Abstract

This study develops and evaluates machine learning models using Random Forest, Naive Bayes, and Support Vector Machine algorithms to predict stroke occurrence based on patient health data. The models are optimized by addressing class imbalance through oversampling and tuning hyperparameters like mtry and cost. The oversampled Random Forest model demonstrates the best performance with 97.7% accuracy on imbalanced test data. The comparisons highlight the importance of sampling techniques and parameter tuning for handling skewed data. The Random Forest model balances precision and recall with F1 scores of 99.9% and AUC of 99.93% on training and test sets, outperforming other algorithms. The results showcase the feasibility of accurate stroke prediction using machine learning given proper data preprocessing and model optimization. The study provides valuable insights for creating reliable AI-powered tools for stroke risk assessment and prevention in clinical settings.

Keywords: machine learning, stroke prediction, class imbalance, oversampling, hyperparameter tuning, model evaluation

Table of Contents

1 Introduction 5
1.1 Problem Statement 5
1.2 Aim & Objective 6
1.3 Scope of the Research 6
2 Literature Review 7
2.1 Related Works 7
2.2 Related Work Matrix 9
3 Dataset Description 10
3.1 Machine Learning Algorithm 10
3.1.1 Random Forest 11
3.1.2 Naïve Bayes 11
3.1.3 Support Vector Machine 12
4 Methodology 13
4.1 Data Preparation 13
4.1.1 Initial Data Exploration (IDA) 13
4.1.2 Exploratory Data Analysis (EDA) 14
4.1.3 Data Preprocessing 17
4.1.4 Balancing the Dataset using SMOTE 21
4.2 Model Implementation 22
4.2.1 Splitting the Dataset 22
4.2.2 Random Forest 22
4.2.3 Naïve Bayes 27
4.2.4 Support Vector Machine (SVM) 30
5 Discussion and Result 35
6 Conclusion 36
7 Future Recommendation 36
8 References 37
9 Appendix 39

List of Figures

Figure 3.1.1 Random Forest Trees 11
Figure 3.1.2 Bayes Theorem 12
Figure 3.1.3 Support Vector Machine Classifier 12
Figure 4.1.1 Structure of dataset. 13
Figure 4.1.2 Summary of the dataset. 14
Figure 4.1.3 Dataset Overview 14
Figure 4.1.4 Percentage of Stroke Occurrence 15
Figure 4.1.5 Boxplot of Continuous Variable 16
Figure 4.1.6 Density of Continuous 16
Figure 4.1.7 Missing Values 17
Figure 4.1.8 After Imputation 18
Figure 4.1.9 Before Imputation 18
Figure 4.1.10 Correlation Plot 1 20
Figure 4.1.11 Correlation Plot 2 20
Figure 4.1.12 Plot of Oversampling Stroke Variable 21
Figure 4.2.1 Summary of RF Hyperparameter Tuning 24
Figure 4.2.2 ROC of Base, Oversampled and Tuned RF Model 26
Figure 4.2.3 ROC of Base and Oversampled Model 29
Figure 4.2.4 Summary of best model. 32
Figure 4.2.5 ROC of Base, Oversampled and Tuned SVM Model 34

List of Tables

Table 3.1 Dataset Description 10
Table 4.2.2.1 Performance Base RF Model 23
Table 4.2.2.2 Performance of Oversampled RF Model 24
Table 4.2.2.3 Performance of Tuned RF Model 25
Table 4.2.3.1 Performance of Base NB Model 27
Table 4.2.3.2 Performance of Oversampled NBModel 28
Table 4.2.4.1 Performance of Base SVM Model 30
Table 4.2.4.2 Performance of Oversampled SVM Model 31
Table 4.2.4.3 Performance of Tuned SVM Model 33
Table 5.1 Comparison of best model performance 34

1 Introduction

Recognizing stroke as a major cause of deaths globally (Singh, 2021), there is a pressing need for novel techniques that enhance early warning and prevention. Due to the grim statistics from the World Health Organization, which ranks stroke as the world's second deadliest disease, innovative healthcare solutions are essential. To address this growing concern, which is largely linked to bad lifestyle and nutrition, we are transitioning to the use of artificial intelligence (AI) for preventive actions. This study undertakes the critical task of developing a machine learning model that can predict strokes based on medical information.

This study is to develop and evaluate machine learning models that can accurately predict strokes based on various health data. Popular algorithms like Random Forest, Naïve Bayes, and Support Vector Machine are used to assess their ability in making these predictions. The research purpose is to create diagnostic tests similar to those used for heart disease that can assess stroke risk. A comprehensive dataset from Kaggle that includes demographic, medical, and lifestyle information, which provides a strong foundation for building reliable AI solutions for stroke prediction.

