of the dataset
model_DT <- rpart(y ~ ., data = train_bx, method = "class")


# Open a new device for the decision tree plot
png(filename = "decision_tree.png")
# Plot the decision tree
rpart.plot(model_DT, main = "Decision Tree on Banking Information")
# Close the device
dev.off()

# Display the decision tree plot
rpart.plot(model_DT, main = "Decision Tree on Banking Information")

# Check for the feature selection
importanceBX <- varImp(model_DT)
importanceBX %>% arrange(desc(Overall))
```

```
# Predict the model based on the test set
test_bf <- test_bx  # Assuming test_bf is meant to be test_bx
preds_DT <- predict(model_DT, newdata = test_bf, type = "class")
head(preds_DT, 10)

# Convert test_bx$y and preds_DT to factors with the same levels
test_bx$y <- factor(test_bx$y, levels = levels(preds_DT))
# Ensure factor levels match between training and testing data
factor_vars <- c('loan', 'contact', 'month', 'job', 'marital', 'education', 'default', 'housing', 'day_of_week', 'poutcome')

for (var in factor_vars) {
  test_bx[[var]] <- factor(test_bx[[var]], levels = levels(train_bx[[var]]))
}

# Create a confusion matrix
conf_matrix <- confusionMatrix(preds_DT, test_bx$y)
print(conf_matrix)

########CORRECTING DATA IMBALANCE ##############
# using the ROSE function and the oversampling and under sampling techniques to balanced the train set of the target variable 'y'
library(ROSE)
# Over-sample the minority class
df_oversampled <- ovun.sample(y ~ ., data = train_bx, method = "over", N = 2 * table(train_bx$y)[[1]], seed = 42)$data

# Check the new class distribution
table(df_oversampled$y)

# Under sample the majority class
class(bank_df$y)
# Convert 'y' to a factor variable
train_bx$y <- as.factor(train_bx$y)

#Perform undersampling on the train set
df_undersampled <- downSample(x = train_bx[, -which(names(train_bx) == "y")],
                              y = train_bx$y,
                              list = FALSE)$x

# Combine the undersampled data with the original minority class
df_balanced <- rbind(df_undersampled, train_bx[train_bx$y == "yes", ])

# Check the new class distribution
table(df_balanced$y)


#balance both samples of the train set
data_balanced_both <- ovun.sample(y ~ ., data = train_bx, method = "both", p=0.5, seed = 1)$data
table(data_balanced_both$y)
```

```
# Check classes distribution again
prop.table(table(data_balanced_both$y))

#using the function Rose to correct any inconsistency from the oversampling and undersampling data
data.rose <- ROSE(y ~ ., data =train_bx, seed = 1)$data
table(data.rose$y)

#Now we build the Decision Tree model based on the balanced data using the ROSE technique
# Train decision tree on the ROSE function set
tree_model <- rpart(y ~ ., data = data.rose, method = "class")

# Predict on the combined data
pred_dt2 <- predict(tree_model, newdata = test_bx , type = "class")

# Evaluate balance
table(pred_dt2, test_bx$y)

#plot for decision tree
rpart.plot(tree_model, main = "Decision Tree on Banking Information")

# check for the feature selection
importanceBF <- varImp(tree_model)
importanceBF %>% arrange(desc(Overall))


# Create a confusion matrix to evaluate the performance of the model again
conf_matrix_balanced <- confusionMatrix(pred_dt2, test_bx$y)
print(conf_matrix_balanced)

#check the ROC curve on the ROSE technique set
roc.curve(test_bx$y, pred_dt2)

# Train decision tree on combined technique to check for best accuracy
tree_balance <- rpart(y ~ ., data = data_balanced_both, method = "class")

# Predict on the combined data
pred_bal <- predict(tree_balance, newdata = test_bx , type = "class")

# Create a confusion matrix to evaluate the performance of the model again
conf_matrix_balanced2 <- confusionMatrix(pred_bal, test_bx$y)
print(conf_matrix_balanced2)

#check the ROC curve on the combined technique
ROC2<- roc.curve(test_bx$y, pred_bal)
ROC2
# Plot confusion matrix
ggplot(data.frame(conf_matrix_balanced$table), aes(x = Reference, y = Prediction, fill = Freq)) +
  geom_tile(color = "white") +
```

