Section C

Thrisha Rajkumar

2024-11-28

SECTION C

1. DECISION TREE – SUPERVIZED LEARNING ALGORITHM

What is Machine Learning – supervised learning?

Machine learning is the ability of the computer to learn from its previous experiences on specific tasks based on the performance measure. The machine learning approach is a very efficient way of automating or solving problems, simplifying the programming required, and performing better than the traditional approach of programming algorithms for each specific task. The machine learning algorithm is accurate with the predictions and evaluates the solution by analyzing errors in the problem and trains itself using the training data set for more accurate results and it can work with large amounts of data effectively.

The Ability of learning is a process of finding statistical regularities or other patterns of data from the training data set and based on the desired outcome and supervision factor during the training of the algorithm it predicts the output of new data or unseen data.

The machine learning algorithms are organized as:

Types of Machine learning systems:

- Supervised Learning
- Unsupervised Learning
- Semi-Supervised Learning
- Reinforcement Learning

Supervised Learning –

Supervised learning is training data is used to develop a function that predicts the desired output values for new inputs. The data set used in supervised learning has defined data set called labels. The training data set is the data of previous inputs and outputs which is trained to predict the outputs for new inputs accurately. It relies on human input to provide both the input data with predetermined classes and the corresponding desired output, along with feedback on the accuracy of predictions during the training process.

The task of the machine learning algorithm is to find the correlation and patterns and construct the statistical and mathematical models. The models are then evaluated on the basis of how accurately it is determining or predicting the results.

Examples of Supervised Learning:

- Decision Trees and Random Forests

- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Neural networks

There are Two Types of Supervised learning:

- Classification – It categorizes the labels and outputs to predict the accurate output for the new input data. Classification supervised learning algorithm also known as classifiers has labels or classes of data points and algorithm predicts the label to which the new input data set belongs to.

- Regression – It determines or predicts a numerical output or value.Regression supervised algorithm takes in the inputs and outputs a continuous numerical output by prediction of the numerical value. Difference between Classification and Regression

Feature	Classification	Regression
Output Type	Categorical or labelled or Discrete outputs	Continuous or numerical outputs
Objective	Identify the appropriate category or class	Predict numerical output/value
Algorithms	Logistic Regression, Support Vector Machines(SVM), Decision Trees (classification)	Linear Regression, Polynomial Regression, Decision Trees (regression)
Evaluation Metrics	Accuracy, Precision, Recall, F1 Score	Mean Squared Error (MSE), R^2 Score

Decision Tree – Supervised Learning algorithm

Decision Tree is in a form of tree structure to represent a number of possible decisions where each internal node corresponds to a decision based on an attribute or label, each branch represents an outcome of that decision, and each leaf node represents a final output or label. It is a hierarchical structure which is formed by the sequence of if and else questions and labelled outputs.

Decision Tree Supervised Learning Algorithm can be applied to model both Classification and Regression Machine learning problems unlike other algorithms as decision trees can easily handle a mix of numeric and categorical attributes and can even classify data for which attributes are missing.

Purpose: Classification: Assigns data points to predefined categories.It categorizes the labels and outputs to predict the accurate output for the new input data. Classification supervised learning algorithm also known as classifiers has labels or classes of data points and algorithm predicts the label to which the new input data set belongs to.

Regression: Predicts continuous values based on input features.

As you can observe in the Table above with differntiation of Classification and Regression, Decision tree is an example of both methods.

CART is an example of application of both Regression and Classification method: CART– Classification and Regression Trees (CART), commonly known as decision trees, can be represented as binary trees. They have the advantage to be very interpretable.

Random Forest is another example of an learning model which is capable of performing both regression and classification tasks. Random tree is an ensemble machine learning algorithm which simplifies the root data node into subsets and then creates a decision tree of each sample or subset and then predicts the output by combining the training model of the decision trees of the subsets to get more accurate and precise result.

We will be covering the CART and Random Forest further in the report.

Structure of Decision tree

The decision tree terms or what they are made up of are:

1. Root Node: the root node consists of the whole dataset and is at the top if the tree hierarchical structure.
2. Internal nodes - the nodes which are breached out of the nodes
3. Leaf nodes - they are the child of the parent node which are attached to the node of the tree.

How Decision Trees Work

The Decision Tree splits the dataset into smaller subsets according to the feature/labels input values. Further recursion process goes on until the preferred depth is reached or when a certain stopping criterion is met. This will be done with the intent of maximizing homogeneity among target variables in the subsets. Major nodes, which are decision points, and edges denoting decision paths, make up these trees, while results from classification or regression are accomplished from leaf nodes.

Key Principles of Decision Trees

1. Hierarchical Structure: The tree is built as a hierarchical structure or tree like structure which helps in sequencing the decisions.
2. Gini Impurity, Entropy, or Variance: these are the metrics used to measure the quality of splits.
3. Interpretability: Easy to interpret due to tree like structure and easy to visualize the tree structure, making them useful for decision making.
4. Recursive Partitioning: The algorithm splits the data into subsets to improve homogeneity of the target variable in each subset.

Formulas used for the problem to visualize and model the dataset using Decision Tree

Splitting Criteria

The decision on splitting the dataset at each node is known as splitting criteria which can be determined by criterias such as Gini Index, Entropy, Gain Ratio, and Information gain. These are known as “attribute selection measure” or “splitting rules” because of its use for selecting the splitting criterion that “best” separates a given data partition.

1. Gini Index

The Gini Index measures the impurity of a decision node. A lower Gini Index indicates a more Standardized and Consistent node. The Gini index measures are more sensitive to the node probabilities. The Giniindex is used in CART.

2. Entropy

Entropy measures the randomness in the data. The main objective using Entropy for classification in the decision tree is to minimize entropy at each split.

3. Information Gain

Information Gain used to detect the best split measures the reduction of entropy after a split. higher Information Gain -> better split

4. Mean Squared Error (MSE)

For regression trees, Mean Squared Error (MSE) is used to measure the impurity of decision nodes.

5. Classification Error Rate

The Classification Error Rate measures the mis-classification rate

6. Pruning

Pruning reduces the size of the tree to prevent overfitting. Two main types are:

-> Pre-pruning: Stops the tree early based on a predefined criterion.

-> Post-pruning: removes branches that have minimal impact on model performance after the tree is complete

- Cross-validation should be used to evaluate the model's performance.
- Feature importance can be derived from tree-based models.
- Bagging and Random Forests can improve performance and stability by combining multiple trees.

PREVENTING OVERFITTING IN DECISION TREE

The rules of criteria to prevent the overfitting in the desicion tree the following are the standard measures used:

- Maximum Depth: limits depth of the decision tree.
- Minimum Samples per Leaf Node: The minimum number of samples required in a leaf node.
- Minimum Information Gain etc.

How the training algorithm works

The training of a Decision Tree algorithm is a process of recursive spliting and partitioning the data at each node with the best feature by evaluating the splits with the Splitting Criterias discussed above which is as follows:

-> Root Node: The entire dataset is considered at the root node when starting and then when spilt into seperate classes two other nodes are built and the dataset splits in sets.

-> Evaluate Splits: The algorithm chooses the feature that minimizes impurity (like Gini Index, Entropy, or MSE) as discussed above to split or partition the data.

-> Recursive decision nodes spilting : The evaluation and splitting keeps being implemented until the desired outputs and the dataset cannot be split further.

-> Leaf Nodes: Once no further splitting is possible, each leaf node is assigned a class label (for classification) or predicted value (for regression).

How new predictions are made on test data

After the Decision Tree is trained, it can make predictions on new data by following these steps:

-> Input Data: Each data will be passed through the training data and traverses through each nodes and in the end it fits into a label or class.

Classification or Regression:

-> Classification: The predicted class is the majority class in the leaf node.

-> Regression: The predicted value is the average of the values in the leaf node.

-> Model Evaluation: After predictions, the performance is evaluated by using metrics such as accuracy, precision, recall, AUC-ROC curve and F1-score for classification or Mean Squared Error for regression.

For overfitting, cross-validation and pruning methods can be applied to ensure that the model generalizes well when confronted with previously unseen data.

Why am I choosing decision trees

-> The reason for the choice of Decision tree for this report is to explore the understanding and deep analysis on the data using both classification and regression methods.

Appropriate Problem Types

-> Suitable for datasets with both categorical and numerical features, solving classification (Loan Eligibility) or regression problems.

-> Their ability to provide interpretable models makes them a good choice for analyzing and explaining decision-making processes.

Type of Problems this method is appropriate for

1. Classification Problems

Decision Trees are widely used in classification tasks where the target variable is categorical.

Examples include: Loan Approval: Predicting whether a loan application will be approved based on factors like income, credit score, and co-applicant income etc.

Email Spam Detection: a very common example used - Classifying emails as spam or not based on features like keywords, sender's address, and email content.

2. Regression Problems

They are also used for regression tasks with continuous target variables. Examples include: House Price Prediction: Estimating house prices based on features like size, location, and number of rooms.

Sales Forecasting: Predicting future sales based on historical data and trends.

2. EXPLAINATION OF THE DATASET

DATASET UNDERSTANDING

Data set chosen for this report is Financial Loan Approval Prediction data

Source Kaggle - “https://www.kaggle.com/datasets/krishnaraj30/finance-loan-approval-prediction-data”

The data set is downloaded and loaded in the Folder to perform calculations and modelling for this report.

There are two csv files - train.csv and test.csv files.

Basically, Train.csv file - It will be used for training the model and split to validation dataset to observe the performance using metrics mentioned before.

Test.csv file - will be used to test the model from the training data and predict the values of the Loan_Status approved or rejected using the modelling and further validate the data output by using various hyperparameter and also exploring tuning of the hyperparameters for more accuracy in the predicted results.

Firstly, Lets load the datasets for implementation using R i.e, test.csv file and train.csv file into a data frame for manipulating and use of the data.

```r
#installing the necessary packages for loading the Data from - "Finance Loan Approval Prediction Data"
#tidyverse is the library used to load the data sets
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

```r
# Load train.csv and test.csv files into a data frames - "train_data" and "test_data"
train_data <- read.csv("train.csv")
test_data <- read.csv("test.csv")
```

To understand the variables in the train and test dataset’s - train_data and test_data, we are using str() which is a function used to display the variables of each column header.

```r
str(train_data)
```

```
## 'data.frame':    614 obs. of  13 variables:
##  $ Loan_ID          : chr  "LP001002" "LP001003" "LP001005" "LP001006" ...
##  $ Gender           : chr  "Male" "Male" "Male" "Male" ...
##  $ Married          : chr  "No" "Yes" "Yes" "Yes" ...
##  $ Dependents       : chr  "0" "1" "0" "0" ...
##  $ Education        : chr  "Graduate" "Graduate" "Graduate" "Not Graduate" ...
##  $ Self_Employed    : chr  "No" "No" "Yes" "No" ...
##  $ ApplicantIncome  : int  5849 4583 3000 2583 6000 5417 2333 3036 4006 12841 ...
##  $ CoapplicantIncome: num  0 1508 0 2358 0 ...
##  $ LoanAmount       : int  NA 128 66 120 141 267 95 158 168 349 ...
##  $ Loan_Amount_Term : int  360 360 360 360 360 360 360 360 360 360 ...
##  $ Credit_History   : int  1 1 1 1 1 1 1 0 1 1 ...
##  $ Property_Area    : chr  "Urban" "Rural" "Urban" "Urban" ...
##  $ Loan_Status      : chr  "Y" "N" "Y" "Y" ...
```

Head of the train_data:

```
# head of the train dataset
head(train_data)
```

```
##    Loan_ID Gender Married Dependents    Education Self_Employed ApplicantIncome
## 1 LP001002   Male      No          0     Graduate            No            5849
## 2 LP001003   Male     Yes          1     Graduate            No            4583
## 3 LP001005   Male     Yes          0     Graduate           Yes            3000
## 4 LP001006   Male     Yes          0 Not Graduate            No            2583
## 5 LP001008   Male      No          0     Graduate            No            6000
## 6 LP001011   Male     Yes          2     Graduate           Yes            5417
##   CoapplicantIncome LoanAmount Loan_Amount_Term Credit_History Property_Area
## 1                 0         NA              360              1         Urban
## 2              1508        128              360              1         Rural
## 3                 0         66              360              1         Urban
## 4              2358        120              360              1         Urban
## 5                 0        141              360              1         Urban
## 6              4196        267              360              1         Urban
##   Loan_Status
## 1           Y
## 2           N
## 3           Y
## 4           Y
## 5           Y
## 6           Y
```

Now Understanding the variables in the test_data:

```
str(test_data)
```

```
## 'data.frame':    367 obs. of  12 variables:
##  $ Loan_ID          : chr  "LP001015" "LP001022" "LP001031" "LP001035" ...
##  $ Gender           : chr  "Male" "Male" "Male" "Male" ...
##  $ Married          : chr  "Yes" "Yes" "Yes" "Yes" ...
##  $ Dependents       : chr  "0" "1" "2" "2" ...
##  $ Education        : chr  "Graduate" "Graduate" "Graduate" "Graduate" ...
##  $ Self_Employed    : chr  "No" "No" "No" "No" ...
##  $ ApplicantIncome  : int  5720 3076 5000 2340 3276 2165 2226 3881 13633 2400 ...
##  $ CoapplicantIncome: int  0 1500 1800 2546 0 3422 0 0 0 2400 ...
##  $ LoanAmount       : int  110 126 208 100 78 152 59 147 280 123 ...
##  $ Loan_Amount_Term : int  360 360 360 360 360 360 360 360 240 360 ...
##  $ Credit_History   : int  1 1 1 NA 1 1 1 0 1 1 ...
##  $ Property_Area    : chr  "Urban" "Urban" "Urban" "Urban" ...
```

Head of the test_data:

```r
# head of the test dataset
head(test_data)
```

```
##    Loan_ID Gender Married Dependents    Education Self_Employed ApplicantIncome
## 1 LP001015   Male     Yes          0     Graduate            No            5720
## 2 LP001022   Male     Yes          1     Graduate            No            3076
## 3 LP001031   Male     Yes          2     Graduate            No            5000
## 4 LP001035   Male     Yes          2     Graduate            No            2340
## 5 LP001051   Male      No          0 Not Graduate            No            3276
## 6 LP001054   Male     Yes          0 Not Graduate           Yes            2165
##   CoapplicantIncome LoanAmount Loan_Amount_Term Credit_History Property_Area
## 1                 0        110              360              1         Urban
## 2              1500        126              360              1         Urban
## 3              1800        208              360              1         Urban
## 4              2546        100              360             NA         Urban
## 5                 0         78              360              1         Urban
## 6              3422        152              360              1         Urban
```

A brief summary of number of columns and and column headers names of the dataset - train_data:

```r
# Number of columns and column names for train dataset
train_columns <- ncol(train_data)
train_column_names <- colnames(train_data)

# Displaying the no. of colummns and the column names to have more clarity upon the data we are working

cat("Train Data - Number of Columns:", train_columns, "\n")
```

```
## Train Data - Number of Columns: 13
```

```r
cat("Train Data - Column Names:\n", train_column_names, "\n\n")
```

```
## Train Data - Column Names:
##  Loan_ID Gender Married Dependents Education Self_Employed ApplicantIncome CoapplicantIncome LoanAmou
```

Similarly, for test_data:

```r
# Number of columns and column names for test dataset
test_columns <- ncol(test_data)
test_column_names <- colnames(test_data)