Later in the research we start by carefully cleaning and preprocessing the raw data. Then, we train and test machine learning models after the data preparation. We focus on fixing class imbalances by taking more samples and finding the best settings for the models through tuning the parameters. This study goes into great detail about the links between healthcare, machine learning, and predictive analytics by reviewing the literature, describing the dataset, explaining the methods we used, and doing in-depth exploratory data analysis. The proceeding sections delve into the intricacies of model implementation, evaluation, and a comparative analysis of the three algorithms, concluding in a brief discussion and conclusive insights on the optimal model for stroke prediction and provide future recommendation.

1.1 Problem Statement

Stroke is the second most common cause of death globally, accounting for around 11% of all fatalities, according to the World Health Organisation (WHO). Recently, there has been a growing concern over this health issue, attributed to the widespread adoption of unhealthy lifestyles and poor dietary choices (Singh, 2021). As a result, there is a growing demand for electronic devices that can track vital health data and use advanced AI techniques to develop automated solutions (Emerging India, 2023). Following the success of predictive lab tests for

heart disease, efforts are now being made to develop similar tests for identifying one's risk of stroke. This dataset features a choice of factors that provide valuable insight into the lifestyles of the patients. As such, it presents us with a good opportunity to develop an AI-driven solution powered by the implementation of machine learning algorithms.

1.2 Aim & Objective

Aim:

This study aims to create and test machine learning models that can predict strokes reliably based on health records. We are going to test out different algorithms, like Random Forest, Naive Bayes, and Support Vector Machine, and evaluate its performance and choose the best one.

Objective:

- Collect and preprocess an appropriate dataset containing features like demographics, medical history, and lifestyle factors related to stroke.
- Develop baseline models using Random Forest, Naive Bayes, and Support Vector Machine algorithms.
- Improve model performance by addressing class imbalance through oversampling of the minority class.
- Fine-tune models by optimizing hyperparameters like mtry, kernel type, and cost through cross-validation.

1.3 Scope of the Research

The scope of the research is to compare how well different models predict stroke under different circumstances to find the most effective approach for predicting stroke. The research is limited to just predicting the occurrence of stroke from the data sourced from Kaggle. It is not applicable to the real-time data.

2 Literature Review

Stroke is a major cause of disability and death worldwide. Detecting and preventing it early is crucial for improving patient outcomes (Du et al., 2023). However, relying on traditional risk factors like age, gender, and medical history may not be enough to accurately predict stroke occurrence. This is where machine learning comes in - with its ability to uncover complex patterns in vast amounts of data, it shows promise in providing personalized and precise risk assessment (Daidone et al., 2023). In this review, we explore the current state of machine learning techniques for stroke prediction, highlighting the diverse range of algorithms, the influence of different methods and balanced datasets, and the potential for outperforming traditional risk factors.

2.1 Related Works

In this study on detecting and predicting stroke disease, Tazin et al.,(2021) utilized a range of machine learning algorithms, including Logistic Regression, Decision Tree, Random Forest, and Voting Classifier. Their research used a robust dataset consisting of 5110 records and 12 features, including crucial factors such as age, gender, hypertension, heart disease, and body mass index. To ensure the accuracy and fairness of their model building, the researchers implemented data preprocessing techniques, such as label encoding and SMOTE. The effectiveness of each algorithm was evaluated using a variety of accuracy metrics, such as accuracy score, precision score, recall score, and F1 score, producing insightful and significant findings. According to the results, Random Forest emerged as the most successful algorithm with a 96 percent accuracy, surpassing Decision Tree (94 percent) and Voting Classifier (91 percent).These findings demonstrate the potential of machine learning models to accurately predict the risk of stroke by using functional parameters. In particular, the study highlights the superiority of Random Forest, which proved to be the most dependable and powerful algorithm for this critical task of predicting stroke.

The research by Md. Monirul et al., (2021) investigates the critical issue of stroke, a rapidly evolving and potentially life-threatening brain disease. This study aims to enhance stroke prevention through early detection and timely intervention They evaluate it against other models such as Logistic Regression, Decision Tree Classifier, and K-NN and find that the Random Forest classifier significantly outperforms the others. With impressive performance metrics, achieving precision, recall, and F1-scores of 96% across the board.

Dritsas and Trigka (2022) conducted a fascinating study using a dataset from Kaggle, including over 3000 participants. The researchers employed a wide range of classifiers, including naive Bayes, logistic regression, SGD, and more, to develop prediction models for strokes. To ensure reliable results, they also utilized techniques like SMOTE, feature ranking, and selection to enhance data quality and balance. Comparing performance metrics like AUC, precision, recall, F-measure, and accuracy, the findings showed that the stacking method yielded the most effective results. With an AUC of 98.9% and strong F-measure, precision, and recall rates of 97.4%, the stacking method proved to be highly effective. Additionally, the stroke class saw an accuracy of 98%, showcasing the method's robust performance. The study not only successfully addresses limitations, but also offers valuable insights for future investigations in the field of stroke prediction.