```
  geom_text(aes(label = Freq), vjust = 1) +
  scale_fill_gradient(low = "lightblue", high = "darkblue") +
  labs(title = "Confusion Matrix - DECISION TREE",
       x = "Predicted Labels",
       y = "Actual Labels") +
  theme_minimal()


# Extract accuracy from the confusion matrix
accuracy_dt <- conf_matrix_balanced$overall['Accuracy']

# Print accuracy
print(paste("Accuracy:", accuracy_dt))


####################NaiveBAyes#################################################
#

# Install and load the 'naivebayes' package
# install.packages("naivebayes")
library(naivebayes)

# Train Naive Bayes model on the balanced data 'data.rose' with Laplace smoothing
nb_model <- naive_bayes(y ~ ., data = data.rose, laplace = 1)

# Display summary of the Naive Bayes model
summary(nb_model)

# Load the required package
library(caret)

# Predict on the balanced data
pred_nb <- predict(nb_model, newdata = test_bx)

# Report the confusion matrix on the 'nb_model'
conf_matrix_nb <- confusionMatrix(pred_nb, test_bx$y)
print(conf_matrix_nb)




# Evaluate balance
table(pred_nb, test_bx$y)

# Plot confusion matrix
ggplot(data.frame(conf_matrix_nb$table), aes(x = Reference, y = Prediction, fill = Freq)) +
  geom_tile(color = "white") +
  geom_text(aes(label = Freq), vjust = 1) +
  scale_fill_gradient(low = "lightblue", high = "darkblue") +
  labs(title = "Confusion Matrix - NAVIE BAYES",
```

```
      x = "Predicted Labels",
      y = "Actual Labels") +
  theme_minimal()

# Extract accuracy from the confusion matrix
accuracy_nb <- conf_matrix_nb$overall['Accuracy']

# Print accuracy
print(paste("Accuracy:", accuracy_nb))



############### SVM ##############################

# Install and load the required package
# install.packages("e1071")
library(e1071)

# Train SVM model on combined data
svm_model <- svm(y ~ ., data = data.rose, kernel = "radial")

# Display summary of the SVM model
summary(svm_model)


# Install and load the required package
# install.packages("caret")
library(caret)

# Predict on the combined data
pred_sv <- predict(svm_model, newdata = test_bx)

# Reporting the confusion matrix on the 'svm_model'
conf_matrix_sv <- confusionMatrix(pred_sv, test_bx$y)
print(conf_matrix_sv)


# Evaluate balance
table(pred_sv, test_bx$y)

# Plot confusion matrix
ggplot(data.frame(conf_matrix_sv$table), aes(x = Reference, y = Prediction, fill = Freq)) +
  geom_tile(color = "white") +
  geom_text(aes(label = Freq), vjust = 1) +
  scale_fill_gradient(low = "lightblue", high = "darkblue") +
  labs(title = "Confusion Matrix - SVM",
      x = "Predicted Labels",
      y = "Actual Labels") +
  theme_minimal()
```

```
# Extract accuracy from the confusion matrix
accuracy_sv <- conf_matrix_sv$overall['Accuracy']

# Print accuracy
print(paste("Accuracy:", accuracy_sv))


#####################################KNN#####################################
####

# Load the required packages
if (!require(class)) {
install.packages("class")
}
if (!require(caret)) {
  install.packages("caret")
}
if (!require(ggplot2)) {
  install.packages("ggplot2")
}

# Load the packages
library(class)
library(caret)
library(ggplot2)

# Train KNN classifier
# Subset the dataset to include only the selected variables
bank_sub <- bank_df[, c(num_var2, 'y')]

# Set seed for reproducibility
set.seed(42)

# Use sample.split function from caTools package
split <- caTools::sample.split(Y = bank_sub$y, SplitRatio = 0.7)

# Create training and testing datasets
train_dx <- bank_sub[split, ]
test_dx <- bank_sub[!split, ]

# Separate predictors (features) and target variable for training and testing datasets
X_train <- train_dx[, num_var2]
y_train <- train_dx$y
X_test <- test_dx[, num_var2]
y_test <- test_dx$y

# Step 3: Running KNN based on different amounts of K
# Creating vectors to store results
k_values <- 1:20 # Try different values of k
misclassification_error <- numeric(length(k_values))
```

```
accuracy <- numeric(length(k_values))