cat("Test Data - Number of Columns:", test_columns, "\n")
```

```
## Test Data - Number of Columns: 12
```

```r
cat("Test Data - Column Names:\n", test_column_names, "\n")
```

```
## Test Data - Column Names:
##  Loan_ID Gender Married Dependents Education Self_Employed ApplicantIncome CoapplicantIncome LoanAmou
```

Summary of the Data sets - train_data and test_data are:

Train Dataset: Features: 13 Target Variable: Loan_Status (Binary - 'Y' or 'N') and NA or null values which will be treated during the data preprocessing

Test Dataset: Features: 12 12 columns and 367 entries (no Loan_Status column as it is to be predicted after model training).

As we can observe in the data exploration below are the feature types in the test_data and train_data

Feature Types:

1. Categorical: Loan_ID, Gender, Married, Dependents, Education, Self_Employed, Property_Area
2. Binary Categorical: Loan_Status, Credit_History
3. Ordinal: Dependents (0, 1, 2, 3+)
4. Continuous: ApplicantIncome, CoapplicantIncome, LoanAmount, Loan_Amount_Term

This bifurcation of features is very important for treating and handling missing values during the data pre-processing.

Problem Statement

The problem statement for this report is to build an automated loan eligibility prediction system to predict whether a customer is likely to get approval for a loan based on several factors like income, the amount of loan, credit history, and so on and enable the company to make data-driven and real-time decisions.

We will be implementing the Decision Tree classification model to accomplish this. A decision tree classification model is most likely suitable for this problem because:

- Decision Trees provide clear and interpretable flowcharts regarding decisions which help in visulaization using rpart.plot(model).
- They effectively handle categorical and continuous data.
- Allow for non-linear decision boundaries; hence, robust for varied customer profiles.

The model will learn from the training dataset to classify the loan applications into the following two categories: - Loan Approved (Y) - Loan Not Approved (N)

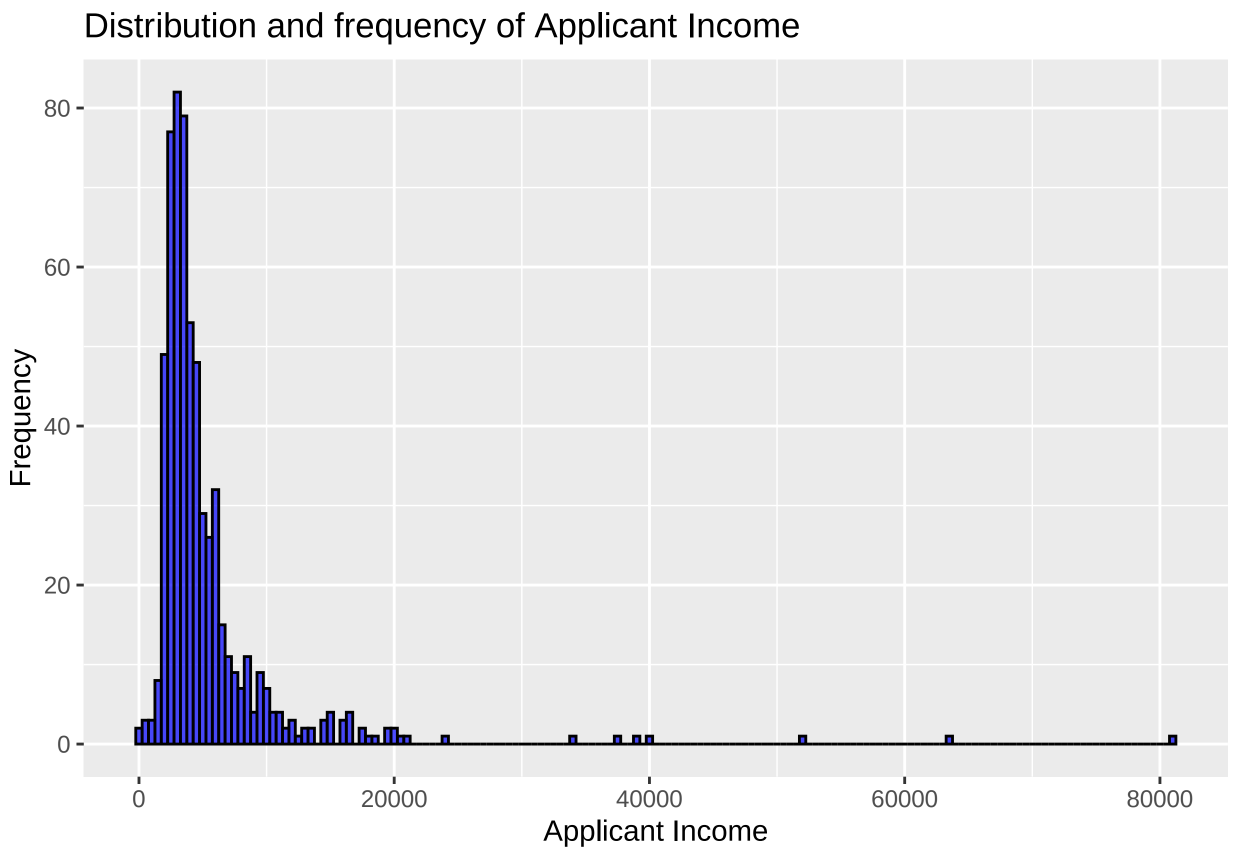
and test, validate, and evaluate the model using the test_data.

Visualizing the dataset will give us more information on how data is spread across the scales and also we can get the information and better insights on Distribution, density, Outlier analysis, Influence and relations between data points and more.

Below we have plotted visualizations using ggplot2 library packages:

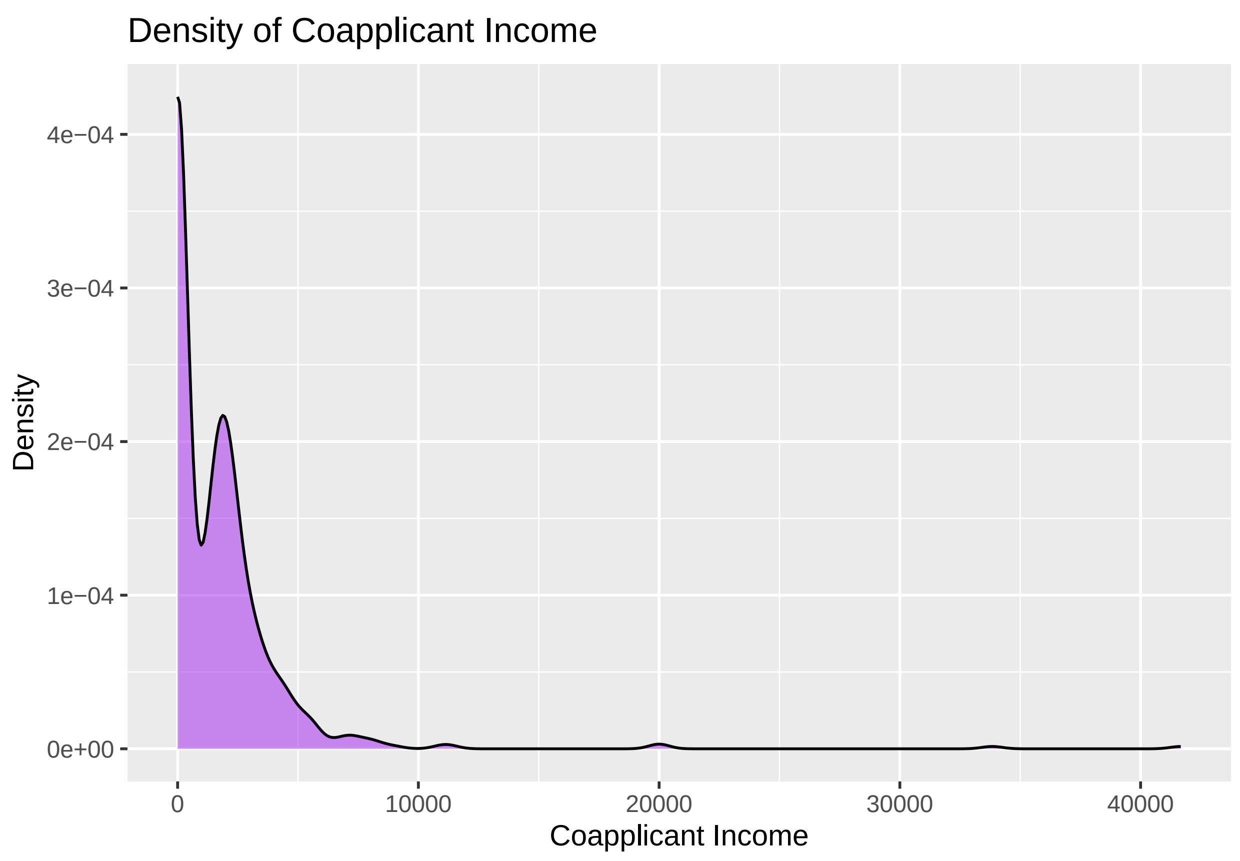
- Distribution of the Income of the customer It is an important factor which helps us determine the variance and extreme and outlier data of the incomes. we are using histogram with the X-axis as ApplicantIncome and Y-axis as the Frequency of the income which helps use visually interpret the mode, or median value to understand the data better.Used geom_histograam function from the ggplot2 library.

```r
# Using ggplot2 packages to use the histogram graph to see the outliers and distribution.

library(ggplot2)

# Here below as you can see we have taken the respective x and y axes as Applicant Income and Frequency
# Used the color Blue to fill with balck borders with width of the bins - binwidth = 500 and transparen
ggplot(train_data, aes(x = ApplicantIncome)) +
  geom_histogram(binwidth = 500, fill = "blue", color = "black", alpha = 0.7) +
  labs(title = "Distribution and frequency of Applicant Income", x = "Applicant Income", y = "Frequency
```



Similarly, We can also plot what is the density of the co-applicant income as it is also an important factor for loan apporval status which we are focusing on in this report to predict for test_data. Below Density plot is derived or visualized for co-applicant income and we again used geom_density function of the ggplot2 library.

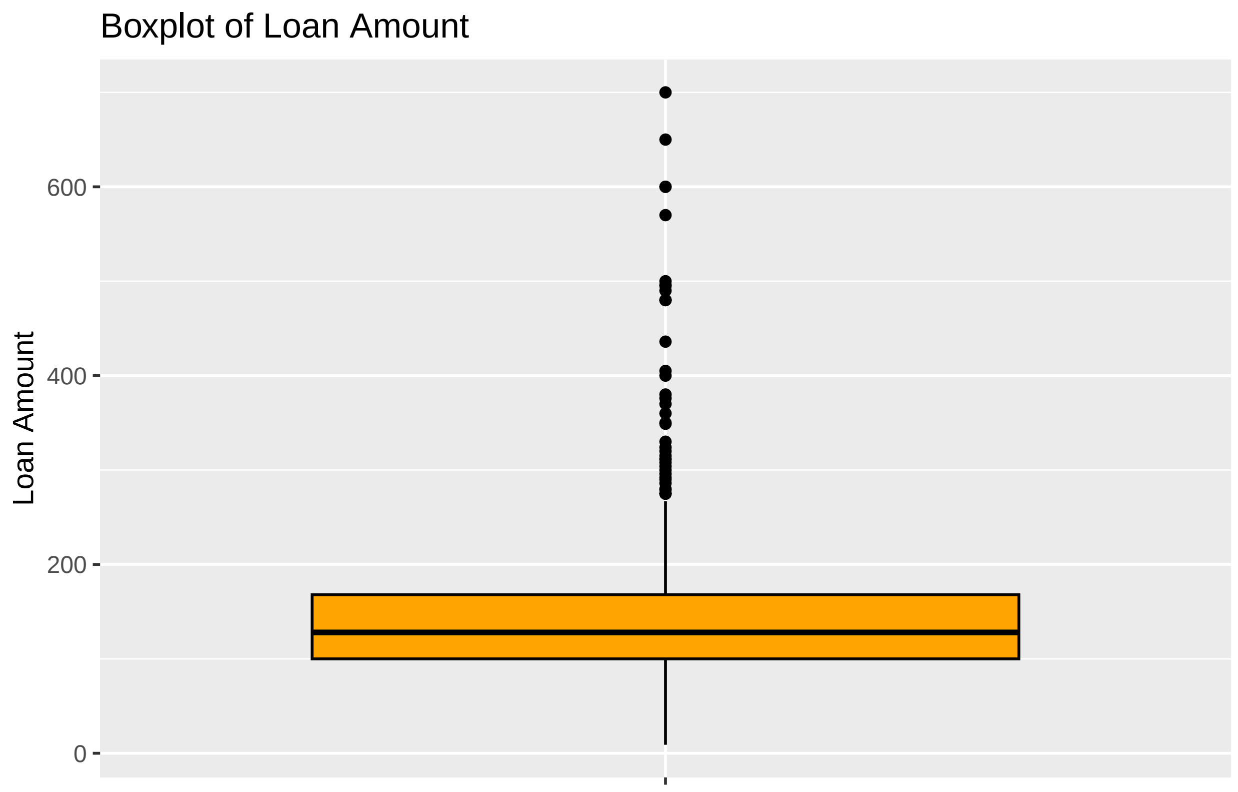
```r
# Added the density function for the Co-applicatnts income with geom_density function and filled purple
ggplot(train_data, aes(x = CoapplicantIncome)) +
  geom_density(fill = "purple", alpha = 0.5) +
  labs(title = "Density of Coapplicant Income", x = "Coapplicant Income", y = "Density")
```



- Analysing the Loan Amount To understand the outliers effect in the training dataset it is necessary to analyze the loan amount of the trian data. we are using boplot funtion from ggplot2 library on the lone amount column.

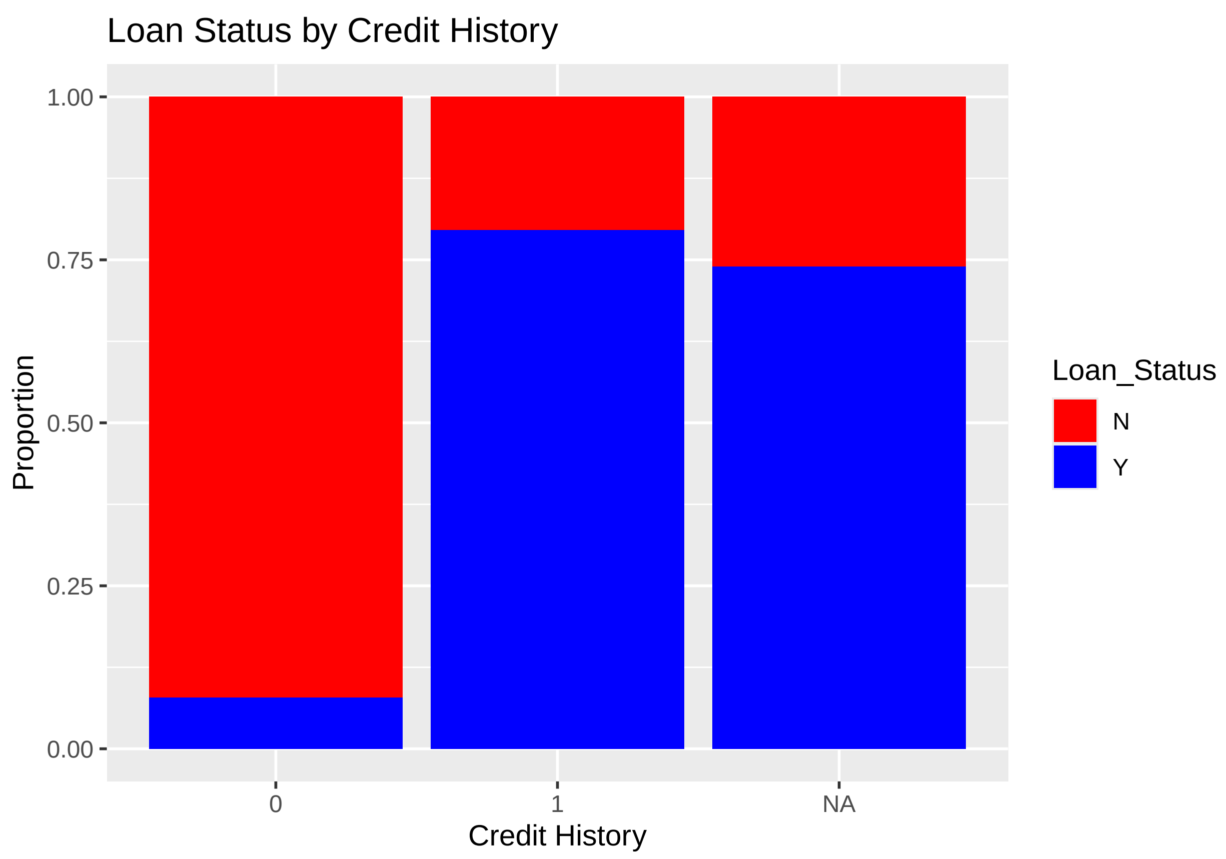
```
# Mentioning x as an empty string to solely understaand only the Loan Amount column in the data set
# Using boxplot for loan amount and fill = orange
ggplot(train_data, aes(x = "", y = LoanAmount)) +
  geom_boxplot(fill = "orange", color = "black") +
  labs(title = "Boxplot of Loan Amount", x = "", y = "Loan Amount")
```

```
## Warning: Removed 22 rows containing non-finite outside the scale range
## (‘stat_boxplot()‘).
```



- Loan Status and Credit History For the problem statement which is loan approval or rejection is proportional to understanding the influence of loan status by the Credit history. Therefore, below we are plotting a geom_bar fntion from ggplot2 library which is a bar plot for the influence analysis. Investigate how credit history influences loan approval using a bar plot.

```
# Using the bar plot for plotting the understanding the relevance Loan status and credit history.
# in x axis we have used Credit history, factor() is used to treat it as a categorical variable
# And with fill = Loan_Status it means it will use the values of loan status which is yes, no and null
ggplot(train_data, aes(x = factor(Credit_History), fill = Loan_Status)) +
  geom_bar(position = "fill") +
  labs(title = "Loan Status by Credit History", x = "Credit History", y = "Proportion") +
  scale_fill_manual(values = c("Y" = "blue", "N" = "red"))
```



If you notice, we have omitted the null values in the heatmap plot to provide a clearer and more effective visualization. In contrast, in the previous plots, we did not omit the null values because visualizing the NA values is important. These values cannot be completely omitted, as addressing them later using techniques like the mean, median, or mode will give us a better understanding of the data, along with a clearer visualization.

- Heatmap for analysis the correlation and relationships between numeric variables

```
# loading libraries ggplot2 for plotting and visualizing the correlation and data visualization, especi
# Using reshape2 as it provides melt() function to reshape data - correlation matrix into a long format
# dplyr is Used for data manipulation for thhe heatmap.

library(ggplot2)
library(reshape2)
```

```
## Warning: package 'reshape2' was built under R version 4.4.2
```

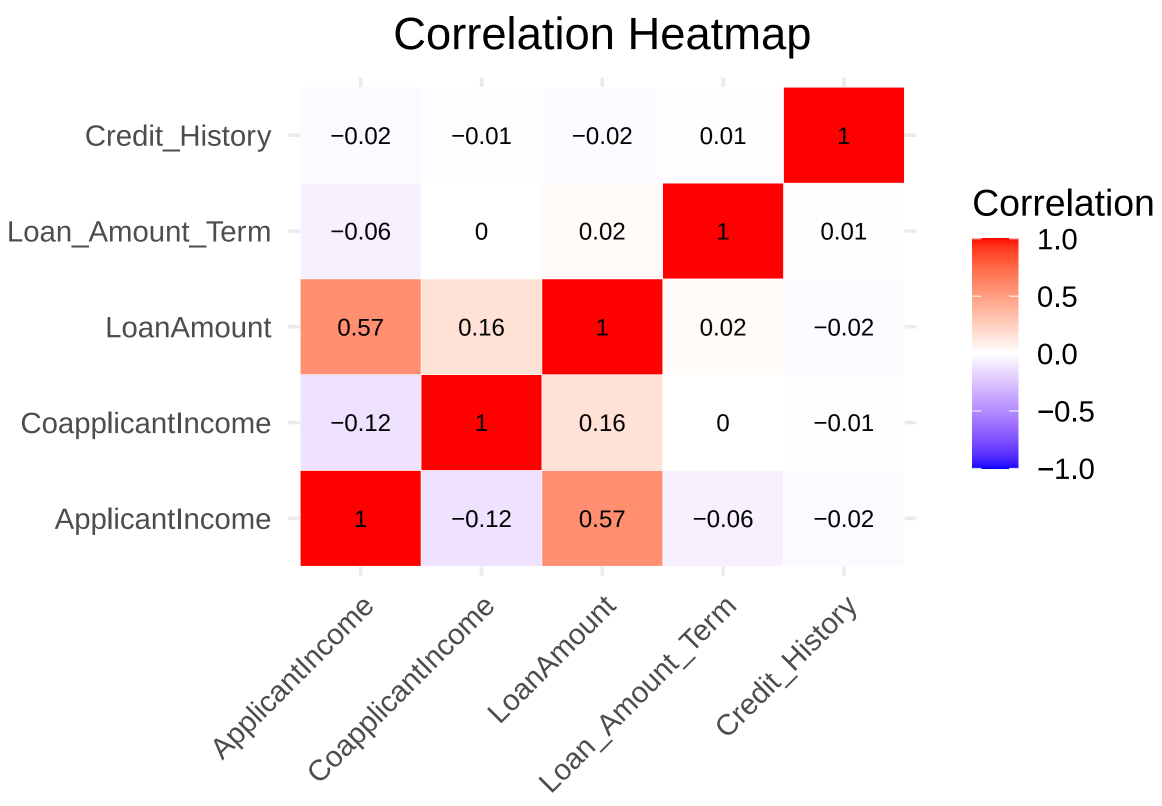
```
##
## Attaching package: 'reshape2'
```

```
## The following object is masked from 'package:tidyr':
##
##     smiths
```

```r
library(dplyr)

# Numerical values - ApplicantIncome, CoapplicantIncome, LoanAmount, Loan_Amount_Term, Credit_History a

numeric_data <- train_data %>%
  select(ApplicantIncome, CoapplicantIncome, LoanAmount, Loan_Amount_Term, Credit_History) %>%
  na.omit() # Used for omitting na values in the above columns

# Correlation matrix is basically the calculation of correlation between the variables and they have th
#-1 (perfect negative correlation)
# +1 (perfect positive correlation)
# A value of 0 indicates no correlation.