In the recent comparative study by Biswas et al. (2022) introduces an innovative machine learning approach for diagnosing stroke using imbalanced data. Using random over sampling, the team successfully balances the data and employs eleven diverse classifiers, including support vector machine, random forest, and decision tree, to accurately predict stroke. To further enhance their results, the authors conduct rigorous hyperparameter tuning and cross-validation techniques. Through their evaluation using multiple metrics such as accuracy, precision, recall, and F1-measure, the team determines that support vector machine achieves the highest accuracy of 99.99%, closely followed by random forest with 99.87%. In addition, the team also creates a web and mobile application based on their best-performing model, providing a user-friendly interface for stroke prediction.

Another study conducted by Kadam et al.,(2022) introduced machine learning algorithms such as Logistic Regression, Decision Tree Classification, Random Forest Classification, KNN, and SVM, the research achieves notable accuracy percentages in predicting stroke risk. Logistic Regression leads with 95.71% accuracy, followed closely by SVM and KNN, emphasizing the efficacy of AI models in identifying individuals at risk of brain strokes based on diverse contributing factors.

In summary, the utilization of machine learning (ML) has proven to be a valuable asset in predicting strokes, demonstrating a wide range of capabilities and potential uses. Nevertheless, it is crucial to address any shortcomings, such as imbalanced data and the requirement for external verification, for its practical implementation. Continual advancements in the field present opportunities for incorporating advanced ML methods and utilizing broader

and more diverse datasets, leading to promising avenues for optimizing and validating models. This, in turn, has the potential to enhance preventive techniques and contribute to better outcomes in public health.

2.2 Related Work Matrix

Reference	Model	Evaluation	Remarks
Tazin et al.,(2021)	Random Forest	Accuracy: 0.96 F1-Score:0.96 Precison:0.95 Recall:0.97	Used SMOTE to Balance the Target Variable
	Decision Tree	Accuracy: 0.94 F1-Score:0.95 Precison:0.94 Recall:0.94	
	Voting Classifier	Accuracy: 0.91 F1-Score:0.91 Precison:0.91 Recall:0.91	
Md. Monirul et al., (2021)	Logistic Regression	Accuracy:0.87 F1-Score:0.87 Precison:0.87 Recall:0.87	Used SMOTE to Balance the Target Variable
	Decision Tress	Accuracy:0.93 F1-Score:0.93 Precison:0.93 Recall:0.93	
	K-NN	Accuracy:0.91 F1-Score:0.90 Precison:0.91 Recall:0.90	
Biswas et al. (2022)	Support Vector Machine	Accuracy:0.99 F1-Score:0.99 Precison:0.99 Recall:0.99	Used Feature Engineering Technique and Hyperparameter Tuning
	Random Forest	Accuracy:0.9987 F1-Score:0.9986 Precison:0.9985 Recall:0.9988	
	K-NN	Accuracy:0.986 F1-Score:0.989 Precison:0.988 Recall:0.988	
	Decision Tree	Accuracy:0.969 F1-Score:0.968 Precison:0.972 Recall:0.963	
Kadam et al.,(2022)	Logistic Regression	Accuracy:95.71%	Used only preprocessing and the Evaluation was done based on accuracy only.
	Decision Tree	Accuracy:90.21%	
	KNN	Accuracy:94.52%	
	SVM	Accuracy:94.71%	
	Random Forest	Accuracy:94.52%	

3 Dataset Description

This study utilizes a dataset obtained from Kaggle which contains 5110 rows and 11 features to predict the likelihood of a patient suffering from a stroke. The comprehensive dataset includes appropriate input variables such as gender, age, medical conditions, and smoking habits. Each row in the data contains relevant information about the patient.

Link: https://www.kaggle.com/datasets/fedesoriano/stroke-prediction-dataset

Attributes	Description	Category
Id	unique identifier of a patient	Numerical
gender	Gender of the patient	Categorical
hypertension	0 if the patient doesn't have hypertension, 1 if the patient has hypertension	Categorical
heart_disease	0 if the patient doesn't have any heart diseases, 1 if the patient has a heart disease	Categorical
ever_married	If the patient was married or unmarried	Categorical
work_type	Work type of the patient children, Govt_job, never_worked, Private or Self-employed	Categorical
residence_type	Whether the patient leave in rural or urban area	Categorical
avg_glucose_level	average glucose level in blood	Numerical
bmi	body mass index	Numerical
smoking_status	Whether the patients formerly smoked, never smoked, smokes or Unknown	Categorical
Stroke	1 if the patient had a stroke or 0 if not	Categorical

Table 3.1 Dataset Description

3.1 Machine Learning Algorithm

Machine learning is a subfield of artificial intelligence that enables computers to learn from data and improve their performance over time without being explicitly programmed. Recent journals have highlighted the importance of machine learning algorithms in various real-world applications, including healthcare, finance, and transportation. For instance, a study published by Sarker (2021) discusses the use of machine learning algorithms in healthcare, focusing on their potential to improve patient outcomes and reduce healthcare costs. Another study by Brown (2021) provides an overview of machine learning algorithms, highlighting their importance in real-world applications and research directions. Overall, machine learning algorithms are a critical tool for improving the performance of computers and enabling them to learn from data in a variety of contexts.