# Running KNN for different k values
for (i in 1:length(k_values)) {
  classifier_knn <- knn(train = X_train[, -c(1)], test = X_test[, -c(1)], cl = y_train, k = k_values[i])
  # Calculating misclassification error
  misclassError <- mean(classifier_knn != y_test)
  misclassification_error[i] <- misclassError
  # Calculating accuracy
  acc <- 1 - misclassError
  accuracy[i] <- acc
}

# Print misclassification error and accuracy for each run
knn_res <- data.frame(K_Value = k_values, Misclassification_Error = misclassification_error,
Accuracy = accuracy)
print(knn_res)

# Report the confusion matrix
conf_matrix_knn <- confusionMatrix(factor(classifier_knn), factor(y_test))
print(conf_matrix_knn)

# Plot confusion matrix
ggplot(data.frame(conf_matrix_knn$table), aes(x = Reference, y = Prediction, fill = Freq)) +
  geom_tile(color = "white") +
  geom_text(aes(label = Freq), vjust = 1) +
  scale_fill_gradient(low = "lightblue", high = "darkblue") +
  labs(title = "Confusion Matrix - KNN",
       x = "Predicted Labels",
       y = "Actual Labels") +
  theme_minimal()


####################################ENSEMBLE
METHODS##############################


#################################### Random
Forest##############################

# install.packages("randomForest")
library(randomForest)

# Train Random Forest model on combined data
rf_model <- randomForest(y ~ ., data = data.rose, ntree = 500, ntry = 6, importance = TRUE,
                         na.action = randomForest::na.roughfix, replace = FALSE)
summary(rf_model)

# Predict on the test set
pred_rf <- predict(rf_model, newdata = test_bx)
```

```
# Report the confusion matrix on the 'svm_model'
conf_matrix_rf <- confusionMatrix(pred_rf, test_bx$y)
print(conf_matrix_rf)

# Evaluate balance
table(pred_rf, test_bx$y)

# Plot confusion matrix
ggplot(data.frame(conf_matrix_rf$table), aes(x = Reference, y = Prediction, fill = Freq)) +
  geom_tile(color = "white") +
  geom_text(aes(label = Freq), vjust = 1) +
  scale_fill_gradient(low = "lightblue", high = "darkblue") +
  labs(title = "Confusion Matrix - RANDOM FOREST",
       x = "Predicted Labels",
       y = "Actual Labels") +
  theme_minimal()

# Extract accuracy from the confusion matrix
accuracy_rf <- conf_matrix_rf$overall['Accuracy']

# Print accuracy
print(paste("Accuracy:", accuracy_rf))

# Plot Variable Importance Calculation using Random Forest
varImpPlot(rf_model, col = 3, main = "Variable Importance Plot for Random Forest")


#####################################                                    BAGGING
#####################################

# Train Bagging model on combined data
bg_model <- ipred::bagging(y ~ ., data = data.rose, nbagg = 100)
summary(bg_model)

# Predict on the combined data
pred_bg <- predict(bg_model, newdata = test_bx)

# Reporting the confusion matrix on the 'svm_model'
conf_matrix_bg <- caret::confusionMatrix(pred_bg, test_bx$y)
print(conf_matrix_bg)

# Plot confusion matrix
ggplot2::ggplot(data.frame(conf_matrix_bg$table), aes(x = Reference, y = Prediction, fill = Freq))
+
  geom_tile(color = "white") +
  geom_text(aes(label = Freq), vjust = 1) +
  scale_fill_gradient(low = "lightblue", high = "darkblue") +
  labs(title = "Confusion Matrix - BAGGING",
       x = "Predicted Labels",
```

```
      y = "Actual Labels") +
  theme_minimal()

# Extract accuracy from the confusion matrix
accuracy_bg <- conf_matrix_bg$overall['Accuracy']

# Print accuracy
print(paste("Accuracy:", accuracy_bg))



#######################################
XGBOOSTING#############################

#Extract features and convert to a numeric matrix
features_matrix <- as.matrix(X_train[, !names(X_train) == "y"])

# Extract labels
labels <- y_train

# Check data structure
str(features_matrix)

#Convert labels into binary or numeric variables.
labels <- ifelse(y_train == "yes", 1, 0)
str(labels)

# Train the xgboost model
boost1 <- xgboost(data = features_matrix,
                  label = labels,
                  max.depth = 2,
                  eta = 1,
                  nthread = 2,
                  nrounds = 2,
                  objective = "binary:logistic")


boost2 <- xgboost(data = features_matrix,
                  label = labels,
                  max.depth = 2,
                  eta = 1,
                  nthread = 2,
                  nrounds = 10,
                  objective = "binary:logistic")