# Confusion matrix cor function
cor_matrix <- cor(numeric_data)

# melting the correlation matrix for longer format data
melted_cormat <- melt(cor_matrix)
print(cor_matrix) # displaying the matrix
```

```
##                   ApplicantIncome CoapplicantIncome  LoanAmount
## ApplicantIncome        1.00000000     -0.1226305807  0.57070849
## CoapplicantIncome     -0.12263058      1.0000000000  0.15915197
## LoanAmount             0.57070849      0.1591519703  1.00000000
## Loan_Amount_Term      -0.06286105     -0.0002895206  0.02323917
## Credit_History        -0.02377860     -0.0108471425 -0.01815573
##                   Loan_Amount_Term Credit_History
## ApplicantIncome      -0.0628610527    -0.02377860
## CoapplicantIncome    -0.0002895206    -0.01084714
## LoanAmount            0.0232391675    -0.01815573
## Loan_Amount_Term      1.0000000000     0.00865753
## Credit_History        0.0086575296     1.00000000
```

```r
# Melting correlation matrix help to modify for more accuracy that is the correlation matirx in long-fo
# Var 1 -> x axis and Var2 -> y axis and filled with accordance of the values

ggplot(melted_cormat, aes(x = Var1, y = Var2, fill = value)) +
  geom_tile(color = "white") +  # geom_tile function is used for the border colour here in this heatmap
  scale_fill_gradient2(low = "blue", high = "red", mid = "white", # Here we are defining the negative c
                       midpoint = 0, limit = c(-1, 1), space = "Lab", #Defining the limits and midpoint
                       name = "Correlation") +
  geom_text(aes(label = round(value, 2)), color = "black", size = 3) +
  labs(title = "Correlation Heatmap", x = "", y = "") +# Heading of the heatmap and rounf value to 2 fl
  theme_minimal(base_size = 14) + # used to set the base font as 14
  theme(
    axis.text.x = element_text(angle = 45, vjust = 1, hjust = 1),  # Initially the variable text were o
    plot.title = element_text(hjust = 0.5),  # Adjusting the title position
  )
```



As we can observe in the above heatmap the diagonal denotes 1 as each variable correalation to its own will give the output as a positive correlation which is +1. And Applicant Income and Loan amount has the highest correlation with value -> +0.57.

Below we are grouping a plotting histogram for Loan Amount, Loan Amount Term and Bar chart for the Credit history:

```r
library(ggplot2)
library(tidyr)
library(gridExtra)
```

```
## Warning: package 'gridExtra' was built under R version 4.4.2
```

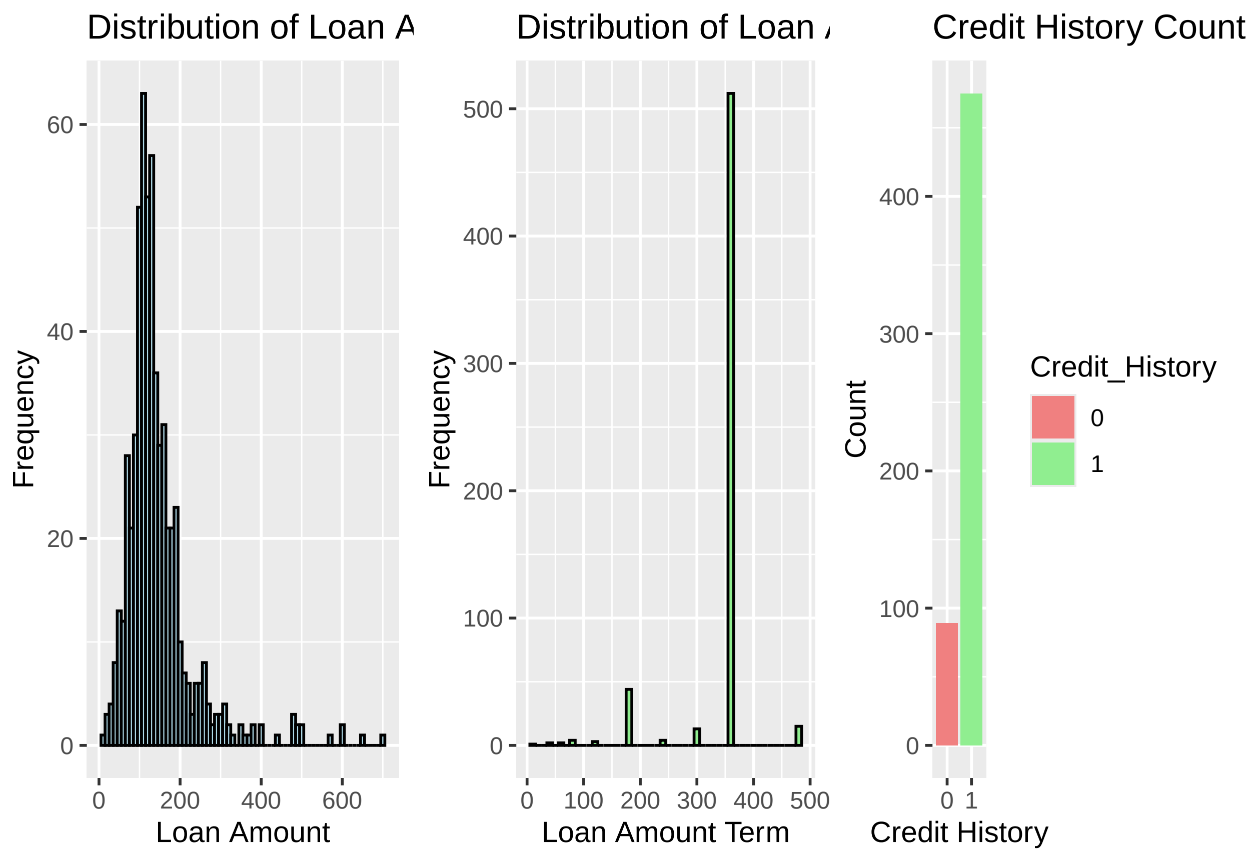
```
##
## Attaching package: 'gridExtra'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```r
# creating a variable p1 to store the plotting of the Histogram for Loan Amount values
p1 <- ggplot(train_data, aes(x = LoanAmount)) +
  geom_histogram(binwidth = 10, fill = "lightblue", color = "black") +
  labs(title = "Distribution of Loan Amount", x = "Loan Amount", y = "Frequency")
```

```r
# creating a variable p2 to store the plotting of the Histogram for Loan_Amount_Term
p2 <- ggplot(train_data, aes(x = Loan_Amount_Term)) +
  geom_histogram(binwidth = 10, fill = "lightgreen", color = "black") +
  labs(title = "Distribution of Loan Amount Term", x = "Loan Amount Term", y = "Frequency")

# credit_history_counts data frame for the frequency table of "Credit_History"
credit_history_counts <- as.data.frame(table(train_data$Credit_History))
names(credit_history_counts) <- c("Credit_History", "Count")

# creating a variable p3 to store the plotting of the Bar Chart for Credit History
p3 <- ggplot(credit_history_counts, aes(x = Credit_History, y = Count, fill = Credit_History)) +
  geom_bar(stat = "identity") +
  labs(title = "Credit History Count", x = "Credit History", y = "Count") +
  scale_fill_manual(values = c("lightcoral", "lightgreen"))

# Combining all plots in a single view
grid.arrange(p1, p2, p3, ncol = 3)
```

```
## Warning: Removed 22 rows containing non-finite outside the scale range
## (‘stat_bin()‘).

## Warning: Removed 14 rows containing non-finite outside the scale range
## (‘stat_bin()‘).
```



3. Model Performance Evaluation and Metrics Selection

Data Preprocessing and Data cleaning

Data pre-processing ensures that the dataset is clean, reliable, and suitable for model training. Following is the detail of the data-cleaning steps performed before the training of a Decision Tree model.

Firstly, before proceeding to model performance evaluation and metrics selection lets begin with the Data Cleaning and pre-processing which is an essential step as it helps to give more accurate results and also helps to ensure that we are using a high-quality dataset for training and evaluation.

Before training the data with Decision Tree model, lets perform some data cleaning steps:

Handling Missing values

Below, we will use the mode and median formulas to replace the null values in the dataset, rather than removing them entirely. Omitting missing values would result in the loss of many rows, which could negatively impact the model's performance, making it less robust and less effective in predicting loan approval or rejection.

```
# Replacing the missing values rather than omitting or removal of the data with Null values to make the

# printing the missing values in the dataset - train_data, considering both NA and empty strings
missing_values <- colSums(is.na(train_data) | train_data == "")
# Checking the missing values
print(missing_values)
```

```
##           Loan_ID            Gender           Married        Dependents
##                 0                13                 3                15
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome
##                 0                32                 0                 0
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area
##                22                14                50                 0
##       Loan_Status
##                 0
```

```
# replacing the missing values in the "Loan Amount" column with numerical variable data
# there are 22 missing values as we can see in the output of sum of NA values of columns
# We can use mean or median for replacing the data
# Mean might not be the right fit due to the differences between extreme loan amounts
# Median of loan amount to fill in the missing values

# Using the median function on the LoanAmount column in train_data
train_data$LoanAmount[is.na(train_data$LoanAmount)] <- median(train_data$LoanAmount, na.rm = TRUE)

# After the replacement checking the missing values again in the columns to treat the other columns and
missing_values <- colSums(is.na(train_data) | train_data == "")
print(missing_values)
```

```
##           Loan_ID            Gender           Married        Dependents
##                 0                13                 3                15
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome
##                 0                32                 0                 0
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area
##                 0                14                50                 0
##       Loan_Status
##                 0
```

```
# Replacing the missing values in the Loan_Amount_Term by first interpreting the mode and then replacin
# Here we are using mode as it is the only way we could replace the missing values as median and mean c
mode_loan_term <- as.numeric(names(sort(table(train_data$Loan_Amount_Term), decreasing = TRUE)[1]))
train_data$Loan_Amount_Term[is.na(train_data$Loan_Amount_Term)] <- mode_loan_term

# Here we used 40 as head because the 40th value in the dataset was a na value and to check if it is re
# head funtion is used to display the first few rows of the column to check the replacement
head(train_data$Loan_Amount_Term,40)
```

```
##  [1] 360 360 360 360 360 360 360 360 360 360 360 360 360 360 120 360 240 360 360
## [20] 360 360 360 360 360 360 360 360 360 360 360 360 360 360 360 360 360 360 360
## [39] 360 360
```

```
# Displaying a summary of the Loan_Amount_Term column to check its distribution after replacement
summary(train_data$Loan_Amount_Term)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    12.0   360.0   360.0   342.4   360.0   480.0
```

```
# Similarly for Gender and Married columns using mode to fill the missing values
mode_gender <- as.character(names(sort(table(train_data$Gender), decreasing = TRUE)[1])) #first sorting
train_data$Gender[is.na(train_data$Gender)] <- mode_gender

#After the replacement checking the missing values again in the columns
missing_values <- colSums(is.na(train_data) | train_data == "")
print(missing_values)
```

```
##           Loan_ID            Gender           Married        Dependents
##                 0                13                 3                15
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome
##                 0                32                 0                 0
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area
##                 0                 0                50                 0
##       Loan_Status
##                 0
```

```
# replacing empty values with NA in the Gender column because the missing values are not recognized to 

train_data$Gender[train_data$Gender == ""] <- NA

# mode of the Gender column
mode_gender <- names(sort(table(train_data$Gender), decreasing = TRUE)[1])

# replacing missing values with the mode of the column
train_data$Gender[is.na(train_data$Gender)] <- mode_gender
```

```r
missing_values <- colSums(is.na(train_data) | train_data == "")
print(missing_values)
```

```
##           Loan_ID            Gender           Married        Dependents 
##                 0                 0                 3                15 
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome 
##                 0                32                 0                 0 
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area 
##                 0                 0                50                 0 
##       Loan_Status 
##                 0
```

```r
head(train_data$Gender, 20)
```

```
##  [1] "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Male"  
##  [9] "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Male"  
## [17] "Male"   "Female" "Male"   "Male"
```

```r
# replacing empty values with NA in the Married column
train_data$Married[train_data$Married == ""] <- NA

# mode of the married column
mode_Married <- names(sort(table(train_data$Married), decreasing = TRUE)[1])

# replacing missing values with the mode of the column
train_data$Married[is.na(train_data$Married)] <- mode_Married

missing_values <- colSums(is.na(train_data) | train_data == "")
print(missing_values)
```

```
##           Loan_ID            Gender           Married        Dependents 
##                 0                 0                 0                15 
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome 
##                 0                32                 0                 0 
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area 
##                 0                 0                50                 0 
##       Loan_Status 
##                 0
```

```r
head(train_data$Married, 20)
```

```
##  [1] "No"  "Yes" "Yes" "Yes" "No"  "Yes" "Yes" "Yes" "Yes" "Yes" "Yes" "Yes"
## [13] "Yes" "No"  "Yes" "No"  "No"  "No"  "Yes" "Yes"
```

```r
# Similarly for Dependents, Self_Employed columns
# Using mode for Dependents and Self_Employed columns as they are categorical feature types
# Again setting the null values or empty cells as "NA"
train_data$Dependents[train_data$Dependents == ""] <- NA
mode_dependents <- names(sort(table(train_data$Dependents), decreasing = TRUE)[1])
train_data$Dependents[is.na(train_data$Dependents)] <- mode_dependents

head(train_data$Dependents,20)
```

```
##  [1] "0"  "1"  "0"  "0"  "0"  "2"  "0"  "3+" "2"  "1"  "2"  "2"  "2"  "0"  "2"
## [16] "0"  "1"  "0"  "0"  "0"
```

```
#Similarly for Self_employed
train_data$Self_Employed[train_data$Self_Employed == ""] <- NA
mode_self_employed <- names(sort(table(train_data$Self_Employed), decreasing = TRUE)[1])
train_data$Self_Employed[is.na(train_data$Self_Employed)] <- mode_self_employed

head(train_data$Self_Employed,21)
```

```
##  [1] "No"  "No"  "Yes" "No"  "No"  "Yes" "No"  "No"  "No"  "No"  "No"  "No"
## [13] "No"  "No"  "No"  "No"  "No"  "No"  "No"  "No"  "No"
```

```
# Credit_history column has a binary categorical type therefore we will be using mode for this data col

train_data$Credit_History[train_data$Credit_History == ""] <- NA
mode_credit_history <- names(sort(table(train_data$Credit_History), decreasing = TRUE)[1])
train_data$Credit_History[is.na(train_data$Credit_History)] <- mode_credit_history

head(train_data$Credit_History,26)
```

```
##  [1] "1" "1" "1" "1" "1" "1" "1" "0" "1" "1" "1" "1" "1" "1" "1" "1" "1" "0" "1"
## [20] "1" "0" "1" "0" "0" "1" "1"
```

```
# Checking in the end for making sure all the missing values are treated
missing_values <- colSums(is.na(train_data) | train_data == "")
print(missing_values)
```

```
##           Loan_ID            Gender           Married        Dependents
##                 0                 0                 0                 0
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome
##                 0                 0                 0                 0
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area
##                 0                 0                 0                 0
##       Loan_Status
##                 0
```

```
# summary() - provides the sumaary of all the data column in train_data
summary(train_data)
```

```
##    Loan_ID              Gender             Married            Dependents
##  Length:614         Length:614         Length:614         Length:614
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##   Education         Self_Employed      ApplicantIncome CoapplicantIncome
##  Length:614         Length:614         Min.   :  150   Min.   :    0
##  Class :character   Class :character   1st Qu.: 2878   1st Qu.:    0
##  Mode  :character   Mode  :character   Median : 3812   Median : 1188
##                                        Mean   : 5403   Mean   : 1621  
##                                        3rd Qu.: 5795   3rd Qu.: 2297  
##                                        Max.   :81000   Max.   :41667  
##    LoanAmount    Loan_Amount_Term Credit_History     Property_Area     
##  Min.   :  9.0   Min.   : 12.0    Length:614         Length:614        
##  1st Qu.:100.2   1st Qu.:360.0    Class :character   Class :character  
##  Median :128.0   Median :360.0    Mode  :character   Mode  :character  
##  Mean   :145.8   Mean   :342.4                                         
##  3rd Qu.:164.8   3rd Qu.:360.0                                         
##  Max.   :700.0   Max.   :480.0                                         
##  Loan_Status       
##  Length:614        
##  Class :character  
##  Mode  :character  
##                    
##                    
##                    
```

All columns now report zero missing values, validating data readiness for modeling.

After pre-processing and cleaning the dataset by replacing the missing values using mode and median as it is important step for model training and evaluation for accurate modelling and evaluate how well the model performs.

For the decision tree modelling for this problem statement, the main objective is to predict whether a loan application will be approved (target variable: Loan_Status). For the model evaluation, a number of performance metrics will be used that are appropriate for binary classification problems. The metrics will help us evaluate how well the model classifies approvals and rejections of loans.

Model Training

A Decision Tree classifier is selected to perform this task due to its simplicity and capability in handling both types of features: numeric and categorical. To do so, we will train our model using the pre-processed dataset and evaluate its performance on the test dataset to check the ability of the model to generalize.

Classification Evaluation Metrics:

1. Precision

Precision: the ratio of correct predictions for positive values out of all the positive predictions made. Mathematically, it is calculated as: The Formula for precision is as below:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Where:

- TP is the **True Positives**, correctly predicted loan approvals.
- FP is the **False Positives**, incorrectly predicted loan approvals.

2. Recall (Sensitivity)

Recall gives the proportion of actual positives (actual loan approvals) that were correctly predicted as positive. It is defined by the formula:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Where: - TP means the **True Positives**. - FN is the **False Negatives** (incorrect loan rejections predicted).

3. F1-Score

The F1-score is the harmonic mean of precision and recall and hence does the balancing between the two quantities. It follows from:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

4. Precision

Precision is the ratio of all correct predictions (True Positives + True Negatives) out of the total predictions. The formula is:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Where:

- TN is the **True Negatives** (correct loan rejections predicted).
- FP is the **False Positives**.

5. AUC - ROC Curve

The AUC of the ROC is a measure of the performance of the model in the area of distinguishing between the positive and negative classes. It may be expressed as follows:

AUC - Area Under the Curve

ROC - Receiver Operating Characteristic

$$\text{AUC} = \text{Area under ROC curve}$$

Training and Evaluation of Model

Let's now assess the model's power using the above metrics on the test data. An implementation in R is given below; this assumes that a model has been trained and the predictions are made:

Loading and Preparing Data and loading the necessary packages

Training and Splitting of Training data into Training and validation Data: We start by dividing the data into two:

1. Training Data: The data on which the model is trained, for instance, 50% to 90% of the complete data set.

2. Validation data : The data on which the model performance has to be evaluated. It is the unseen data by the model during training.

We will not be using the test_data for prediction, as it does not include the Loan_Status column. Instead, we will split the training data into two subsets: training data and validation data. This approach allows us to evaluate the model's performance by assessing the accuracy and precision of its predictions.

```
# loading the necessary libraries before starting with the splitting
# tidyverse for data manipulation and plotting, caret is used for the confusion matrix and evaluation o
#pROC for the performance metrics AUC-ROC curve analysis,
# ggplot2 which was used before for Plotting grapghs for visual understanding,
# pivot_longer for reshaping of the data values
library(tidyverse)
library(caret)
```

```
## Warning: package 'caret' was built under R version 4.4.2

## Loading required package: lattice

##
## Attaching package: 'caret'

## The following object is masked from 'package:purrr':
##
##     lift
```

```
library(rpart)
library(pROC)
```

```
## Warning: package 'pROC' was built under R version 4.4.2

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

##
## Attaching package: 'pROC'

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

```
library(ggplot2)
# tidyr library is used for reshaping and tidying up data.
# rpart - used for Recursive Partitioning to create the decision tree
library(rpart)
library(tidyr)

# checking the summary of the Loan_Status and Loan_ID of the train_data to split the data accordingly a

summary(train_data$Loan_Status)
```

```
##    Length     Class      Mode
##       614 character character
```

```r
summary(train_data$Loan_ID)
```

```
##    Length     Class      Mode 
##       614 character character
```

```r
# factoring the Loan_Status as it has binary yes and no and NA data and factor() helps to categorize th

train_data$Loan_Status <- factor(train_data$Loan_Status)

# Checking the summary of the Loan_Status
summary(train_data$Loan_Status)
```

```
##   N   Y 
## 192 422
```

```r
#After conversion you can see the that the levels Y and N are categorized now for better processing now


# by observing the summary of Loan_ID as it is acting like a primary key and it cannot be categorize or

summary(train_data$Loan_ID)
```

```
##    Length     Class      Mode 
##       614 character character
```

```r
#Dropping the Loan_ID column
train_data <- train_data %>% select(-Loan_ID)
```

Varying Training Data Size Since the question mentions evaluating the training data with respect to variation in its size, we will train our model on various portions of the available training data - 50%, 60%, 70%, 80%, and 90% - of the total available training data.