3.1.1 Random Forest

Random Forest is a popular supervised machine learning algorithm used by undergrads in R for solving both classification and regression problems. Random Forest algorithm stands out in classification tasks for several reasons. It's known for its flexibility and reliability, working well with different data types and sizes. Additionally, it excels in handling situations with numerous features. Random Forest builds multiple decision trees, selecting relevant features while preventing overfitting. This approach enhances prediction accuracy. Furthermore, Random Forest can be trained quickly and typically generates precise predictions even with default parameter settings, making it user-friendly for various applications. Additionally, the algorithm's ability to handle unbalanced data sets and prevent overfitting by using multiple trees in the forest enhances its performance in classification tasks. Overall, Random Forest's strength lies in its ability to provide accurate and stable predictions across various industries such as finance, healthcare, and e-commerce (Donges, 2023) .

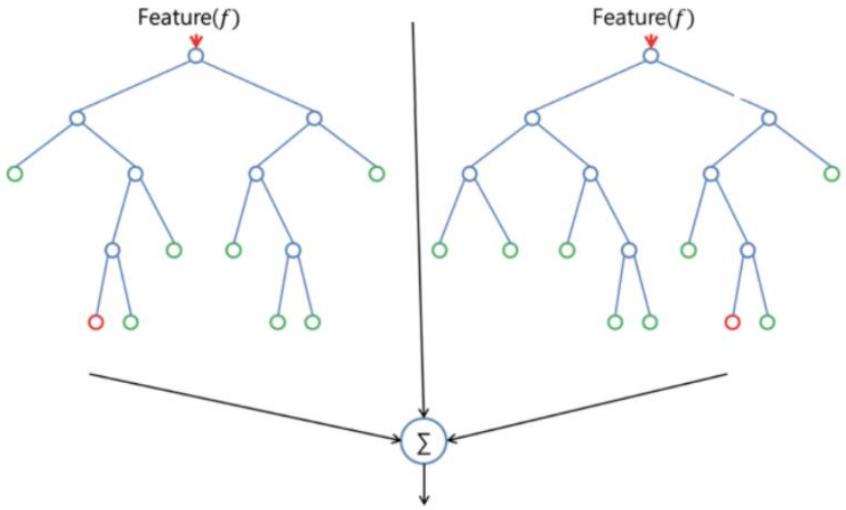


Figure 3.1.1 Random Forest Trees

3.1.2 Naïve Bayes

Naive Bayes is regarded as one of the greatest classification algorithms due to its numerous advantages. The procedure is based on Bayes' Theorem and operates under the assumption that the presence of a specific feature in a class is independent of other features; this is where the word naïve comes from. Naive Bayes classifiers are known for their simplicity, efficiency, and speed, making them ideal for large datasets where there is a requirement of quick model deployment. Despite its seemingly basic assumptions, Naive Bayes has shown

significant performance in various real-world applications such as sentiment analysis, spam filtering, and medical diagnoses. Additionally, Naive Bayes classifiers require minimal training data to estimate parameters, making them effective even with limited datasets. While other classification algorithms may outperform Naive Bayes in certain scenarios, its ease of implementation and ability to work well with small datasets make it a popular choice for many practical applications (Chauhan, 2022).

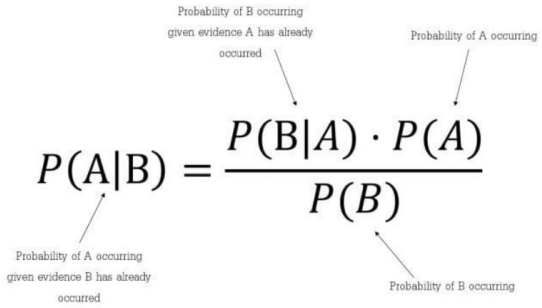


Figure 3.1.2 Bayes Theorem

3.1.3 Support Vector Machine

Support Vector Machine (SVM) is widely recognized as a great classification algorithm, with several benefits. It excels in handling data with a high number of features, making them ideal for applications with complex datasets. They effectively avoid overfitting, ensuring reliable results even with unseen data. It can efficiently handle nonlinear data using kernel functions, enabling them to precisely model complex patterns. They are versatile, supporting both classification and regression tasks through the use of various kernel functions that capture intricate data relationships. Despite these strengths, SVMs can be computationally intensive, especially with large datasets, and sensitive to parameter tuning, requiring careful optimization for optimal performance (Raj, 2022).