Imp1 <-xgb.importance(feature_names = colnames(X_train), model = boost2)
xgb.plot.importance(importance_matrix = Imp1[1:10], main = 'XGBOOST feature Importance')
print(Imp1)

#Reporting the confusion matrix
```

```
test_matrix <- as.matrix(X_test[, !names(X_test) == "y"])
test_labels <- ifelse(y_test == "yes", 1, 0)

# Make predictions on the test set
pred_prob <- predict(boost2, test_matrix)

# Convert probabilities to binary labels
pred_labels <- ifelse(pred_prob > 0.5, 1, 0)

# Create confusion matrix
conf_matrix_boost <- confusionMatrix(as.factor(pred_labels), as.factor(test_labels))

# Print the confusion matrix
print(conf_matrix_boost)

#Plot the confusion matrix
# Plot confusion matrix
ggplot(data.frame(conf_matrix_boost$table), aes(x = Reference, y = Prediction, fill = Freq)) +
  geom_tile(color = "white") +
  geom_text(aes(label = Freq), vjust = 1) +
  scale_fill_gradient(low = "lightblue", high = "darkblue") +
  labs(title = "Confusion Matrix -XGBOOSTING",
       x = "Predicted Labels",
       y = "Actual Labels") +
  theme_minimal()

# Extract accuracy from the confusion matrix
accuracy_boost <- conf_matrix_boost$overall['Accuracy']

# Print accuracy
print(paste("Accuracy:", accuracy_boost))




##########################Random Forest on the Numeric variables################

# Train Random Forest model on numeric train set
rf_model_num <- randomForest(x = X_train[, -c(1)], y = y_train, ntree = 100)

# Predict on the testing data
y_pred_rf <- predict(rf_model_num, newdata = X_test[, -c(1)])

# Compute accuracy
accuracy_rf_num <- sum(y_pred_rf == y_test) / length(y_test)
print(paste("Random Forest Accuracy:", accuracy_rf_num))

# Compute confusion matrix
conf_matrix_rf2 <- confusionMatrix(y_pred_rf, y_test)
```

```
# Print confusion matrix
print(conf_matrix_rf2)
# Plot confusion matrix
ggplot(data.frame(conf_matrix_rf2$table), aes(x = Reference, y = Prediction, fill = Freq)) +
  geom_tile(color = "white") +
  geom_text(aes(label = Freq), vjust = 1) +
  scale_fill_gradient(low = "lightblue", high = "darkblue") +
  labs(title = "Confusion Matrix -Random Forest for Numeric features",
       x = "Predicted Labels",
       y = "Actual Labels") +
  theme_minimal()



##############################B)                                    CLUSTERING
#####################################



###################################                                    KMEANS
#######################################

# Create a copy of the dataset excluding the target variable 'y'
bank_num <- bank_sub
bank_num$y <- NULL

# Check the structure of the dataset
str(bank_num)

# Perform K-means clustering
k_means_rx <- kmeans(bank_num, centers = 3)  # Specify the number of clusters (centers = 3)

# View the clustering results
k_means_rx
# Compare clusters with original labels
cluster_labels <- k_means_rx$cluster
table(bank_sub$y,k_means_rx$cluster)

##Finding the best number of cluster using:
#Elbow Method
wcss <- numeric(10)  # Initialize vector to store within-cluster sum of squares
for (i in 1:10) {
  kmeans_model <- kmeans(bank_num, centers = i)
  wcss[i] <- kmeans_model$tot.withinss
}
# Plot the Elbow Method
plot(1:10, wcss, type = "b", pch = 19, frame = FALSE, xlab = "Number of Clusters", ylab =
"WCSS",
     main = "Elbow Method")
```

```
# Silhouette Method

# Define a range of K values to evaluate
k_values <- 2:10

# Initialize a vector to store silhouette widths for each value of K
silhouette_widths <- numeric(length(k_values))

# Perform K-means clustering for each value of K and evaluate the results
for (i in seq_along(k_values)) {
  k <- k_values[i]
  kmeans_result <- kmeans(bank_num, centers = k)
  silhouette_width <- silhouette(kmeans_result$cluster, dist(bank_num))
  silhouette_widths[i] <- mean(silhouette_width[, "sil_width"])
}