In each iteration, we will: Use the training subset to train the model. Score the performance on validation data - test data.

```r
# creating vectors using "training_size" the sequence of vectors of different sized from 0.5 i.e, 50% t
sizes_train_data_split <- seq(0.5, 0.9, by = 0.1)

# we are also creating
performance_metrics <- data.frame(
  Size_of_train_data = numeric(), # this dataframe contains of the size of the training data in each it
  #Performance metrics with numeric type dataframe used for storing values in each iteration of the tra
  Accuracy = numeric(),
  F1_Score = numeric(),
  Precision = numeric(),
  Recall = numeric(),
  AUC = numeric()
)

# creating a for loop to each varying spitting size
for (train_size in sizes_train_data_split) {
  #random sampling to ensure the reproducibility of the data splitting in looping
  set.seed(123)
  train_index <- sample(1:nrow(train_data), size = floor(train_size * nrow(train_data))) #this line imp
  # train_split contains the the data with train_index number of indices to be included for the split s
  train_split <- train_data[train_index, ]
  #Whereas - (minus) here for validation_split indicates that the remaining rows are split and stored i
  validation_split <- train_data[-train_index, ]

#rpart is used for the building the decision tree model and the Loan_Status is the target variable and

 model<-rpart(Loan_Status ~ .,data=train_split,method="class")

# Now making predictions on the validation_split data
 predictions<-predict(model,validation_split,type="class")

# confusion matrix of the predicted values of the validation split of the column Loan_Status
 confusion_matrix<-confusionMatrix(predictions,validation_split$Loan_Status)


# Calculating the ROC curve - AUC
 predictions_numeric<-as.numeric(predictions)-1
 roc_curve<-roc(validation_split$Loan_Status,predictions_numeric)

 # storing the performance metrics
 performance_metrics<-rbind(performance_metrics,
 data.frame(
 Size_of_train_data=train_size,
 Accuracy=confusion_matrix$overall["Accuracy"],
 F1_Score=confusion_matrix$byClass["F1"],
 Precision=confusion_matrix$byClass["Precision"],
 Recall=confusion_matrix$byClass["Recall"],
 AUC=roc_curve$auc
 )
 )
 # displaying the performance metrics of each split in the loop
 cat("\nTraining Data Size:",train_size,"\n")
 cat("Accuracy:",confusion_matrix$overall["Accuracy"],"\n")
 cat("F1Score:",confusion_matrix$byClass["F1"],"\n")
 cat("Precision:",confusion_matrix$byClass["Precision"],"\n")
 cat("Recall:",confusion_matrix$byClass["Recall"],"\n")
 cat("AUC:",roc_curve$auc,"\n")


}
```

```
## Setting levels: control = N, case = Y
```

```
## Setting direction: controls < cases
```

```
##
## Training Data Size: 0.5
## Accuracy: 0.7915309
## F1Score: 0.5616438
## Precision: 0.8913043
## Recall: 0.41
## AUC: 0.6929227

## Setting levels: control = N, case = Y
## Setting direction: controls < cases

##
## Training Data Size: 0.6
## Accuracy: 0.7276423
## F1Score: 0.5248227
## Precision: 0.5692308
## Recall: 0.4868421
## AUC: 0.6610681

## Setting levels: control = N, case = Y
## Setting direction: controls < cases

##
## Training Data Size: 0.7
## Accuracy: 0.7837838
## F1Score: 0.5454545
## Precision: 0.7272727
## Recall: 0.4363636
## AUC: 0.6835664

## Setting levels: control = N, case = Y
## Setting direction: controls < cases

##
## Training Data Size: 0.8
## Accuracy: 0.7723577
## F1Score: 0.5333333
## Precision: 0.6666667
## Recall: 0.4444444
## AUC: 0.6762452

## Setting levels: control = N, case = Y
## Setting direction: controls < cases

##
## Training Data Size: 0.9
## Accuracy: 0.8064516
## F1Score: 0.625
## Precision: 0.8333333
## Recall: 0.5
## AUC: 0.7261905
```

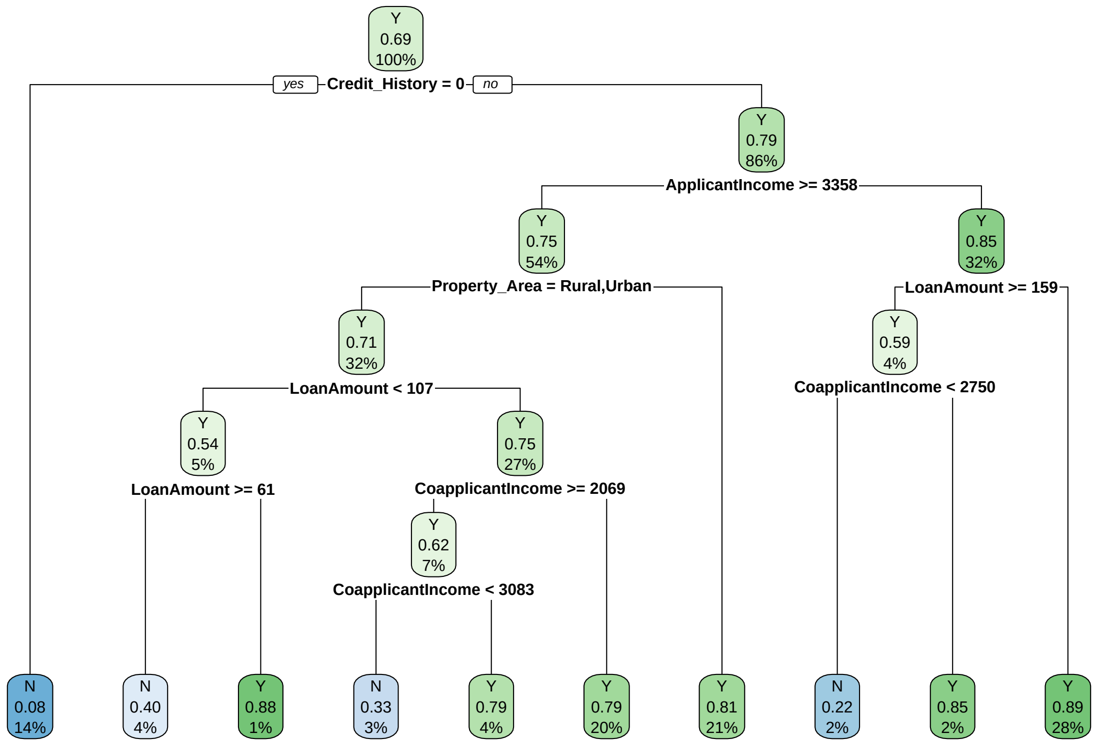
```
print(confusion_matrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  N  Y
##          N 10  2
##          Y 10 40
##
##                Accuracy : 0.8065
##                  95% CI : (0.6863, 0.8958)
##     No Information Rate : 0.6774
##     P-Value [Acc > NIR] : 0.01765
##
##                   Kappa : 0.5053
##
##  Mcnemar's Test P-Value : 0.04331
##
##             Sensitivity : 0.5000
##             Specificity : 0.9524
##          Pos Pred Value : 0.8333
##          Neg Pred Value : 0.8000
##              Prevalence : 0.3226
##          Detection Rate : 0.1613
##    Detection Prevalence : 0.1935
##       Balanced Accuracy : 0.7262
##
##        'Positive' Class : N
##
```

```r
# Loading rpart.plot package for plotting the decision tree model
library(rpart.plot)
```

```
## Warning: package 'rpart.plot' was built under R version 4.4.2
```

```r
# The visual decison tree model displayed
rpart.plot(model)
```

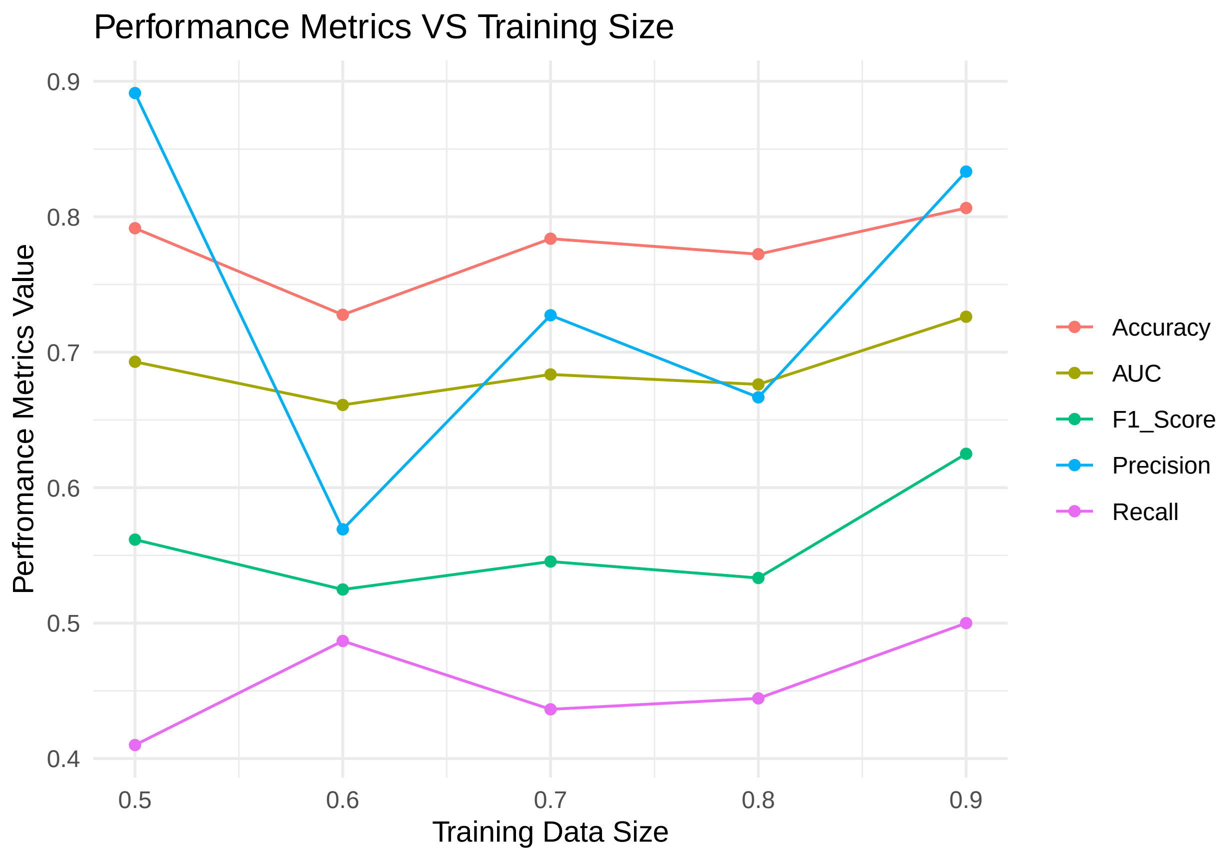


```
#Converting the AUC to numeric values for plotting
 performance_metrics$AUC<-as.numeric(performance_metrics$AUC)

# converting the values of the performance_metrics to longer format and storing in "performance_metrics_
 performance_metrics_long<-pivot_longer(performance_metrics,
 cols=-Size_of_train_data,
 names_to="Metric",
 values_to="Value")

 # checking the variables and columns of the data stored in performance_metrics_long
 str(performance_metrics_long)
```

```
## tibble [25 x 3] (S3: tbl_df/tbl/data.frame)
##  $ Size_of_train_data: num [1:25] 0.5 0.5 0.5 0.5 0.5 0.6 0.6 0.6 0.6 0.6 ...
##  $ Metric            : chr [1:25] "Accuracy" "F1_Score" "Precision" "Recall" ...
##  $ Value             : num [1:25] 0.792 0.562 0.891 0.41 0.693 ...
```

The above code snippet creates a model of the train_data and the makes predictions on the validation data and evaluates the performance of the model using the performance metrics to analyse the accuracy of the model. the best. Now below lets visualize the performance of the model in each training size with he performance metrics values.

```
 # creating a line and point plot for visualization of the variation and comparing between the performa
# X axis contains the values of the Training data size and Y axis contains the values of the Performanc
 ggplot(performance_metrics_long, aes(x=Size_of_train_data,y=Value,color=Metric,group=Metric)) +
 geom_line()+ # line plot
 geom_point() + # point plotting for each metric
 theme_minimal()+
 labs(
 title = "Performance Metrics VS Training Size",
 x="Training Data Size",
 y="Perfromance Metrics Value"
 )+
 theme(legend.title= element_blank())
```



In the above line and point plotting of the values we can observe that F1 Score and AUC does not have much deviation and also is remains consistent with the performance measures even accross different training data sizes. Therefore, it is prominent that AUC and F1 score is an ideal performance metrics to measure the performance of the model of the developed decision tree.

Analysing Model performance of F1-Score and AUC-ROC

Understanding the metrics:

F1-score: Harmonic mean and a balance of precision and recall; provides an overall measure regarding model performance. Highly useful in cases where there is a problem of class imbalance. A higher F1-score means a better model performance.

AUC-ROC Curve: It measures the capability of a model to differentiate between positive and negative classes effectively. A higher value of AUC-ROC indicates a better model performance. This measure is not affected by class imbalance, hence more robust.

Interpreting the Results: From the above plotting, based on the training data, the following can be observed:

F1-Score and AUC: Both these metrics depict very promising performances across different sizes of training data. These are more resistant to fluctuations and give a good, overall idea regarding model accuracy.

Precision and Recall: these metrics are more volatile, possibly because of the class imbalance within the dataset. Precision: Although the most commonly used metric, accuracy may be misleading in imbalanced datasets. In this case, accuracy seems also to be biased by the class distribution.

Why F1-Score and AUC are Preferred?

F1-Score: It takes into consideration both false positives and false negatives, hence suitable in cases where both types of errors are of importance. It gives the balanced measure of precision and recall. AUC-ROC Curve: It assesses the strength of a model by its ability to discriminate between positive and negative classes at different thresholds. It is unaffected by class imbalance.

Improving Model Performance:

For further improvement of the model, one can apply the following techniques:

- Hyperparameter Tuning: Hyperparameters to try out: maximum depth of the tree, minimum samples per leaf, and minimum information gain.
- Feature Engineering: Create new features or transform existing ones to improve model accuracy.
- Ensemble Methods: Combine several decision trees to get better performance and to reduce overfitting.
- Cross-Validation: The performance of any model can be better estimated using cross-validation.

By considering all these carefully, you can build more robust and accurate loan approval prediction models.

4. Exploring the Performance of the Decision Tree model varies on both the training data and the validation data as we vary the Hyperparameters.

We will be performing the following by implementation below:

- Performance of your model varing on both training data and the validation data
- hyperparameter tuning
- Visualize the performance metrics and hyperparameter settings
- predict the Loan_Status values in the test_data using the modelling with hyper parameter tuning.

Exploration by how the performance of your model changes across a range of different hyperparameter settings.

We are going to do some exploration, seeing how performance changes upon tuning some hyperparameters: max_depth, min_split, min_bucket.

We will also test said hyperparameters by observing the change they make to the model's F1 score and AUC on both training and validation data.

Hyperparameter Tuning for Decision Tree

Adjusting the decision tree's hyperparameters and observe how the performance metrics change with different configurations.

This will help us understand whether increasing the complexity (e.g., deeper trees) leads to overfitting or underfitting, and which settings yield the best performance on both the training and validation datasets.

We'll implement a loop to iterate over different hyperparameter combinations and assess model performance for each.

Model Training: In the loop, we will be training the decision tree model, using rpart() for each hyperparameter combination.

Model Evaluation: Having trained the model, we will make predictions on the validation data and calculate the performance metrics- F1 score and AUC by calling F1_Score(), auc()and roc().

Performance Storage: The performance metrics are stored in a data frame, results.

Visualization: we display the table using knitr library.

First we will begin with the data splitting i.e, splitting the training data into training data and validation data.

```r
# Loading the libraries - rpart for the modeling recursive partitioning, caret is used here for trainin
library(rpart)
library(caret)
library(pROC)
library(MLmetrics)
```

```
## Warning: package 'MLmetrics' was built under R version 4.4.2
```

```
##
## Attaching package: 'MLmetrics'
```

```
## The following objects are masked from 'package:caret':
##
##     MAE, RMSE
```

```
## The following object is masked from 'package:base':
##
##     Recall
```

```r
#random sampling to ensure the reproducibility of the data splitting in looping
set.seed(123)

# Again we will be splitting the training data into 2 subsets which is split this time with in proporti
# we are splitting the data in accordance with the train_index which shows the indixes to include in tr

train_index <- createDataPartition(train_data$Loan_Status, p = 0.8, list = FALSE)
# train_set contains the indices rounded of to the closest integer from thr above
train_set <- train_data[train_index, ]
# validation_set contains the data of the remaining indices
validation_set <- train_data[-train_index, ]
```

Explanation of the Hyperparameters we will be using:

1. maxdepth:

This is the maximum depth of each decision tree. Effect: Controls how deep the tree can grow. It prevents the tree from overfitting with a small number of splits. This is a very important tuning parameter, and a large value may allow the tree to create splits resulting in overfitting. Value ranges from 5 to 15.

2. minsplit:

Minimum number of observations required to split an internal node. Effect: This controls the complexity of the tree. The higher this value is, the fewer the splits will be, potentially leading to a simpler tree. In contrast, with low values, overfitting is possible because the tree will split with fewer data points. Values ranges from 10 to 20

3. minbucket:

Minimum number of required observations in any terminal node (leaf). Effect: It prevents overfitting, just like minsplit. A higher value will make the tree more conservative, making sure each leaf node contains at least more data points, whereas a smaller value could lead to more trees having small leaf nodes. Values ranges from 5 to 10 in this case.

Below we will be creating a hyperparameter grid with the conditions or settings of the above hyperparameters discussed for modeling.

```r
# we can now create a hyperparameter grid
hyper_grid <- list(
  #moderate conditions for splitting and forming leaf nodes
  list(maxdepth = 5, minsplit = 10, minbucket = 5),
  #deeper tree with equal conditions for splitting and leaf nodes
  list(maxdepth = 7, minsplit = 10, minbucket = 5),
  #More points of data will be needed for it to split and create leaf nodes.
  list(maxdepth = 10, minsplit = 20, minbucket = 10),
  #deep tree with moderate conditions for splitting and leaf formation.
  list(maxdepth = 10, minsplit = 10, minbucket = 5),
  #very deep tree, with the strictest conditions on the split and formation of leaf nodes
  list(maxdepth = 15, minsplit = 20, minbucket = 10)
)
```

Now that hyperparameter grid is created lets move ahead with the performance of the model with the modeling using the above conditional settings of the hyperparameters.

```r
# creating the empty dataframe- "results" to store the values of performance metric
# in each iteration of the hyperparameter tuning
results <- data.frame(
  maxdepth = integer(),
  minsplit = integer(),
  minbucket = integer(),
  F1_Score = numeric(),
  AUC = numeric()
)

# looping and iterating through each condition we defined above of maxdepth, minsplit, minbucket
for (params in hyper_grid) {

  # Training the model using the hyperparameters in each iteration
  # rpart is used to create the model and Loan_Status as the target variable
  model <- rpart(Loan_Status ~ ., data = train_set,
                method = "class",
                # control parameter is used to specify the hyperparameters- settings
                # that control how the decision tree is built.
                control = rpart.control(
                  maxdepth = params$maxdepth,
                  minsplit = params$minsplit,
                  minbucket = params$minbucket
                ))

  # After training the model let us predict the values in the validation
  # predict() makes predictions based on a trained model on the validation data set into class labels
  predictions <- predict(model, validation_set, type = "class")

  # for returning the probabilities for each class
  pred_prob <- predict(model, validation_set, type = "prob")[, 2]

  # Calculating the F1-score of each combination of the hyperparameter tuning
  # and F1_Score() is a function from the MLmetrics which is used here
  f1 <- F1_Score(y_true = validation_set$Loan_Status, y_pred = predictions)

  # similarly calculating auc value
  roc_curve <- roc(validation_set$Loan_Status, pred_prob)
  auc_value <- auc(roc_curve)

  # storing the results for each iteration in the empty dataframe- results which was created initially
  results <- rbind(results, data.frame(
    maxdepth = params$maxdepth,
    minsplit = params$minsplit,
    minbucket = params$minbucket,
    F1_Score = f1,
    AUC = auc_value
  ))
}
```

```
## Setting levels: control = N, case = Y
```

```
## Setting direction: controls < cases
## Setting levels: control = N, case = Y

## Setting direction: controls < cases

## Setting levels: control = N, case = Y

## Setting direction: controls < cases

## Setting levels: control = N, case = Y

## Setting direction: controls < cases

## Setting levels: control = N, case = Y

## Setting direction: controls < cases
```

```r
print(results)
```

```
##   maxdepth minsplit minbucket  F1_Score       AUC
## 1        5       10         5 0.7000000 0.7703634
## 2        7       10         5 0.6562500 0.7908835
## 3       10       20        10 0.7000000 0.7703634
## 4       10       10         5 0.6470588 0.7949561
## 5       15       20        10 0.7000000 0.7703634
```

Let us show more visually the in the form of a table the Values of the performance metrics and the values of maxdepth, minsplit, minbucket hyperparameters.

```r
# library knitr is used for creating a table for better visual representation of the values of the perf
library(knitr)

# kable()- display the table with formatting
kable(results,
      caption = "Performance Metrics (F1 Score and AUC) for Different Hyperparameters",
      col.names = c("Max Depth", "Min Split", "Min Bucket", "F1 Score", "AUC"),
      format = "pipe",
      align = "ccccc")
```

Table 2: Performance Metrics (F1 Score and AUC) for Different Hyperparameters

Max Depth	Min Split	Min Bucket	F1 Score	AUC
5	10	5	0.7000000	0.7703634
7	10	5	0.6562500	0.7908835
10	20	10	0.7000000	0.7703634
10	10	5	0.6470588	0.7949561
15	20	10	0.7000000	0.7703634

In the table we can see that AUC has the highest value of 0.7908835 with the hyperparameter setting of maxdepth = 7, min split = 10 and min bucket = 5. and there is not much fluctuation of the performance metric score of F1-Score.