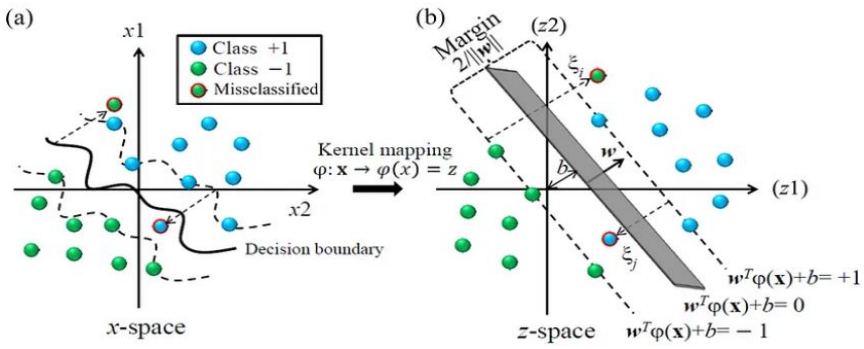


Figure 3.1.3 Support Vector Machine Classifier

4 Methodology

4.1 Data Preparation

Data preparation is essential in machine learning to ensure datasets are ready for model training and evaluation. In R, data preparation involves cleaning, transforming, and arranging raw data. This includes handling missing values, dealing with outliers, and normalizing features. R offers packages like dplyr and tidyr to simplify data wrangling. Exploratory data analysis (EDA) helps in comprehending data distribution and relationships. Categorical variables may be encoded, and data may be divided into training and testing sets. Properly prepared datasets improve model accuracy and generalization, contributing to the overall success of machine learning models in R.

4.1.1 Initial Data Exploration (IDA)

In R, data exploration begins with gaining insights into the dataset's properties. Functions like summary(), str(), and head() are used to understand variable types, data structure, and sample observations. This examination helps identify outliers and missing values, guiding data cleaning strategies for reliable analysis.

4.1.1.1 Structure of the dataset

The str() function provides information about the structure of a dataset. It shows the number of observations i.e. 5110 and the 12 variables, and their attributes, including data types and values. In this dataset, there is a 'bmi' column with a factor data type. This indicates that the 'bmi' values are categorical and need to be converted to a numeric format for further analysis.

```
'data.frame':   5110 obs. of  12 variables:
 $ id               : int  9046 51676 31112 60182 1665 56669 53882 10434 27419 60491 ...
 $ gender           : Factor w/ 3 levels "Female","Male",..: 2 1 2 1 1 2 2 1 1 1 ...
 $ age              : num  67 61 80 49 79 81 74 69 59 78 ...
 $ hypertension     : int  0 0 0 0 1 0 1 0 0 0 ...
 $ heart_disease    : int  1 0 1 0 0 0 1 0 0 0 ...
 $ ever_married     : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 2 2 1 2 2 ...
 $ work_type        : Factor w/ 5 levels "children","Govt_job",..: 4 5 4 4 5 4 4 4 4 4 ...
 $ Residence_type   : Factor w/ 2 levels "Rural","Urban": 2 1 1 2 1 2 1 2 1 2 ...
 $ avg_glucose_level: num  229 202 106 171 174 ...
 $ bmi              : Factor w/ 419 levels "10.3","11.3",..: 240 419 199 218 114 164 148 102 419 116 ...
 $ smoking_status   : Factor w/ 4 levels "formerly smoked",..: 1 2 2 3 2 1 2 2 4 4 ...
 $ stroke           : int  1 1 1 1 1 1 1 1 1 1 ...
```

Figure 4.1.1 Structure of dataset.

4.1.1.2 Descriptive Summary of the dataset

In R, the summary() function is a powerful tool for summarizing datasets. It provides a quick overview of numeric variables, displaying minimum, 1st quartile, median, mean, 3rd quartile, and maximum values. For categorical variables, it presents the frequency distribution. This summary aids in understanding data characteristics, identifying potential issues, and guiding decisions during exploratory data analysis (EDA).

```
       id          gender          age         hypertension     heart_disease     ever_married         work_type
 Min.   :   67   Female:2994   Min.   : 0.08   Min.   :0.00000   Min.   :0.00000   No :1757     children     : 687
 1st Qu.:17741   Male  :2115   1st Qu.:25.00   1st Qu.:0.00000   1st Qu.:0.00000   Yes:3353     Govt_job     : 657
 Median :36932   Other :   1   Median :45.00   Median :0.00000   Median :0.00000                Never_worked :  22
 Mean   :36518                 Mean   :43.23   Mean   :0.09746   Mean   :0.05401                Private      :2925
 3rd Qu.:54682                 3rd Qu.:61.00   3rd Qu.:0.00000   3rd Qu.:0.00000                Self-employed: 819
 Max.   :72940                 Max.   :82.00   Max.   :1.00000   Max.   :1.00000

 Residence_type avg_glucose_level      bmi               smoking_status     stroke
 Rural:2514     Min.   : 55.12    N/A    : 201   formerly smoked: 885   Min.   :0.00000
 Urban:2596     1st Qu.: 77.25    28.7   :  41   never smoked   :1892   1st Qu.:0.00000
                Median : 91.89    28.4   :  38   smokes         : 789   Median :0.00000
                Mean   :106.15    26.1   :  37   Unknown        :1544   Mean   :0.04873
                3rd Qu.:114.09    26.7   :  37                          3rd Qu.:0.00000
                Max.   :271.74    27.6   :  37                          Max.   :1.00000
                                  (Other):4719
```

Figure 4.1.2 Summary of the dataset.