# Plot the silhouette widths for each value of K
plot(k_values, silhouette_widths, type = "b", xlab = "Number of clusters (K)", ylab = "Mean silhouette width")

# Find the optimal number of clusters based on silhouette width
optimal_k <- k_values[which.max(silhouette_widths)]
cat("Optimal number of clusters based on silhouette width:", optimal_k, "\n")

#Perform K-means clustering with the optimal number of clusters
optimal_kmeans_result <- kmeans(bank_num, centers = optimal_k)
optimal_kmeans_result


####################################Using                                NbClust
################################


# checking for best number of cluster by computing NbClust
set.seed(123)  # For reproducibility
bank_num_sample <- bank_num[sample(nrow(bank_num), size = 10000), ]  # Adjust size as needed
nbclust_result <- NbClust(bank_num_sample, min.nc = 2, max.nc = 10, method = "kmeans")

# Compute NbClust for determining the best number of clusters
nbclust_result <- NbClust(bank_num, min.nc = 2, max.nc = 10, method = "kmeans")

best_clust <- table(nbclust_result$Best.n[1,])
best_clust


########################################Using
PAMK############################
```

```
pamk_bf <- pamk(bank_num_sample, 2:10)
pamk_bf

pam_res <- pamk_bf$pamobject
pamk_bf$pamobject$medoids

layout(matrix(c(1,2),1,2))
plot(pamk_bf$pamobject)
layout(matrix(1))

# Plotting the result based on Silhouette parameter
sil_width <- silhouette(pam_res)
plot(sil_width)



############## DBSCAN #################

# Implementations with different parameter settings
eps_values <- 0.5
min_points_values <- c(5, 10)

best_silhouette <- -1
best_eps <- 0
best_min_points <- 0
best_cluster <- NULL

for (eps in eps_values) {
  for (min_points in min_points_values) {
    # DBSCAN clustering
    dbscan_result <- dbscan(bank_num_sample, eps = eps, MinPts = min_points)

    # Compute silhouette width
    silhouette_width <- silhouette(dbscan_result$cluster, dist(bank_num_sample))

    # Check if silhouette width is valid
    if (!is.null(silhouette_width) && !anyNA(silhouette_width)) {
      # Check if silhouette width is better than the current best
      if (mean(silhouette_width) > best_silhouette) {
        best_silhouette <- mean(silhouette_width)
        best_eps <- eps
        best_min_points <- min_points
        best_cluster <- dbscan_result$cluster
      }
    }
  }
}
# Perform DBSCAN clustering
# Setting the best value for eps and min_points
best_dbscan_result <- dbscan(bank_num_sample, eps = best_eps, minPts = best_min_points)
```

```
# Plot the DBSCAN result
plot(bank_num_sample, col = best_dbscan_result$cluster + 1,
    main = paste("DBSCAN Clustering (eps =", best_eps, ", MinPts =", best_min_points, ")"),
    pch = 20)
legend("topright", legend = unique(best_dbscan_result$cluster), col =
unique(best_dbscan_result$cluster + 1), pch = 20)
bank_num_sample



kNNdistplot(bank_num_sample, k=4)
abline(h=0.5, col ="red")


eps<-function(mydata){
  dist <- dbscan ::kNNdist(mydata,ncol(mydata)+1)
  dist <- dist[order(dist)]
  dist <- dist/max(dist)
  ddist <- diff(dist)/(1/length(dist))
  EPS <- dist[length(ddist)-length(ddist[ddist>1])]
  print(EPS)
}

eps(bank_num_sample)

abline(h=0.01830199, col = 2, lty = 5)
bank_dbs<- dbscan(bank_num_sample, eps = 0.01830199, minPts = ncol(bank_num_sample)+1)
bank_dbs

hullplot(bank_num_sample, bank_dbs)
plot(bank_num_sample, col = bank_dbs$cluster)

noise = which(bank_dbs$cluster == 0)
noise
```

PYTHON

```
# Libraries used
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_score,
f1_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier
import xgboost as xgb
from sklearn.metrics import roc_curve, auc

# Load dataset
df = pd.read_csv("C:/Users/Rakesh/Desktop/Varada_ALY6040/bank_marketing.csv",
delimiter=';')

# Univariable check
numerical_columns = ['age', 'duration', 'campaign', 'pdays', 'previous',
                     'emp.var.rate', 'cons.price.idx', 'cons.conf.idx',
                     'euribor3m', 'nr.employed']

plt.figure(figsize=(12, 8))
for i, column in enumerate(numerical_columns, 1):
    plt.subplot(3, 4, i)
    plt.hist(df[column], bins=20, color='skyblue', edgecolor='black', linewidth=1.5)
    plt.title(column)
    plt.xlabel('Value')
    plt.ylabel('Frequency')

plt.tight_layout()
plt.show()