5. Choosing a Hyper-parameter and Reporting the performance based on the test data.

We will be testing and evaluating the performance of our model by using hyperparameters to predict the Loan_Status in the test_data with a decision tree model.

But before predicting, one important preprocessing that has to be performed is handling the missing values present in the test_data - exactly as we did for the train_data-so that the predictions can be made accurately and effectively. Alternatively, we could have ignored the rows containing missing values; however, for this analysis, we deal with missing values. By doing so, we can make intelligent predictions concerning approval or rejection of more loans. Note that this is just an example for illustration purposes to practice data analysis and statistical computing in R programming.

```r
#Similarly to the data handling and preprocessing done for train_data let us perform preprocessing in t
# Handling Missing Values for test_data (same as for train_data)
# Print the missing values summary for test_data
missing_values_test_data <- colSums(is.na(test_data) | test_data == "")
print(missing_values_test_data)
```

```
##           Loan_ID            Gender           Married        Dependents
##                 0                11                 0                10
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome
##                 0                23                 0                 0
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area
##                 5                 6                29                 0
```

```r
# Replacing the missing values in the "Loan Amount" column with median (numerical data)
# Median is used because of the differences in loan amounts
test_data$LoanAmount[is.na(test_data$LoanAmount)] <- median(test_data$LoanAmount, na.rm = TRUE)

# Checking the missing values again after the replacement
missing_values_test_data <- colSums(is.na(test_data) | test_data == "")
print(missing_values_test_data)
```

```
##           Loan_ID            Gender           Married        Dependents
##                 0                11                 0                10
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome
##                 0                23                 0                 0
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area
##                 0                 6                29                 0
```

```r
# Replacing missing values in Loan_Amount_Term by using the mode
mode_loan_term_test_data <- as.numeric(names(sort(table(test_data$Loan_Amount_Term), decreasing = TRUE)
test_data$Loan_Amount_Term[is.na(test_data$Loan_Amount_Term)] <- mode_loan_term_test_data

# Checking the replacement
head(test_data$Loan_Amount_Term, 40)
```

```
##  [1] 360 360 360 360 360 360 360 360 240 360 360 360 180 360 360 360 360 360 360
## [20] 180 360 180 360 360 360 360 360 360 360 180 360 360 360 360 360 360 180 360
## [39] 360 360
```

```r
# Displaying a summary of the Loan_Amount_Term column after replacement
summary(test_data$Loan_Amount_Term)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     6.0   360.0   360.0   342.8   360.0   480.0
```

```r
# Replacing missing values in 'Gender' column with mode
# First, replace empty strings with NA in the Gender column
test_data$Gender[test_data$Gender == ""] <- NA

# Now, replace missing Gender values with the mode of the column
mode_gender_test_data <- names(sort(table(test_data$Gender), decreasing = TRUE)[1])
test_data$Gender[is.na(test_data$Gender)] <- mode_gender_test_data

# Check missing values after replacement
missing_values_test_data <- colSums(is.na(test_data) | test_data == "")
print(missing_values_test_data)
```

```
##           Loan_ID            Gender           Married        Dependents
##                 0                 0                 0                10
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome
##                 0                23                 0                 0
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area
##                 0                 0                29                 0
```

```r
# Checking the first 20 rows of the Gender column to confirm the replacement
head(test_data$Gender, 20)
```

```
##  [1] "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Female" "Male"
##  [9] "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Female" "Male"
## [17] "Male"   "Male"   "Male"   "Male"
```

```r
# Similarly, replace missing values in the 'Married' column
test_data$Married[test_data$Married == ""] <- NA
mode_married_test_data <- names(sort(table(test_data$Married), decreasing = TRUE)[1])
test_data$Married[is.na(test_data$Married)] <- mode_married_test_data

# Check for missing values again after the replacement
missing_values_test_data <- colSums(is.na(test_data) | test_data == "")
print(missing_values_test_data)
```

```
##           Loan_ID            Gender           Married        Dependents
##                 0                 0                 0                10
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome
##                 0                23                 0                 0
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area
##                 0                 0                29                 0
```

```r
# Checking the first 20 rows of the Married column to confirm replacement
head(test_data$Married, 20)
```

```
##  [1] "Yes" "Yes" "Yes" "Yes" "No"  "Yes" "No"  "Yes" "Yes" "No"  "No"  "Yes"
## [13] "No"  "Yes" "No"  "No"  "Yes" "Yes" "Yes" "No"
```

```
# For the 'Dependents' column, replacing missing values with mode
test_data$Dependents[test_data$Dependents == ""] <- NA
mode_dependents_test_data <- names(sort(table(test_data$Dependents), decreasing = TRUE)[1])
test_data$Dependents[is.na(test_data$Dependents)] <- mode_dependents_test_data

# Checking the first 20 rows of the Dependents column
head(test_data$Dependents, 20)
```

```
##  [1] "0"  "1"  "2"  "2"  "0"  "0"  "1"  "2"  "2"  "0"  "0"  "1"  "3+" "2"  "0"
## [16] "1"  "2"  "3+" "0"  "0"
```

```
# Similarly for 'Self_Employed' column, replace missing values with mode
test_data$Self_Employed[test_data$Self_Employed == ""] <- NA
mode_self_employed_test_data <- names(sort(table(test_data$Self_Employed), decreasing = TRUE)[1])
test_data$Self_Employed[is.na(test_data$Self_Employed)] <- mode_self_employed_test_data

# Checking the first 21 rows of the Self_Employed column
head(test_data$Self_Employed, 21)
```

```
##  [1] "No"  "No"  "No"  "No"  "No"  "Yes" "No"  "No"  "No"  "No"  "No"  "No"
## [13] "No"  "No"  "No"  "No"  "No"  "No"  "No"  "No"  "No"
```

```
# For the 'Credit_History' column, replace missing values with mode
test_data$Credit_History[test_data$Credit_History == ""] <- NA
mode_credit_history_test_data <- names(sort(table(test_data$Credit_History), decreasing = TRUE)[1])
test_data$Credit_History[is.na(test_data$Credit_History)] <- mode_credit_history_test_data

# Checking the first 26 rows of the Credit_History column
head(test_data$Credit_History, 26)
```

```
##  [1] "1" "1" "1" "1" "1" "1" "1" "0" "1" "1" "1" "1" "1" "0" "1" "1" "1" "1" "1"
## [20] "1" "1" "1" "1" "1" "1" "0"
```

```
# Checking if there are any missing values left after all replacements
missing_values_test_data <- colSums(is.na(test_data) | test_data == "")
print(missing_values_test_data)
```

```
##           Loan_ID            Gender           Married        Dependents
##                 0                 0                 0                 0
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome
##                 0                 0                 0                 0
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area
##                 0                 0                 0                 0
```

```
# Displaying a summary of the test_data to ensure all missing values have been handled
summary(test_data)
```

```
##    Loan_ID             Gender            Married           Dependents
##  Length:367         Length:367         Length:367         Length:367
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##   Education         Self_Employed      ApplicantIncome CoapplicantIncome
##  Length:367         Length:367         Min.   :    0   Min.   :    0
##  Class :character   Class :character   1st Qu.: 2864   1st Qu.:    0
##  Mode  :character   Mode  :character   Median : 3786   Median : 1025
##                                        Mean   : 4806   Mean   : 1570
##                                        3rd Qu.: 5060   3rd Qu.: 2430
##                                        Max.   :72529   Max.   :24000
##    LoanAmount    Loan_Amount_Term Credit_History     Property_Area
##  Min.   : 28.0   Min.   :  6.0    Length:367         Length:367
##  1st Qu.:101.0   1st Qu.:360.0    Class :character   Class :character
##  Median :125.0   Median :360.0    Mode  :character   Mode  :character
##  Mean   :136.0   Mean   :342.8
##  3rd Qu.:157.5   3rd Qu.:360.0
##  Max.   :550.0   Max.   :480.0
```

After Data Handling and pre-processing of test_data, we will be duplicating the test_data as test_data1 as we will be using the test_data again further for other modelling with hyperparameter tuning and evaluation techniques.

```
# Duplicating the test_data as "test_data1"
test_data1 <- test_data

# Here we are utilizing the combination which was best fitting from the previous hyperparameter tuning
# the best modelling combination of the hyperparameteer we perfored before was #deeper tree with equal
# maxdepth = 7, minsplit = 10, minbucket = 5  with the performance of F1 - Score as  0.6562500 and AUC
# maxdepth = 10, minsplit = 10, minbucket = 5  with the performance of F1 - Score as  0.6562500 and AUC

#results dataframe contains the values of the performance metrices values and the hyperparameter settin
# Checking if 'results' exists and contains the best model information

# We are choosing AUC as the performance metric to evaluate our model as it has the best value of perfo
if (exists("results") && "AUC" %in% colnames(results)) {
  # Finding the best model based on highest AUC
  best_model_auc <- results[which.max(results$AUC), ]

  # displaying the best hyperparameters based on AUC
  print("Best hyperparameters based on AUC:")
  print(best_model_auc)

  # Now, let us use the best hyperparameters to perform the modeling
  best_maxdepth <- best_model_auc$maxdepth
  best_minsplit <- best_model_auc$minsplit
  best_minbucket <- best_model_auc$minbucket

  # creating a data frame best_combination_model which will hold the values of the best hyperparameters
  # using rpart to derive a decision tree using the parameters
  best_combination_model <- rpart(Loan_Status ~ ., data = train_data,
                        method = "class",
                        control = rpart.control(
                          maxdepth = best_maxdepth,
                          minsplit = best_minsplit,
                          minbucket = best_minbucket
                        ))

  # We are ensuring here as an additional step to check if all the factors are in the same leveling
  factor_columns <- sapply(train_data, is.factor)
  for (col_name in names(factor_columns)[factor_columns]) {
    if (col_name %in% names(test_data1)) {
      levels(test_data1[[col_name]]) <- levels(train_data[[col_name]])
    }
  }

  # Now predicting on the test_data1 using the best_combination_model and class labelling
  test_predictions1 <- predict(best_combination_model, test_data1, type = "class")

  # We are performing an additional step here by saving it on a csv file so we can trace back to the cs
  test_results1 <- data.frame(Loan_ID = test_data1$Loan_ID, Loan_Status_Predicted = test_predictions1)
  write.csv(test_results1, "test_predictions1.csv", row.names = FALSE)

  # Optionally, display the first few predictions
  head(test_results1)
} else {
  print("Error: 'results' object does not exist or is missing necessary columns.")
}
```

```
## [1] "Best hyperparameters based on AUC:"
##   maxdepth minsplit minbucket  F1_Score       AUC
## 4       10       10         5 0.6470588 0.7949561
```

```
##    Loan_ID Loan_Status_Predicted
## 1 LP001015                     Y
## 2 LP001022                     Y
## 3 LP001031                     Y
## 4 LP001035                     Y
## 5 LP001051                     Y
## 6 LP001054                     Y
```

Using Max_depth hyperparameter

We will be creating another duplicate test_data2 from test_data now to implement only the max_depth hyperparameter here:

```
# Duplicating the test_data2 from test_data as we have already treated the test_data with missing value
test_data2 <- test_data

# Checking before processing with the next step
missing_values_test2 <- colSums(is.na(test_data2) | test_data2 == "")
print("Missing values before imputation:")
```

```
## [1] "Missing values before imputation:"
```

```
print(missing_values_test2)
```

```
##           Loan_ID            Gender           Married        Dependents
##                 0                 0                 0                 0
##         Education     Self_Employed   ApplicantIncome CoapplicantIncome
##                 0                 0                 0                 0
##        LoanAmount  Loan_Amount_Term    Credit_History     Property_Area
##                 0                 0                 0                 0
```

We will be implementing the best_maxdepth hyperparameter from the above to use to train a now model and predict on the test_data to later compare between different models and make a more effective decision.

```
# Ensuring and aligning factors are leveled between train_data and test_data2 data sets to proceed furt
factor_columns <- sapply(train_data, is.factor)
for (col_name in names(factor_columns)[factor_columns]) {
  if (col_name %in% names(test_data2)) {
    levels(test_data2[[col_name]]) <- levels(train_data[[col_name]])
  }
}

# Creating a decision tree model with only max_depth hyperparameter and rpart function
# Also we are choosing the best hyperparameter value from the previous code chunk
final_model_test2 <- rpart(Loan_Status ~ ., data = train_data,
                           method = "class",
                           control = rpart.control(maxdepth = best_maxdepth))

# predicting the values for the test_data2 and class labelling
test_predictions2 <- predict(final_model_test2, test_data2, type = "class")

# Factoring the resulting predictions before displaying
test_predictions2 <- factor(test_predictions2, levels = c("N", "Y"))

# Now displaying with the loan_ID along with the prediction made in the test_data
test_predictions_2 <- data.frame(Loan_ID = test_data2$Loan_ID, Loan_Status = test_predictions2)

print(best_maxdepth)
```

```
## [1] 10
```

```
# displaying the first few rows of test_predictions_2
head(test_predictions_2, 10)
```

```
##     Loan_ID Loan_Status
## 1  LP001015           Y
## 2  LP001022           Y
## 3  LP001031           Y
## 4  LP001035           Y
## 5  LP001051           Y
## 6  LP001054           Y
## 7  LP001055           Y
## 8  LP001056          N
## 9  LP001059          Y
## 10 LP001067          Y
```

Understanding Cross-Validation

What is Cross - Validation and use in decsion Tree:

Cross-Validation in Decision Tree Modeling It's a technique in machine learning and statistical modeling for the evaluation of a model according to data partitioning into multiple subsets. One trains a model on some subsets and evaluates it on others. This provides good guarantees of generalizing to unseen data, not just fitting to the data that it has been trained on. Cross-validation, therefore, forms an important technique for these decision tree models, which are prone to overfitting.

Working and Types of Cross Validation:

1. K-Fold Cross- Validation: The most common type of cross-validation is K-fold CV, justb like splitting of the tree this process splitting the dataset into K equal - sized "folds". In each fold, the model is trained on the K - 1 folds and tested on the remaining one. This is repeated K times, each fold being used once as a test set. The resulting metric is then averaged out.
2. Leave-One-Out Cross-Validation (LOOCV): In LOOCV each data point is considered as a K fold and each data point is used once to predict on a test data and the model will be training on the remaining points. It is computationally very expensive but can be really useful if one has small datasets.
3. Stratified Cross-Validation: For problems involving classification, the variant of K-fold CV, stratified K-fold ensures that each fold retains the same distribution of target labels as that of the original dataset. This becomes important when dealing with imbalanced classes.

Importance of Cross-Validation for Decision Trees

Decision trees can overfit the data, especially when the tree is deep and learns to capture noise in the training data rather than generalizable patterns. Overfitting leads to poor performance on unseen data, which is why cross-validation is crucial.

1. Avoiding Overfitting: Decision trees, while learning, go deeper and create very complex models that give good performance on the training data but do poorly on the test data. Cross-validation helps in assessing the performance of the model using different splits, hence giving a more reliable estimate of its performance in real-world applications.
2. Model Selection: Cross-validation can also be used to do model selection. Suppose one is doing some tuning of hyperparameters-e.g., the maximum depth of a decision tree. Cross-validation will help pick a setting of the parameters that should generalize best.
3. Better Performance Estimates: Cross-validation provides a more robust and less biased estimate of the model's performance by testing the model on multiple subsets compared to a single train-test split.

Why Use Cross-Validation in Decision Tree Modeling?

- Accuracy Estimation: Cross-validation measures model accuracy or robustness with high reliability because the model has been tested on different data splits. This can help understand what will happen when the decision tree performs in various ways on subsets of the data.

- Model Tuning: Some important hyperparameters in decision trees include depth, minimum samples for a split, and minimum samples for a leaf node. Cross-validation provides an objective way of evaluating these hyperparameters for the best configuration that will help to avoid overfitting.
- Improved Generalization: Cross-validation prevents a model from becoming overly fit to the specifics of one subset of data. Rather, it encourages generalization by testing the model on multiple unseen subsets.
- Handling Small Datasets: In small data situations, cross-validation maximizes the available training data on each point since each is used in turn as part of the training set when it is being tested on other folds.