The summary demonstrates that there are more females (2994) than males (2115) in the group. Additionally, it reveals that the oldest person is 82 years old. This data offers valuable information about the group's demographics.

4.1.2 Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is the fundamental step in data analysis and machine learning. Its goal is to explore the data and discover insights about it. EDA helps to understand the structure of the data, identify patterns and trends, and find relationships between variables. It also helps to identify missing data, outliers, and other data quality issues. EDA informs feature selection and engineering and helps to make sure that the data is ready for modelling.

4.1.2.1 Dataset Overview

The 'introduce()' method in the DataExplorer package provides an overview of the data, including information on missing values, discrete and continuous values, and other characteristics.

```
> introduce(ds)
  rows columns discrete_columns continuous_columns all_missing_columns total_missing_values complete_rows
1 5110      12               5                   7                   0                  202          4908
  total_observations memory_usage
1              61320       313024
```

Figure 4.1.3 Dataset Overview

4.1.2.2 Bar Plot for Stroke Occurrence

To effectively construct a model, it is important to understand the distribution of the target variable. This allows us to gain insights into the data, identify patterns and relationships, and make informed decisions about the modelling approach.

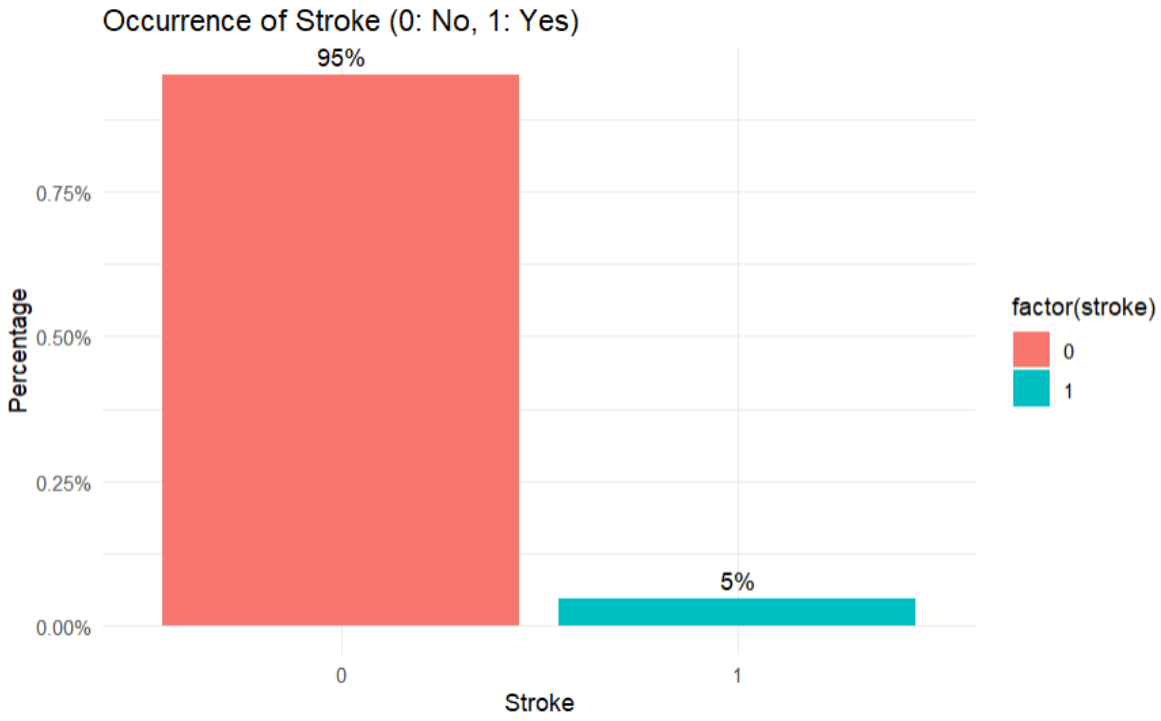


Figure 4.1.4 Percentage of Stroke Occurrence

The graph depicts the distribution of the stroke variable, illustrating a significant imbalance in the dataset. Individuals without a history of stroke account for approximately 95% of the data, while those with a history of stroke make up only 5%. To address this imbalance and ensure a proper dataset, data balancing techniques will be used during the data preprocessing stage.

4.1.2.3 Checking for Outliers

Boxplots visually represent the distribution of data and identify outliers by displaying the spread of the interquartile range (IQR). Outliers, lying beyond the whiskers, are distinct points that fall outside typical data ranges. Boxplots help highlight extreme values, aiding in the detection and interpretation of potential anomalies in a dataset.