# Categorical columns
categorical_columns = ['job', 'marital', 'education', 'default', 'housing', 'loan',
                       'contact', 'month', 'day_of_week', 'poutcome', 'y']

plt.figure(figsize=(20, 30))
for i, column in enumerate(categorical_columns, 1):
    plt.subplot(6, 3, i)
    sns.countplot(data=df, x=column, palette='viridis')
    plt.title(column, fontsize=14, fontweight='bold')
    plt.xlabel(None)
    plt.ylabel(None)
    plt.xticks(rotation=45)

plt.tight_layout()
plt.show()

# Data preprocessing
# Removing columns with high correlation
variables_to_remove = ['euribor3m', 'nr.employed', 'emp.var.rate']
df_filtered = df.drop(variables_to_remove, axis=1)

# Removing irrelevant columns
variables_to_remove2 = ['contact', 'month', 'day_of_week', 'poutcome']
df_filtered = df_filtered.drop(variables_to_remove2, axis=1)
```

```
# Categorical Treatment
df_filtered = df_filtered[df_filtered['marital'] != 'unknown']
df_filtered = df_filtered[df_filtered['housing'] != 'unknown']
df_filtered['education'] = df_filtered['education'].replace(['basic.4y', 'basic.6y', 'basic.9y'], 'basic.education')

# Age grouping
num_bins = 5
df_filtered['age_group'] = pd.cut(df_filtered['age'], bins=num_bins, labels=[f'Group {i+1}' for i in range(num_bins)])
df_filtered = df_filtered.drop(columns=['age'])

# Encoding categorical variables
cat_columns = ['job', 'marital', 'education', 'default', 'housing', 'loan', 'age_group','y']
label_encoder = LabelEncoder()
for column in cat_columns:
    df_filtered[column] = label_encoder.fit_transform(df_filtered[column])

# Feature Scaling
num_columns = ['duration', 'campaign', 'pdays', 'previous', 'cons.price.idx', 'cons.conf.idx']
scaler = StandardScaler()
df_filtered[num_columns] = scaler.fit_transform(df_filtered[num_columns])

# ML Model
X = df_filtered.drop(columns=['y'])
y = df_filtered['y']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Decision Tree
dt_classifier = DecisionTreeClassifier()
dt_classifier.fit(X_train, y_train)
y_pred_dt = dt_classifier.predict(X_test)
accuracy_dt = accuracy_score(y_test, y_pred_dt)
print("Decision Tree Accuracy:", accuracy_dt)

# SVM
svm_classifier = SVC()
svm_classifier.fit(X_train, y_train)
y_pred_svm = svm_classifier.predict(X_test)
accuracy_svm = accuracy_score(y_test, y_pred_svm)
print("SVM Accuracy:", accuracy_svm)

# KNN
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
accuracy_knn = accuracy_score(y_test, y_pred_knn)
print("KNN Accuracy:", accuracy_knn)

# Naive Bayes
```

```
nb = GaussianNB()
nb.fit(X_train, y_train)
y_pred_nb = nb.predict(X_test)
accuracy_nb = accuracy_score(y_test, y_pred_nb)
print("Naive Bayes Accuracy:", accuracy_nb)

# Random Forest
rf_classifier = RandomForestClassifier()
rf_classifier.fit(X_train, y_train)
y_pred_rf = rf_classifier.predict(X_test)
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print("Random Forest Accuracy:", accuracy_rf)

# Bagging
base_classifier = DecisionTreeClassifier()
bagging_classifier = BaggingClassifier(base_classifier, n_estimators=10, random_state=42)
bagging_classifier.fit(X_train, y_train)
y_pred_bagging = bagging_classifier.predict(X_test)
accuracy_bagging = accuracy_score(y_test, y_pred_bagging)
print("Bagging Accuracy:", accuracy_bagging)

# XG Boosting
xgb_classifier = xgb.XGBClassifier()
xgb_classifier.fit(X_train, y_train)
y_pred_xgb = xgb_classifier.predict(X_test)
accuracy_xgb = accuracy_score(y_test, y_pred_xgb)
print("XG Boosting Accuracy:", accuracy_xgb)