So now we will be duplicating the tes_data into test_data3 to predict eitht K-Fold Cross-Validation and check the performance of the modeling using the F1-Score, Accuracy, and AUC. The reason for doing this is to keep test_data in its virgin state; it will be used later for prediction once the model is trained. This is common practice when working with a machine learning workflow to prevent data leakage and to keep the test data segregated from the training process.

```r
# Duplicating the test_data3 of the test_data
test_data3 <- test_data
```

We are modeling a decision tree below that trains and evaluates a decision tree model with 10-fold cross-validation in order to find an optimal value of the complexity parameter, cp, for pruning.

```r
#trainControl() is used for the cross-validation setting or conditions here.
# train control for 10-fold cross-validation
Cross_Validation_model <- trainControl(method = "cv", number = 10, savePredictions = "all")

# Setting up tuning grid for 'cp' for pruning using the rpart for modeling
# tuning complexity parameter - cp by sequencing it by 0.01 to 0.1 by incrementing by 0.01
tune_grid <- expand.grid(cp = seq(0.01, 0.1, by = 0.01))

# training the 10-fold K cv decsion tree by using rpart and tuning my the cp tune_grid
cv_model <- train(Loan_Status ~ ., data = train_data, method = "rpart",
                  trControl = Cross_Validation_model, tuneGrid = tune_grid)

# Displaying the best Tuning parameter value
best_cp <- cv_model$bestTune$cp
cat("Best CP Value: ", best_cp, "\n")
```

```
## Best CP Value:  0.1
```

```r
best_model <- cv_model$finalModel

# Factoring and leveling the data in train_data and test_data3 for better display of the values
factor_columns <- sapply(train_data, is.factor)
for (col_name in names(factor_columns)[factor_columns]) {
  if (col_name %in% names(test_data3)) {
    levels(test_data3[[col_name]]) <- levels(train_data[[col_name]])
  }
}

# final model - "final_model_test3" with the best cp as an hyperparameter for more accurate tuning
```

```
final_model_test3 <- rpart(Loan_Status ~ ., data = train_data,
                           method = "class",
                           control = rpart.control(cp = best_cp))

# predicting the Loan_Status for test_data3 using the final_model_test3
test_predictions3 <- predict(final_model_test3, test_data3, type = "class")

# factoring or leveling the predictions made so we can display or visualize later
test_predictions3 <- factor(test_predictions3, levels = c("N", "Y"))

# We are adding the Loan_ID with the predictiosn made and storing it in dataframe test_predictions_3
test_predictions_3 <- data.frame(Loan_ID = test_data3$Loan_ID, Loan_Status = test_predictions3)

# displaying the test_predictions_3 predicted result
head(test_predictions_3, 10)
```

```
##     Loan_ID Loan_Status
## 1  LP001015           Y
## 2  LP001022           Y
## 3  LP001031           Y
## 4  LP001035           Y
## 5  LP001051           Y
## 6  LP001054           Y
## 7  LP001055           Y
## 8  LP001056           N
## 9  LP001059           Y
## 10 LP001067           Y
```

Now, We can compare the models predictions on test_data with best hyperparameters - max_depth, min_split, min_bucket and hyperparameter with best max_depth with highest AUC and then 10- K folds cross-validation. We can compare the performances metrics of the models to compare the predictions or results accurary and precision.

```
# We are ensuring for more clarity the factoring of the Loan_Status in test_results1, test_predictions_
test_results1 <- factor(test_results1$Loan_Status, levels = c("N", "Y")) # predictions made based on th
test_predictions_2 <- factor(test_predictions_2$Loan_Status, levels = c("N", "Y")) # predictions made u
test_predictions_3 <- factor(test_predictions_3$Loan_Status, levels = c("N", "Y")) # predictiosn made u

# Loading the libraries , caret and pROC for confusion matric to calculate F1score and AUC calculation
library(caret)
library(pROC)

# Comparing of test_predictions_3 vs test_results1 in terms of Accuracy, F1 score and AUC
conf_matrix_3_vs_1 <- confusionMatrix(test_predictions_3, test_results1)
accuracy_3_vs_1 <- conf_matrix_3_vs_1$overall["Accuracy"]
f1_3_vs_1 <- conf_matrix_3_vs_1$byClass["F1"]
roc_3_vs_1 <- roc(test_results1, as.numeric(test_predictions_3), levels = c("N", "Y"))
```

```
## Setting direction: controls < cases
```

```r
auc_3_vs_1 <- auc(roc_3_vs_1)

# Comparing of test_predictions_3 vs test_predictions_2in terms of Accuracy, F1 score and AUC
conf_matrix_3_vs_2 <- confusionMatrix(test_predictions_3, test_predictions_2)
accuracy_3_vs_2 <- conf_matrix_3_vs_2$overall["Accuracy"]
f1_3_vs_2 <- conf_matrix_3_vs_2$byClass["F1"]
roc_3_vs_2 <- roc(test_predictions_2, as.numeric(test_predictions_3), levels = c("N", "Y"))
```

```
## Setting direction: controls < cases
```

```r
auc_3_vs_2 <- auc(roc_3_vs_2)

# Comparing of test_predictions_1 vs test_predictions_2 in terms of Accuracy, F1 score and AUC
conf_matrix_1_vs_2 <- confusionMatrix(test_results1, test_predictions_2)
accuracy_1_vs_2 <- conf_matrix_1_vs_2$overall["Accuracy"]
f1_1_vs_2 <- conf_matrix_1_vs_2$byClass["F1"]
roc_1_vs_2 <- roc(test_predictions_2, as.numeric(test_results1), levels = c("N", "Y"))
```

```
## Setting direction: controls < cases
```

```r
auc_1_vs_2 <- auc(roc_1_vs_2)

# creating a data frame comparison_results to present the comparison perfromance metrics
comparison_results <- data.frame(
  Metric = c("Accuracy", "F1 Score", "AUC"),
  `Test Predictions 3 vs Test Predictions 1` = c(accuracy_3_vs_1, f1_3_vs_1, auc_3_vs_1),
  `Test Predictions 3 vs Test Predictions 2` = c(accuracy_3_vs_2, f1_3_vs_2, auc_3_vs_2),
  `Test Predictions 1 vs Test Predictions 2` = c(accuracy_1_vs_2, f1_1_vs_2, auc_1_vs_2)
)

# displaying the comparison_results
print(comparison_results)
```

```
##            Metric Test.Predictions.3.vs.Test.Predictions.1
## Accuracy Accuracy                                0.9891008
## F1       F1 Score                                0.9672131
##               AUC                                0.9682540
##          Test.Predictions.3.vs.Test.Predictions.2
## Accuracy                                        1
## F1                                              1
##                                                 1
##          Test.Predictions.1.vs.Test.Predictions.2
## Accuracy                                0.9891008
## F1                                      0.9672131
##                                         0.9935065
```

Let us further use knitr library to see the clear understanding of the comparison of the performance in the models:

```r
# Using knitr to format the values into a clearer table
library(knitr)

# kable() to display the comparison results in the clear table
kable(comparison_results,
      col.names = c("Metric", "Test Predictions 3 vs Test Predictions 1", "Test Predictions 3 vs Test Pr
      caption = "Comparison of Model Performance (Test Predictions 3 vs Test Predictions 1, 2 and 3)",
      align = c("l", "c", "c", "c"))
```

Table 3: Comparison of Model Performance (Test Predictions 3 vs Test Predictions 1, 2 and 3)

	Metric	Test Predictions 3 vs Test Predictions 1	Test Predictions 3 vs Test Predictions 2	Test Predictions 1 vs Test Predictions 2
Accuracy	Accuracy	0.9891008	1	0.9891008
F1	F1 Score	0.9672131	1	0.9672131
	AUC	0.9682540	1	0.9935065

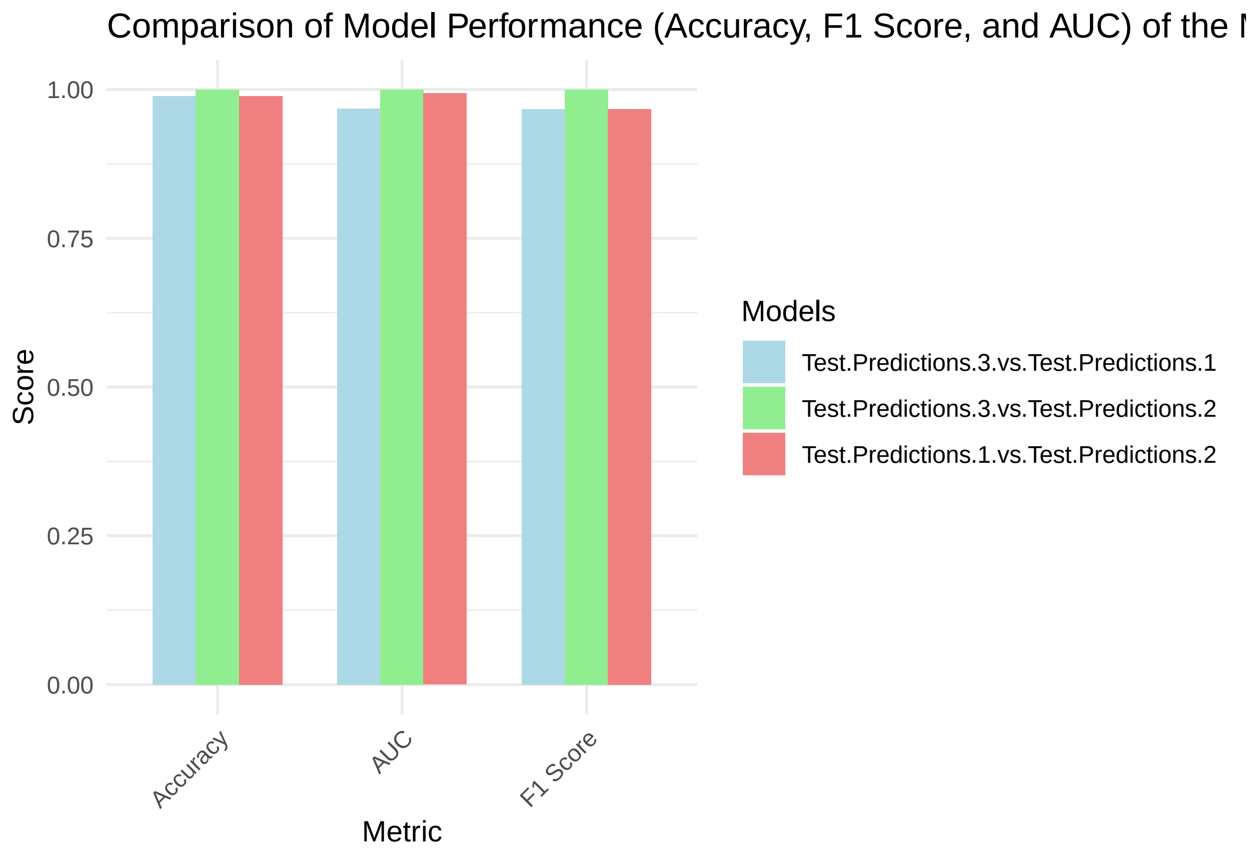
We can observe in the data above that TestPredictions3 vs TestPredictions2 i.e. the Predictions made by the cross-validation k fold and the best maxdepth value based on the higher AUC value, the comparison results with the highest prediction Accuracy, F1 Score and AUC with 1, by this analysis we can conclude by saying this hyperparameteer tuning using cv and max_depth is giving the best results with accuracy and precision.

Likewise, when comparing the TestPredictions3 vs TestPredictions1 which is predictions from cv k fold and the best max_depth, min_split and min_bucket based on the higher AUC value is the also giving the most highest value of performance which is higher than 0.96 for all Accuracy, F1-Score ad AUC performance metrices.

Similarly, Comparison between the best max_depth, min_split and min_bucket based on the higher AUC value and the predictions made by model using best max_depth has the values of performance metrics higher than 0.96.

Let us also visualize using the bar plotting of these comparisons as below:

```r
#loading the necessary packages for plotting s bar graph
library(ggplot2)
library(reshape2)

# shaping the comparison_results for bar plot
comparison_results_melted <- melt(comparison_results, id.vars = "Metric")
# ggplot to make a bar plot of comparison_results_melted
ggplot(comparison_results_melted, aes(x = Metric, y = value, fill = variable)) +
geom_bar(stat = "identity", position = "dodge", width = 0.7) +
labs(title = "Comparison of Model Performance (Accuracy, F1 Score, and AUC) of the Models",
x = "Metric",
y = "Score",
fill = "Models") +
scale_fill_manual(values = c("lightblue", "lightgreen", "lightcoral")) +
theme_minimal() +
theme(axis.text.x = element_text(angle = 45, hjust = 1)) # rotating X labels for better readability
```



6. Exploring further on Classification and Regression Trees (CART) and Random Forests Supervised Learning Methods

CART (Classification and Regression Trees)

Decision Trees (CART)

CART is one of the main algorithms in machine learning. It can be used for classification and regression problems. The CART represents a binary tree: containing internal nodes, each one represents a decision on the value of an input feature and the leaf nodes representing the prediction. Each leaf will house one predicted class in the case of classification, and one predicted value in the case of regression. Each node will have exactly 2 leaves as it is a binary tree.

How CART works:

For each node it takes the best feature .

Recursion: this procedure is repeated for every subgroup of data.

Stopping Criterion: Recursive splitting is terminated based on a stopping criterion, such as a maximum tree depth or a minimum sample size at a leaf.

Limitations of CART:

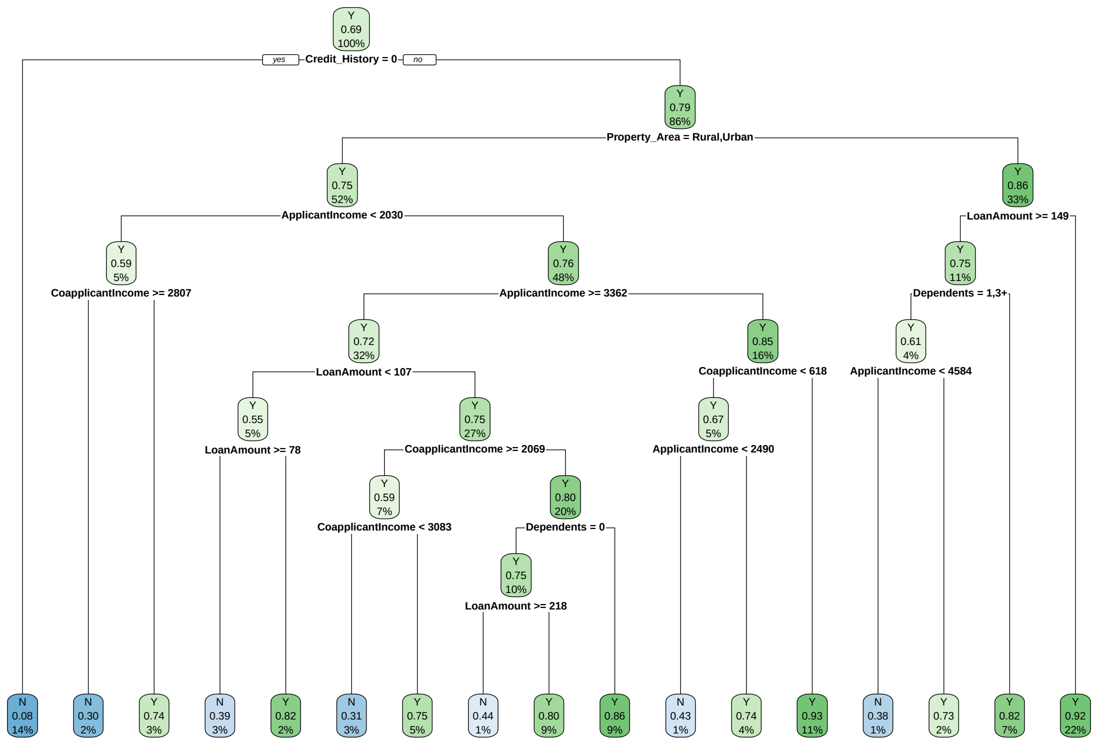
Overfitting: The major disadvantages of decision trees are overfitting-especially in case the tree grows too deep-and fitting noise rather than general patterns in the training data.

Bias towards Features with More Levels: CART tends to favor features with more categories or continuous features over categorical features with fewer categories, which makes biased splits.

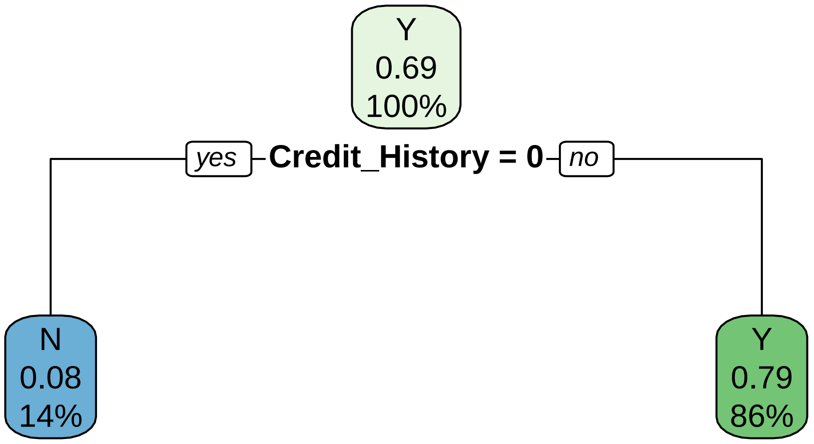
Limited Interactions: Unlike some models, such as linear models or neural networks, decision trees might miss complex interactions between features.

Implementing CART

```r
# Loading necessary libraries rpart for the decision tree structure implementation, caret for the confu
#pROC for the performance metrics AUC-ROC curve analysis,
#knitr for the table formatting
# rpart.plot for plotting the graph
library(rpart)
library(caret)
library(pROC)
library(knitr)
library(rpart.plot)

# Training the CART model on the training data set with the cp as 0.001 and class labelling
cart_model <- rpart(Loan_Status ~ ., data = train_data, method = "class", cp = 0.001)

# visualizing the cart model using rpart.plot
rpart.plot(cart_model)
```



```r
# Pruning the tree to avoid overfitting and more accurare predictions on the test data.

# finding the complexity parameter - cp that minimizes cross-validation error
optimal_cp <- cart_model$cptable[which.min(cart_model$cptable[,"xerror"]), "CP"]

# Pruning the tree here and creating a pruned cart model using the optimal complexity parameter.
pruned_cart_model <- prune(cart_model, cp = optimal_cp)

# Visualizing the pruned tree
rpart.plot(pruned_cart_model)
```



```r
#duplicating the test_data into test_data_cart
test_data_cart <- test_data

# we are now making the predictions on the test data using the pruned model of the CART
test_predictions <- predict(pruned_cart_model, test_data_cart, type = "class")

# Convert predictions to a factor with appropriate levels
test_predictions <- factor(test_predictions, levels = c("N", "Y"))

# creating a data frame to store the predictions
test_predictions_df <- data.frame(Loan_ID = test_data$Loan_ID, Loan_Status = test_predictions)

# checking the first few rows of the test_predictions_df
head(test_predictions_df)
```

```
##     Loan_ID Loan_Status
## 1 LP001015           Y
## 2 LP001022           Y
## 3 LP001031           Y
## 4 LP001035           Y
## 5 LP001051           Y
## 6 LP001054           Y
```

Random Forests

Random Forest is an ensemble method that is used to build multiple decision trees that combines their predictions to enhance the performance of a model and accuracy of the predictions made. Random Tree helps to treat the limitations of individual decision trees, overfitting, and instability.

Key Concepts of Random Forest

1. Ensemble Learning:

Ensemble learning denotes the techniques that combine multiple and weaker models into a stronger combined overall model by the intuition is that by combining models, one obtains a predictive performance better than the best individual model through reduced variance and bias of the combined models.

2. Bagging (Bootstrap Aggregating):

Random Forest uses bagging which is a technique where several models fit under decision trees on bootstrapped subsets of the original data, i.e., random samples with replacement. Because of the way in which random forests take the average of predictions for regression or use a majority vote for classification, the overall variance and overfitting are reduced in yielding robust models.

3. Random Feature Selection:

In addition to bagging at each node during the process of tree building, Random Forest introduces random feature selection. Instead of considering all features for making the split, only a random subset of them are evaluated. This decorrelates the trees due to which more diverse models are obtained, and this leads to the prevention of overfitting.

4. Improved Accuracy:

By aggregating the output from multiple trees, Random Forest generates more precise and more generalized predictions than a single decision tree, particularly on larger and complicated datasets.

Advantages of Random Forest over CART:

- Reducing Overfitting:

Random Forest reduces overfitting by combining multiple decision trees. Each tree in the forest has only partial information, therefore, predicts an average output, unlike a single, deep decision tree which can easily suffer from overfitting.

- Greater Stability:

In doing so, Random Forest aggregates predictions from many trees and consequently yields a more stable model: it reduces the variance, making the model less sensitive to noise in the training data.

- Better Generalization:

The main benefits of an ensemble of trees are that Random Forest generalizes better to new, unseen data, which typically leads to better performance on test sets than individual decision trees do.

- Feature Importance:

While making predictions, Random Forests have a built-in method to assess the importance of different features. You can get insight into which features are most influential by looking at the reduction in prediction error when a feature is permuted.