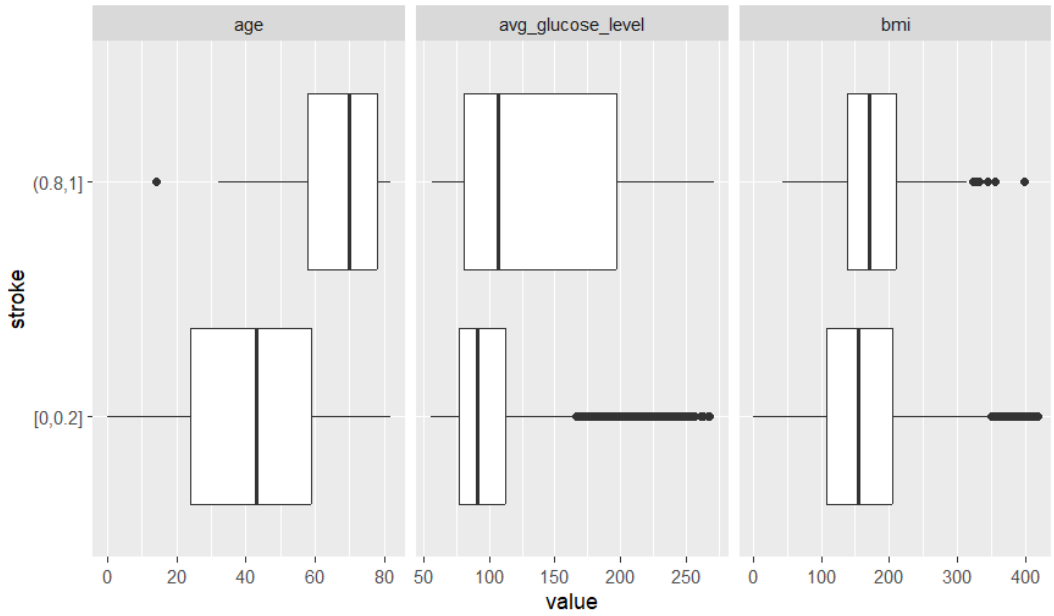


Figure 4.1.5 Boxplot of Continuous Variable

The graph represents the presence of outliers in all three variables. These outliers must be handled during the preprocessing phase of data analysis.

4.1.2.4 Checking the Distribution of Continuous Variable

Density plots are used to visualize the distribution of a continuous variable, providing insights into its shape, central tendency, and variability. They smooth histograms, offering a clearer representation of data patterns. By estimating the probability density function, density plots reveal peaks, skewness, and modes, aiding in the identification of underlying data structures.

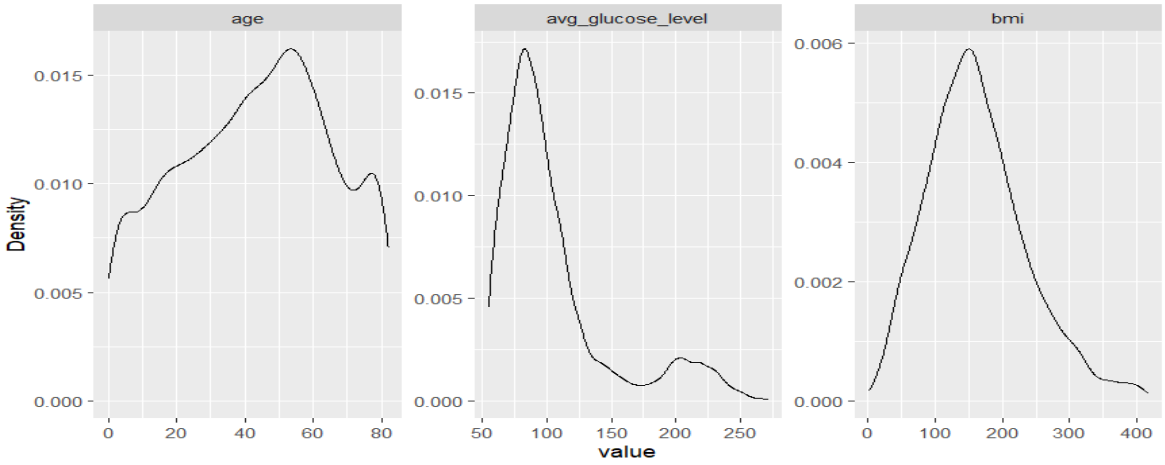


Figure 4.1.6 Density of Continuous

The above density plot represents the distributions of the variable which needs to be normalized to get a better performance.

4.1.3 Data Preprocessing

Preprocessing is an important step before using machine learning algorithm on data. It involves transforming and cleaning raw data into a useful format, improving its quality and relevance. Common tasks include managing missing values, adjusting feature scales, encoding categories, and eliminating outliers. This ensures that data is compatible, standardized, and suitable for analysis or modelling, leading to more accurate and dependable outcomes.