# Feature importance analysis
feature_importances_xgb = xgb_classifier.feature_importances_
feature_importance_df_xgb = pd.DataFrame({'Feature': X_train.columns, 'Importance':
feature_importances_xgb})
feature_importance_df_xgb = feature_importance_df_xgb.sort_values(by='Importance',
ascending=False)
print("XG Boosting Feature Importance:")
print(feature_importance_df_xgb)

# ROC Curve analysis
y_prob_xgb = xgb_classifier.predict_proba(X_test)[:, 1]
fpr_xgb, tpr_xgb, _ = roc_curve(y_test, y_prob_xgb)
roc_auc_xgb = auc(fpr_xgb, tpr_xgb)
plt.figure(figsize=(8, 6))
plt.plot(fpr_xgb, tpr_xgb, color='green', lw=2, label=f'XG Boosting (AUC = {roc_auc_xgb:.2f})')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

 # Clustering

```
# K-Means
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score
from sklearn.preprocessing import StandardScaler
from sklearn_extra.cluster import KMedoids
from sklearn.neighbors import NearestNeighbors
from sklearn.metrics import pairwise_distances_argmin_min
from scipy.spatial.distance import cdist
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.cluster import DBSCAN

# Calculate Within-Cluster Sum of Squares (WSS) for different values of K
wss = []
for k in range(1, 11):
    k_means_model = KMeans(n_clusters=k)
    k_means_model.fit(X)
    wss.append(k_means_model.inertia_)

# Plot the elbow curve
plt.plot(range(1, 11), wss, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Within-Cluster Sum of Squares (WSS)')
plt.title('Elbow Method')
plt.show()

# Compute silhouette score for different values of K
k_values = range(2, 11)
silhouette_avg = []

for k in k_values:
    kmeans = KMeans(n_clusters=k)
    kmeans_labels = kmeans.fit_predict(X)
    silhouette_avg.append(silhouette_score(X, kmeans_labels))

# Plot silhouette scores for each value of K
plt.plot(k_values, silhouette_avg, 'bo-', markersize=8)
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette Score')
plt.title('Silhouette Score for Different Values of K')
plt.show()

# Choose the optimal number of clusters based on silhouette score
optimal_k = k_values[np.argmax(silhouette_avg)]
print("Optimal number of clusters based on silhouette width:", optimal_k)

# Perform K-means clustering with the optimal number of clusters
```

```
optimal_kmeans_result = KMeans(n_clusters=optimal_k).fit(X)

# Plot the silhouette scores for each value of K
plt.plot(k_values, silhouette_avg, 'bo-', markersize=8)
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette Score')
plt.title('Silhouette Score for Different Values of K')
plt.show()

# K=2 is the best according to WSS
# fitting the model with number of clusters = 2
ClusteringC = KMeans(n_clusters=2)
ClusteringC.fit(X)

# Predict the model
cust_label = ClusteringC.predict(X)
print(cust_label)

cust_centroidC = ClusteringC.cluster_centers_

# Create scatter plot
plt.scatter(X.iloc[:, 6], X.iloc[:, 12], c=cust_label, cmap='viridis')
plt.scatter(cust_centroidC[:, 0], cust_centroidC[:, 12], c='red', marker='X', s=100)
plt.show()

# Making dataframe for y train v/s y predicted using x train
bank_df = pd.DataFrame()
bank_df['y'] = y
bank_df['cluster_label'] = cust_label
print(bank_df)

# Check accuracy
y_pred = bank_df['cluster_label']
y_test = bank_df['y']
y_test_1 = [1 if i == '1' else 0 for i in y_test]
y_test_2 = [1 if i == '0' else 0 for i in y_test]
acc_1 = accuracy_score(y_test_1, y_pred)
acc_2 = accuracy_score(y_test_2, y_pred)
print('if consider 1 in column cluster_label as yes then the accuracy of this column with respect to
the column y will be %f, otherwise the accuracy will be %f' % (acc_1, acc_2))

# Perform K-Means clustering with K=3
ClusteringC = KMeans(n_clusters=3)
ClusteringC.fit(X)

# Predict the model
cust_label = ClusteringC.predict(X)
print(cust_label)

cust_centroidC = ClusteringC.cluster_centers_
```

```
# Create scatter plot
plt.scatter(X.iloc[:, 6], X.iloc[:, 12], c=cust_label, cmap='viridis')
plt.scatter(cust_centroidC[:, 6], cust_centroidC[:, 12], c='red', marker='X', s=100)