Implementing Random Forest

```r
# loading the randomforest library for implementing the random forest
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 4.4.2

## randomForest 4.7-1.2

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:gridExtra':
##
##     combine

## The following object is masked from 'package:dplyr':
##
##     combine

## The following object is masked from 'package:ggplot2':
##
##     margin
```

```r
# Duplicating test_data for non-regularized modelling first to understand the effect of regularization
test_data_non_regularized <- test_data

# Training a Random Forest model on the training data
rf_model_non_regularized <- randomForest(Loan_Status ~ ., data = train_data)

# Predicting Loan_Status on the test_data_non_regularized
rf_predictions_non_regularized <- predict(rf_model_non_regularized, test_data_non_regularized)
```

```
# Storing the predictions in the test_data_non_regularized for non-regularized model
test_data_non_regularized$Predicted_Loan_Status_Non_Regularized <- rf_predictions_non_regularized

# displaying the first few rows of the predictions made
head(test_data_non_regularized, 10)
```

```
##     Loan_ID Gender Married Dependents    Education Self_Employed
## 1  LP001015   Male     Yes          0     Graduate            No
## 2  LP001022   Male     Yes          1     Graduate            No
## 3  LP001031   Male     Yes          2     Graduate            No
## 4  LP001035   Male     Yes          2     Graduate            No
## 5  LP001051   Male      No          0 Not Graduate            No
## 6  LP001054   Male     Yes          0 Not Graduate           Yes
## 7  LP001055 Female      No          1 Not Graduate            No
## 8  LP001056   Male     Yes          2 Not Graduate            No
## 9  LP001059   Male     Yes          2     Graduate            No
## 10 LP001067   Male      No          0 Not Graduate            No
##    ApplicantIncome CoapplicantIncome LoanAmount Loan_Amount_Term Credit_History
## 1             5720                 0        110              360              1
## 2             3076              1500        126              360              1
## 3             5000              1800        208              360              1
## 4             2340              2546        100              360              1
## 5             3276                 0         78              360              1
## 6             2165              3422        152              360              1
## 7             2226                 0         59              360              1
## 8             3881                 0        147              360              0
## 9            13633                 0        280              240              1
## 10            2400              2400        123              360              1
##    Property_Area Predicted_Loan_Status_Non_Regularized
## 1          Urban                                     Y
## 2          Urban                                     Y
## 3          Urban                                     Y
## 4          Urban                                     Y
## 5          Urban                                     Y
## 6          Urban                                     Y
## 7      Semiurban                                     Y
## 8          Rural                                     N
## 9          Urban                                     Y
## 10     Semiurban                                     Y
```

```
#visualizing the Random Forest model by using importance to check the importance of the features and ov
importance(rf_model_non_regularized)  # Feature importance is plotted
```

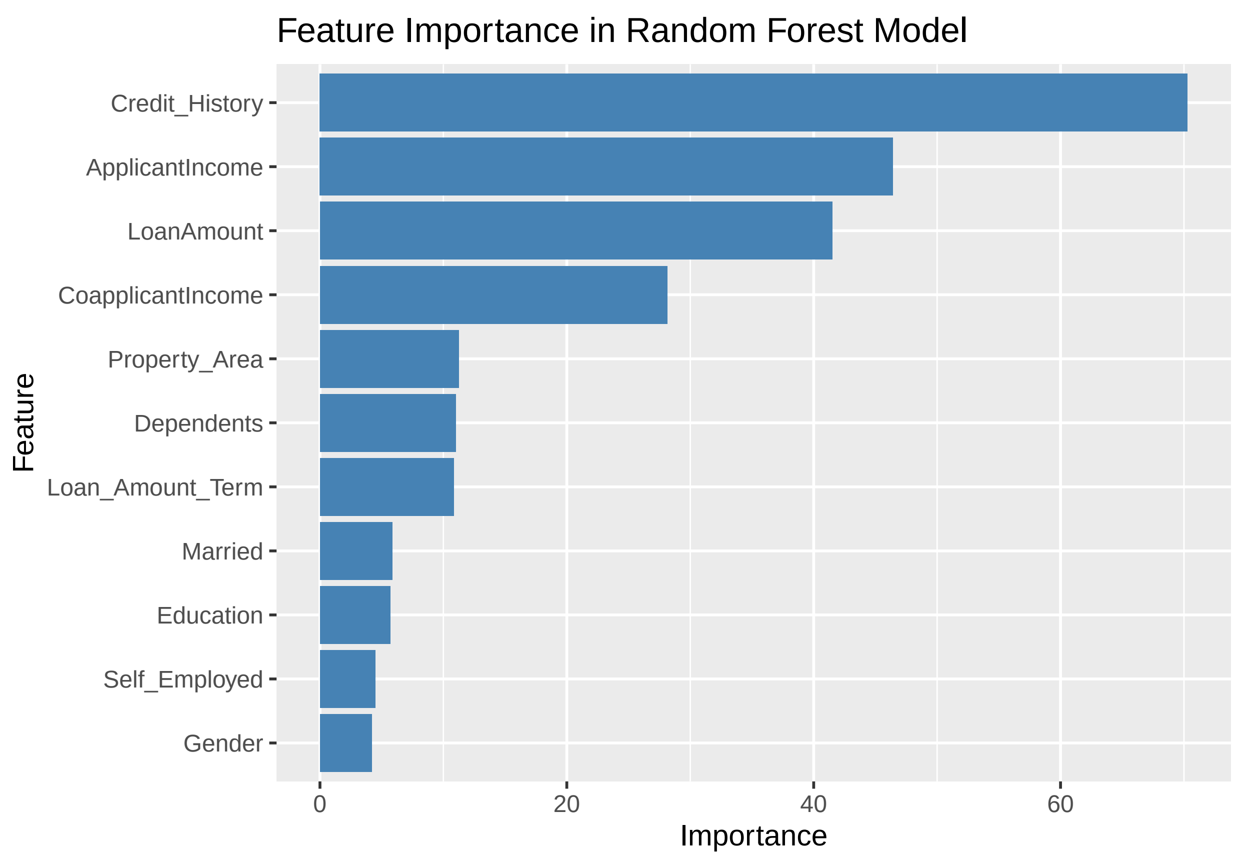
```
##                   MeanDecreaseGini
## Gender                    4.207265
## Married                   5.861235
## Dependents               10.991100
## Education                 5.693099
## Self_Employed             4.487307
## ApplicantIncome          46.423847
## CoapplicantIncome        28.122514
## LoanAmount               41.501141
## Loan_Amount_Term         10.837508
## Credit_History           70.285628
## Property_Area            11.235852
```

```r
# Displaying the summary of the Random Forest model
print(rf_model_non_regularized)
```

```
##
## Call:
##  randomForest(formula = Loan_Status ~ ., data = train_data)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 20.68%
## Confusion matrix:
##    N   Y class.error
## N 85 107  0.55729167
## Y 20 402  0.04739336
```

Plotting the feature importance before in the random forest modeling to analyse the overfitting of the feature which could be rectified in order to make better and accurate prediction.

```r
# Plotting the feature importance of the Random Forest model and using the ggplot2 library
library(ggplot2)

# storing the values in feature_importance of the rf_model_non_regularized
importance_values <- importance(rf_model_non_regularized)
feature_importance <- data.frame(Feature = rownames(importance_values),
                                 Importance = importance_values[, 1])

# Sorting by importance values of the features in the random forest for better interpretation
feature_importance <- feature_importance[order(-feature_importance$Importance), ]

# Creating a bar plot of the features with highest importance
ggplot(feature_importance, aes(x = reorder(Feature, Importance), y = Importance)) +
  geom_bar(stat = "identity", fill = "steelblue") +
  coord_flip() +
  labs(title = "Feature Importance in Random Forest Model",
       x = "Feature", y = "Importance")
```



Regularization Techniques in Random Forest and Decision Tree:

Regularization can be defined as one of the techniques that avoids overfitting just by adding a penalty term to the model's complexity. Overfitting is a situation whereby a model learns the noise in the training data instead of the underlying pattern; this results in poor performance on unseen data. For Decision Trees and Random Forests, regularization is primarily a matter of managing the complexity of the trees to generalize well. The Regularization techniques used in Decision Tee and Random Forest are as below:

Decision Trees Regularization Techniques

- Maximum Depth (max_depth or maxnodes): Restricts how deep the tree can be. A very deep tree could overfit the training data by learning overly specific patterns.
- Minimum Samples per Split (min_samples_split or nodesize): This is the minimum number of samples required to split an internal node.
- Minimum Samples per Leaf (min_samples_leaf): Ensures that each terminal node (leaf) contains at least a certain number of samples. This prevents overly specific splits.
- Maximum Features (max_features or mtry): Limits the number of features considered at each split. Fewer features reduce the likelihood of overfitting.
- Cost Complexity Pruning (CCP): Prunes the tree by adding a penalty for the number of nodes. It reaches an optimum between model fit and complexity by minimizing a cost function.

Regularization Techniques in Random Forests

Random Forests have some regularization by their design, but further regularization can help in improving performance:

- Number of Trees (ntree): A larger number of trees increases stability and reduces variance. However, there is a trade-off with computational cost.
- Maximum Features (mtry): Random Forests use a random subset of available features at each split. This randomness reduces overfitting and improves generalization.
- Minimum Node Size (nodesize): As in decision trees, it controls the minimum number of samples needed in a node before a split can be performed.
- Maximum Number of Terminal Nodes (maxnodes): Bounds the total number of terminal nodes in each tree and prevents too complex trees.

-> As we have already used the regularization techniques - hyperparamters max_depth, min_split and min_buckets in the Decision tree during hyperparameter tuning, let us use the regularization techniques in the Random Forest to understand and analyse the variance in the predictions before and after applying regularization in the dataset while training.

Use of Regularization Technique in the Random Forest

As we can observe using the importance of features in the Random Forest Credit_History below, it holds the highest importance when implementing a random forest which is overfitting and will hinder our predictions on the data. Therefore, Let us use the regularization Techniques - number trees and number of features at each split,minimum samples in the terminal nodes, and Maximum number of terminal nodes.

```r
# loading the randomforest library to implement the regularized random forest
library(randomForest)

# Duplicating test_data for regularized model as test_data_regularized
test_data_regularized <- test_data

# Training a Random Forest model on the training data (Regularized)
rf_model_regularization <- randomForest(
  Loan_Status ~ .,
  data = train_data,
  ntree = 200,    # Number of trees
  mtry = 3,       # Number of features at each split
  nodesize = 10,  # Minimum samples in terminal nodes
  maxnodes = 20   # Maximum number of terminal nodes
)

# Predicting Loan_Status on test_data_regularized
rf_predictions_regularized <- predict(rf_model_regularization, test_data_regularized)

# Storing predictions in the test_data_regularized for regularized model
test_data_regularized$Predicted_Loan_Status_Regularized <- rf_predictions_regularized

# feature importance of the Random Forest model after regularization
importance(rf_model_regularization)  # Feature importance
```

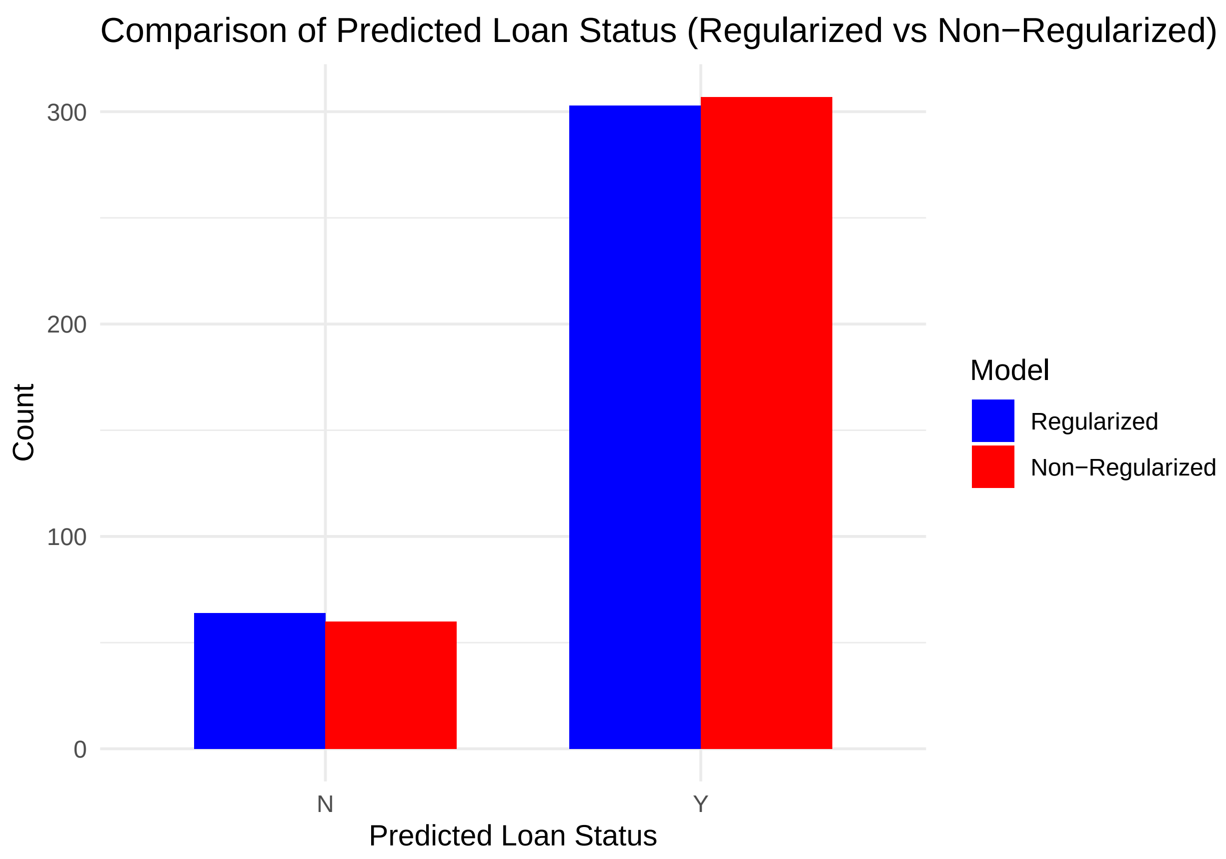
```
##                    MeanDecreaseGini
## Gender                     0.7503836
## Married                    1.9829156
## Dependents                 1.9056412
## Education                  1.7683859
## Self_Employed              0.7931597
## ApplicantIncome           10.0694102
## CoapplicantIncome          8.4202048
## LoanAmount                10.0216095
## Loan_Amount_Term           3.8107263
## Credit_History            61.7179679
## Property_Area              3.2850851
```

```r
# Displaying the summary of the Regularized Random Forest model
print(rf_model_regularization)
```

```
##
## Call:
##  randomForest(formula = Loan_Status ~ ., data = train_data, ntree = 200,      mtry = 3, nodesize = 1
##                Type of random forest: classification
##                      Number of trees: 200
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 19.22%
## Confusion matrix:
##    N   Y class.error
## N 83 109  0.56770833
## Y  9 413  0.02132701
```

We are plotting below a graph to show the comparison but we do not see much variance with the predictions made before and after using regularization techniques but we can notice there is a slight variance. The slight variance can be due the treated data values in place of the missing values and also the feature importance in Credit history is an influential parameter which is making the predictions on the dataset. Below is the comparison Bar plot for regularized and non-regularized Random Forest:

```r
# Storing the predictions on loan status in test_data for comparison
test_data$Predicted_Loan_Status_Regularized <- rf_predictions_regularized
test_data$Predicted_Loan_Status_Non_Regularized <- rf_predictions_non_regularized

# Plotting the comparison of predictions between regularized and non-regularized random forest model
# Reshaping the data into long format
library(tidyr)
comparison_data <- test_data %>%
  pivot_longer(cols = c(Predicted_Loan_Status_Regularized, Predicted_Loan_Status_Non_Regularized),
               names_to = "Model", values_to = "Prediction")

# Creating the bar plot
ggplot(comparison_data, aes(x = Prediction, fill = Model)) +
  geom_bar(position = "dodge", width = 0.7) +
  labs(title = "Comparison of Predicted Loan Status (Regularized vs Non-Regularized)",
       x = "Predicted Loan Status",
       y = "Count") +
  scale_fill_manual(values = c("blue", "red"),
                    labels = c("Regularized", "Non-Regularized")) +
  theme_minimal()
```



```r
# displaying the comparison
print(comparison_data)
```

```
## # A tibble: 734 x 14
##    Loan_ID  Gender Married Dependents Education    Self_Employed ApplicantIncome
##    <chr>    <chr>  <chr>   <chr>      <chr>        <chr>                   <int>
##  1 LP001015 Male   Yes     0          Graduate     No                       5720
##  2 LP001015 Male   Yes     0          Graduate     No                       5720
##  3 LP001022 Male   Yes     1          Graduate     No                       3076
##  4 LP001022 Male   Yes     1          Graduate     No                       3076
##  5 LP001031 Male   Yes     2          Graduate     No                       5000
##  6 LP001031 Male   Yes     2          Graduate     No                       5000
##  7 LP001035 Male   Yes     2          Graduate     No                       2340
##  8 LP001035 Male   Yes     2          Graduate     No                       2340
##  9 LP001051 Male   No      0          Not Graduate No                       3276
## 10 LP001051 Male   No      0          Not Graduate No                       3276
## # i 724 more rows
## # i 7 more variables: CoapplicantIncome <int>, LoanAmount <dbl>,
## #   Loan_Amount_Term <dbl>, Credit_History <chr>, Property_Area <chr>,
## #   Model <chr>, Prediction <fct>
```

How does the performance of the method change when applied to imbalanced datasets?

Decision Tree Performance on Imbalanced Datasets

Decision tree performance and other machine learning models usually get affected by imbalanced datasets where one class that is the majority class grossly gets outnumbered by the other, the minority class. This can be problematic for the modelling, which may end up biased towards the majority class, because it minimizes overall error by just predicting the majority class for most instances. That may cause extremely poor performances on the minority class.

Challenges with Imbalanced Dataset in Decision Trees:

- Class Imbalance and Overfitting: Decision trees inherently make splits to achieve an overall good accuracy by tending towards the majority class. A tree might overfit the majority class leading to poor generalization and hence low performance on the minority class.
- Predictive Bias: In imbalanced datasets, the decision tree might favor the majority class to achieve higher accuracy by totally ignoring the minority class.

Strategies to Address Imbalanced Datasets in Decision Tree Modeling

1. Class Weighting

Class weighting assigns higher penalties towards mistakes made on the minority class therefore, providing for encouragement towards making these models pay more attention towards the instances.

How it works: The decision tree is modified to give more importance to the minority class by assigning higher weights to it. During the learning process, misclassifications of the minority class incur a higher cost (penalty), leading the model to make splits that are more likely to capture the minority class instances. Bias in Decision Trees: With the increased weight of the minority class, the decision tree will now have a better incentive to split in ways that it's more likely to be well-represented in the tree. This helps decrease the bias toward the majority class.

Below we will be implementing the Class weights by specifying while training a decision tree using the rpart function in R by incorporating class weights with 10-fold cross-validation:

```
# Checking the frequency i.e. the number of counts of each class or label in the target variable 'Loan_
class_counts <- table(train_data$Loan_Status)

# Displaying the counting for each class
print(class_counts)
```

```
##
##   N   Y
## 192 422
```

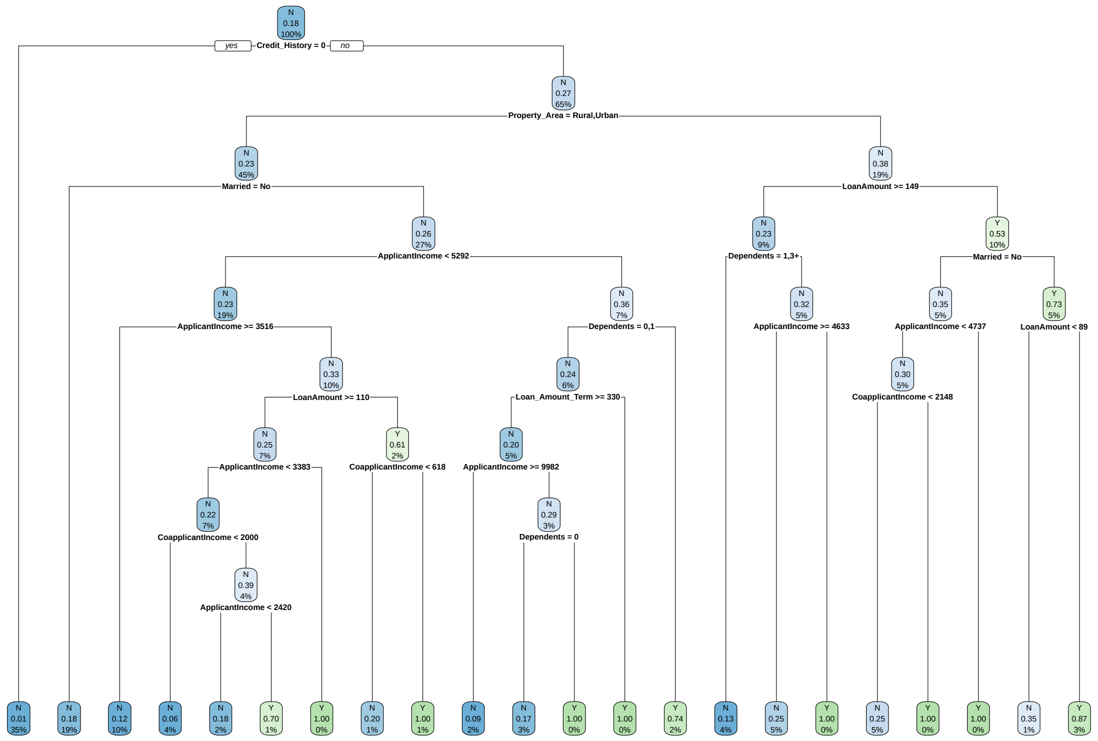
```
# Identifying the majority and minority class
majority_class <- names(class_counts)[which.max(class_counts)]
minority_class <- names(class_counts)[which.min(class_counts)]

#Displaying the majority and minority class
cat("Majority Class: ", majority_class, "\n")
```

```
## Majority Class:  Y
```

```r
cat("Minority Class: ", minority_class, "\n")
```

```
## Minority Class:  N
```

```r
# Creating a vector for class weights
# As 'N' is the minority class and 'Y' is the majority class therefore assigning the higher value of 'N
class_weights <- ifelse(train_data$Loan_Status == "N", 10, 1)

# Training the decision tree with class weights using rpart
model <- rpart(Loan_Status ~ ., data = train_data,
               weights = class_weights, method = "class")

# Visualizing the decision tree
rpart.plot(model)
```



2. Oversampling the Minority Class Oversampling the minority class is a strategy used to duplicate the minority classes which will be used to effectively balance the dataset by creating more copies of the minority classes.
3. Undersampling the Majority Class Undersampling the Majority Class is used to reduce the majority class to balance the dataset to avoid ovrfitting caused by oversampling.
4. SMOTE This is a sophisticated technique that generates synthetic samples for the minority class, rather than simply duplicating existing samples, to better capture the underlying distribution of the minority class.

Bias-variance trade-off impacts the performance of the Decision Tree, CART, and Random Forest?

The bias-variance tradeoff is a balance between a model's ability to capture true patterns in the data (low bias) and its tendency to fluctuate too much as a result of the specific training data it is presented with (high variance).

CART (Classification and Regression Trees)

- High Variance: Decision Trees, especially when grown deep (without pruning), can easily overfit to the noise in the training data. This leads to high variance, as small changes in the training data could result in completely different tree structures.
- High Bias: Shallow trees or trees with too many restrictions (like limiting depth or number of samples per leaf), can underfit the data, leading to high bias.
- Bias-Variance Trade-off in CART: The major problem with CART is to find an appropriate balance between underfitting high bias and overfitting high variance. Regularization methods such as pruning which can be done by setting an optimal complexity parameter can be used to reduce the variance without focusing too much bias.

Random Forests

- Low Bias: Since Random Forest is an ensemble of trees, it usually has lower bias. The combination of multiple trees results in a more accurate, generalized prediction.
- Lower Variance: The processes of bagging and random feature selection in building the trees ensure that individual trees are less correlated, hence lowering overall variance. This, in turn, makes Random Forests less prone to overfitting compared to a single decision tree.
- Bias-Variance: The random forest tames the variance of any one of the decision trees by averaging the results over multiple trees. This is an ensemble approach that often results in a more stable model and improves generalization, especially when compared to CART.

The bias-variance trade-off is a very fundamental view in the learning of machines. That is a trade-off between the low bias of a model fitting a given training dataset and low variance concerning how well that model generalizes to unseen data.

Impact on Decision Trees:

- High Bias: A shallow decision tree might underfit the data, hence high bias.
- High Variance: A deep and complex decision tree will overfit the data, hence leading to a high variance.

Does the models work on small data and if not is there an suitable alternative?

CART and Small Datasets Handling by Decision Trees Decision Trees are sensitive to small datasets, especially in terms of overfitting. With fewer data points, the tree might learn noisy data patterns rather than the underlying relationship between features and the target variable. This makes the model unstable with small changes in the dataset, resulting in drastically different tree structures each time the data is changed.

Cross-validation can help in training the model on different subsets of data and testing it on the test data. This helps to maximize the use of the available data. However, overfitting can still occur, and the model's performance might not generalize well on unseen data.

Alternative for Small Datasets: Although ensemble methods like Random Forests can still work on small datasets, the performance gain compared to a single decision tree is limited by the dataset size. Actually, Random Forests may not take full advantage of their power on small datasets since they require enough diversity in the data to make useful ensembles of trees.

Small Dataset Alternative: For very small datasets, Support Vector Machines (SVM) or k-Nearest Neighbors may be more suitable. The SVMs, for instance, search for this hyperplane that maximizes the margin between classes and can generalize well with few data points. k-NN is non-parametric and it also works with a small number of data because it stores training instances and uses them directly to give predictions. Regularization becomes even more critical with small datasets. Using techniques like pruning in decision trees and setting parameters like max_depth, min_samples_split, and min_samples_leaf can help make sure the tree does not get too complex and overfit to the limited data.

Is Robust the Models?

Robustness of Decision Trees

Noise Robustness : In general, decision trees can be somewhat robust with regard to noise because each decision is based on more or a set of decisions within every node. However, for data that is highly noisy, the trees are not at all resilient. With some slight change in data but very specifically in the training set of the system, there can be enormous variations in the structure of a tree, hence leading it toward model instability.

Improving Robustness: Random Forests greatly enhance robustness. By averaging the predictions of many trees, each trained on a different random subset of the data, the model becomes much less sensitive to noise or outliers in the data. Random Forests can handle noise better because they rely on multiple trees that are less likely to be overfit to the noise in the data.

Robustness to Small Changes in Data CART: A decision tree is quite sensitive for small datasets or small perturbations in the data. In fact, this sensitivity to small changes in the data is considered one of the major reasons for the tendency of decision trees to overfit. The sensitivity can be dampened to some degree and robustness increased but either by pruning the tree or setting complexity parameters.

Random Forest: In contrast, Random Forests are much more robust to such changes, as the aggregation of many trees helps smooth out the noise that results from small changes. Thus, Random Forests normally yield more stable and reliable predictions on new, unseen data than single decision trees.

Test on small dataset or suitable alternative

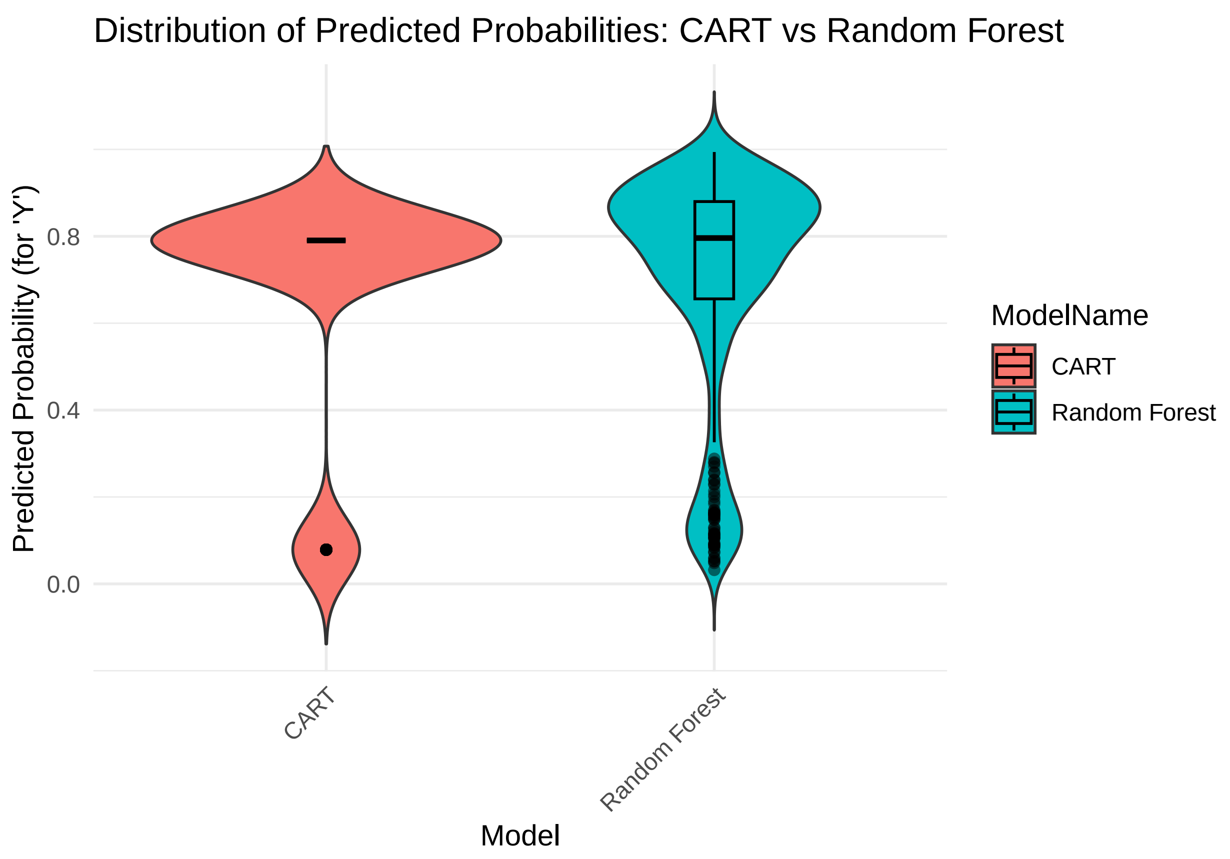
Performance of decision trees and random forests on small datasets:

Decision Trees: As discussed, decision trees are sensitive to small datasets. For example, on a small number of examples, the tree may overfit the peculiarities of the training data and provide poor generalization. The risk can be reduced by regularizing the tree using pruning, setting limits on depth, and minimum samples per leaf.

Random Forests While Random Forests reduce the variance problem by averaging several trees, much of this advantage is lost when only a small dataset is available. However, they may still turn in a better performance than using a single decision tree alone because of the ability of the ensemble learning method in reducing variance.

```r
# probabilities for CART Decision Tree
cart_probabilities <- predict(pruned_cart_model, test_data, type = "prob")
cart_probabilities <- data.frame(Loan_ID = test_data$Loan_ID,
                                 CART_Y = cart_probabilities[, "Y"],
                                 CART_N = cart_probabilities[, "N"])
```

```
# probabilities for Random Forest
rf_probabilities <- predict(rf_model_non_regularized, test_data, type = "prob")
rf_probabilities <- data.frame(Loan_ID = test_data$Loan_ID,
                               RF_Y = rf_probabilities[, "Y"],
                               RF_N = rf_probabilities[, "N"])

# Combining the probabilities them into a single data frame for plotting
combined_probabilities <- data.frame(
  Loan_ID = test_data$Loan_ID,
  CART_Y = cart_probabilities$CART_Y,
  RF_Y = rf_probabilities$RF_Y
)

# reshaping the data in longer format for the plotting
library(tidyr)
long_data <- combined_probabilities %>%
  gather(key = "Model", value = "Probability", -Loan_ID)

# Adding models labels for easy interpretation of the plotting
long_data$Model <- factor(long_data$Model, levels = c("CART_Y", "RF_Y"))
long_data$ModelName <- ifelse(long_data$Model == "CART_Y", "CART", "Random Forest")
# loading the ggplot for the plotting of te violin and box plot
library(ggplot2)

# Creating the violin plot X axis as ModelName, Y axis as Probability and fill with ModelName
ggplot(long_data, aes(x = ModelName, y = Probability, fill = ModelName)) +
  geom_violin(trim = FALSE) +    # Showing full distribution without trimming
  geom_boxplot(width = 0.1, color = "black", alpha = 0.5) + # Adding boxplot inside the violin plot
  labs(title = "Distribution of Predicted Probabilities: CART vs Random Forest",
       x = "Model",
       y = "Predicted Probability (for 'Y')") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



The violin plot together with the boxplot from the above code provide the visual of the different ranges of predicted probabilities across the target class (‘Y’) by the CART and Random Forest models. This helps to show how each model has approached the prediction:

CART: In general, the probability distribution from the decision tree model might be more discrete (sharp peaks) since it’s a categorical decision at every node.

Random Forest: Since Random Forests aggregate predictions from several trees, the resulting distribution is usually much smoother and more stable, perhaps reflected in the violin plot as more spread out or symmetric.

By visually comparing the predictions of these two models, we can estimate not only their precision but also the confidence of a model in making decisions that may help us in further improvements and selection of the best model for the problem we deal with.

CONCLUSION

In this report, we have implemented the Decision Tree supervised learning algorithm using the R programming language. The main objective of this report is to gain a deeper understanding of the chosen supervised learning method by applying it to an available dataset. The analysis was carried out using R for statistical computing and empirical analysis, with a focus on experimentation.

The dataset chosen for this report is the Financial Loan Approval Prediction data from Kaggle – “https://www.kaggle.com/datasets/krishnaraj30/finance-loan-approval-prediction-data”. There are two CSV files: the train_data, which is used for training the model and is split into a validation dataset to assess performance using various metrics, and the Test.csv file, which is used to test the model trained on the data and predict the loan approval status (approved or rejected). The model’s predictions are further validated by tuning various hyperparameters to improve accuracy.

The problem statement of this report is to build an automated loan eligibility prediction system that can determine whether a customer is likely to receive loan approval based on factors such as income, loan amount, credit history, and other features.

We visualized the distribution of features in the train_data set and conducted a model performance evaluation along with metric selection. This process began with handling missing values and performing data preprocessing. We then used classification evaluation metrics—accuracy, precision, AUC, recall, and F1 score—to assess the performance of the decision tree model. Additionally, we plotted the model's performance across various training and validation sizes.

Explored how the performance of the Decision Tree model varies on both the training and validation data as hyperparameters are adjusted. Hyperparameter tuning was conducted on parameters such as max depth, min split, and min bucket, with the best settings selected based on the highest AUC performance. The optimal hyperparameter was chosen, and the performance was reported using the test data. Specifically, the max_depth hyperparameter was used, along with cross-validation in decision tree modeling to avoid overfitting. Predictions made on the test data were compared, analyzing the variance in hyperparameters and evaluating the results using K-fold cross-validation.

Further extended the learning on Classification and Regression Trees (CART) and Random Forests as supervised learning methods, where the implementation of a CART binary tree with an optimal complexity parameter was explored. Additionally, the advantages and limitations of Random Forests were examined, along with its implementation and the study of feature importance. Regularization techniques in both Random Forests and Decision Trees were explored, including parameters such as the number of trees (ntree), maximum features (mtry), minimum node size (nodesize), and maximum number of terminal nodes (maxnodes). A comparison was made between Random Forests with and without regularization techniques to prevent overfitting. Strategies for addressing imbalanced datasets in decision tree modeling were discussed, along with the impact of the bias-variance trade-off on the performance of Decision Trees, CART, and Random Forest.

Also, Compared the performance of Decision Trees and Random Forests on small datasets, the decision trees have the potential to overfit and not generalize well. This risk was reduced by regularizing a tree through pruning, using the limitation of depth, or specifying minimum samples per leaf. For the case of Random Forests, since it reduces the variance just by averaging out multiple trees, this advantage is mostly lost for small datasets. However, random forests still often outperform a single decision tree owing to their ensemble learning approach, which helps in reducing variance and improving generalization.

Finally, we visualized the variance in predicted probabilities using a violin plot, comparing the performance of CART and Random Forest models. This report provided a comprehensive understanding of using the decision tree model to predict loan approval or rejection based on the training data. Additionally, we compared its performance with other modeling techniques, incorporating methods like regularization, data preprocessing, hyperparameter tuning, and cross-validation.

In conclusion, we conducted an in-depth **exploratory analysis of decision tree models** and other related methods, applying various techniques to optimize performance and improve effectiveness. The aim was to refine the decision tree model to its fullest potential, making it as accurate and reliable as possible for predicting loan eligibility.

REFERENCES

1. 1. Müller, A.C. and Guido, S., 2016. Introduction to Machine Learning with Python. O'Reilly Media.
2. Géron, A., 2019. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly Media.

3. Hastie, T., Tibshirani, R. and Friedman, J., 2009. The Elements of Statistical Learning. Springer.
4. Bishop, C.M., 2006. Pattern Recognition and Machine Learning. Springer.
5. Murphy, K.P., 2012. Machine Learning: A Probabilistic Perspective. MIT Press.
6. Witten, I.H., Frank, E. and Hall, M.A., 2016. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann.
7. Ng, A., 2017. Machine Learning Yearning. Self-published. Available at: https://www.mlyearning.org [Accessed 27 November 2024].
8. Goodfellow, I., Bengio, Y. and Courville, A., 2016. Deep Learning. MIT Press.
9. Bruce, P., Bruce, A. and Gedeck, P., 2016. Practical Statistics for Data Scientists. O'Reilly Media.
10. Provost, F. and Fawcett, T., 2013. Data Science for Business. O'Reilly Media.
11. Burkov, A., 2019. The Hundred-Page Machine Learning Book. Andriy Burkov.
12. James, G., Witten, D., Hastie, T. and Tibshirani, R., 2013. Introduction to Statistical Learning. Springer.
13. Kumar, R.S.M.G.S., 2013. Machine Learning: A Review of Techniques. International Journal of Computer Science and Information Technologies, 4(1), pp.16-22.
14. Kuhn, M. and Johnson, K., 2013. Applied Predictive Modeling. Springer.
15. Raschka, S., 2015. Python Machine Learning. Packt Publishing.
16. KDnuggets, 2022. Decision Tree Pruning: How's and Why's. Available at: https://www.kdnuggets.com/2022/09/decision-tree-pruning-hows-whys.html
17. R Documentation, n.d. Libraries and Functions: ggplot2, dplyr, caret, pROC, tidyverse, reshape2, rpart, knitr, MLmetrics, gridExtra, ggplot(), geom_histogram(), geom_boxplot(), geom_density(), createDataPartition(), sample(), confusionMatrix(), predict(), roc(), auc(), rpart(), rpart.control(), rpart.plot(), grid.arrange(), kable(). Available at: https://www.rdocumentation.org/