plt.show()

# Perform DBSCAN clustering
neigh = NearestNeighbors(n_neighbors=2)
nbrs = neigh.fit(X)
distances, indices = nbrs.kneighbors(X)
distances = np.sort(distances, axis=0)
distances = distances[:, 1]
plt.plot(distances)

# Calculate eps
def calculate_eps(data):
    knn_dist = NearestNeighbors(n_neighbors=2).fit(data)
    distances, _ = knn_dist.kneighbors(data)
    distances = np.sort(distances, axis=0)
    distances = distances[:, 1]
    return distances[-1]

eps_value = calculate_eps(X)
print("Calculated eps value:", eps_value)

# Perform DBSCAN clustering
dbscan_result = DBSCAN(eps=eps_value, min_samples=2).fit(X)

# Plotting the results
plt.scatter(X.iloc[:, 6], X.iloc[:, 12], c=dbscan_result.labels_, cmap='viridis')
plt.xlabel('duration')
plt.ylabel('AGE GROUP')
plt.title('DBSCAN Clustering')
plt.show()

# Perform DBSCAN clustering with higher min_samples
db = DBSCAN(eps=eps_value, min_samples=10).fit(X)

# Get the cluster labels and outliers (noise points)
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).



print('Estimated number of clusters: %d' % n_clusters_)
print('Estimated number of noise points: %d' % n_noise_)

# Plotting the clusters
```

```
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]

for k, col in zip(unique_labels, colors):
    if k == -1:
        col = [0, 0, 0, 1]  # Black color for noise points

    class_member_mask = (labels == k)

    xy = X[class_member_mask]
    plt.plot(xy.iloc[:, 6], xy.iloc[:, 12], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()



import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn_extra.cluster import KMedoids
from sklearn.metrics import silhouette_samples, silhouette_score
from sklearn.preprocessing import StandardScaler

# Assuming df_filtered is the DataFrame
# Sample up to 10,000 rows from df_filtered
df_sampled = df_filtered.sample(n=20000, random_state=42)

# Separate features and target variable
X = df_sampled.drop(columns=['y'])
y = df_sampled['y']

# Normalize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Perform KMedoids clustering with the optimal number of clusters (k=2)
pamk = KMedoids(n_clusters=2, random_state=42).fit(X_scaled)

# Get the cluster labels and medoids
cluster_labels = pamk.labels_
cluster_medoids = pamk.cluster_centers_

# Calculate Silhouette scores
silhouette_avg = silhouette_score(X_scaled, cluster_labels)
silhouette_values = silhouette_samples(X_scaled, cluster_labels)
# Plot Silhouette plot
plt.figure(figsize=(8, 6))
y_lower = 10
```

```
for i in range(2):  # Assuming 2 clusters
    ith_cluster_silhouette_values = silhouette_values[cluster_labels == i]
    ith_cluster_silhouette_values.sort()

    size_cluster_i = ith_cluster_silhouette_values.shape[0]
    y_upper = y_lower + size_cluster_i

    color = plt.cm.viridis(float(i) / 2)
    plt.fill_betweenx(np.arange(y_lower, y_upper),
                      0, ith_cluster_silhouette_values,
                      facecolor=color, edgecolor=color, alpha=0.7)

    # Label the silhouette plots with their cluster numbers at the middle
    plt.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

    # Compute the new y_lower for next plot
    y_lower = y_upper + 10  # 10 for the 0 samples

plt.title("Silhouette plot for PAMK clustering")
plt.xlabel("Silhouette coefficient values")
plt.ylabel("Cluster label")

# The vertical line for average silhouette score of all the values
plt.axvline(x=silhouette_avg, color="red", linestyle="--")

plt.yticks([])  # Clear the yaxis labels / ticks
plt.xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

plt.show()

# Analyze the clustering results based on the Silhouette score
print("Silhouette score:", silhouette_avg)

# Create scatter plot of the clusters
plt.scatter(X_scaled[:, 6], X_scaled[:, 12], c=cluster_labels, cmap='viridis')
plt.scatter(cluster_medoids[:, 0], cluster_medoids[:, 1], c='red', marker='X', s=100)
plt.title('PAMK Clustering')
plt.xlabel('DURATION')
plt.ylabel('AgeGroup')
plt.show()
```