4.1.3.1 Data Imputation

Data imputation is a necessary step in managing datasets. It fills in missing values to keep the dataset complete. It helps maintain the sample size and prevent bias in analyses. To identify missing values in the dataset, we plot the 'NA'. Upon examining the plot, we observe that there are missing values for 'bmi' variables.

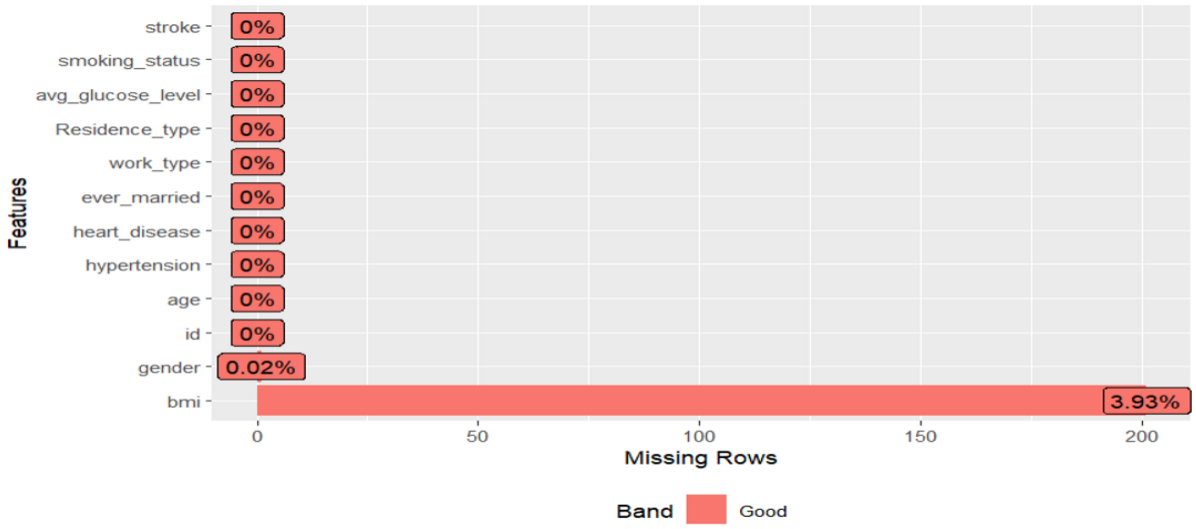


Figure 4.1.7 Missing Values

Once the missing values have been identified, they are imputed. For the "bmi" (body mass index) column, the missing values are imputed with the mean since it is a continuous value. Below plot shows that there is no missing data and the data have been successfully imputed.

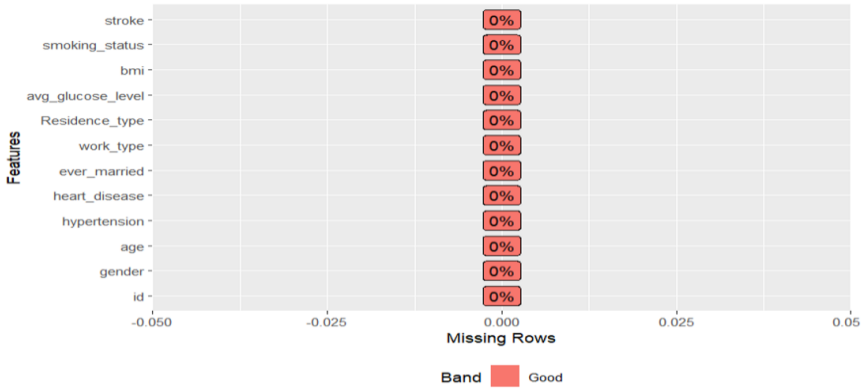


Figure 4.1.8 After Imputation

Now, checking for the unknown values in the smoking status we plot a bar graph for the easier visualization of the attribute.

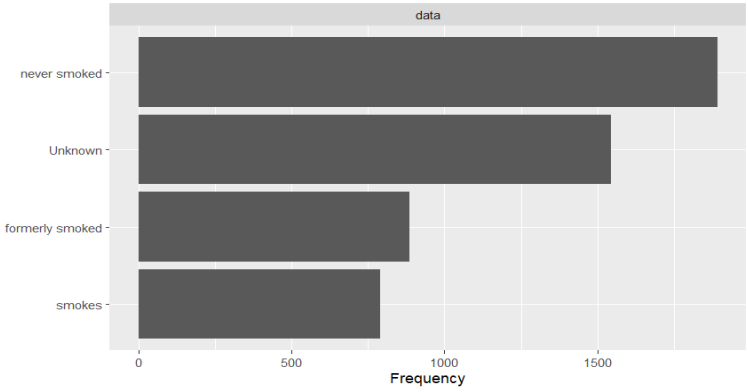
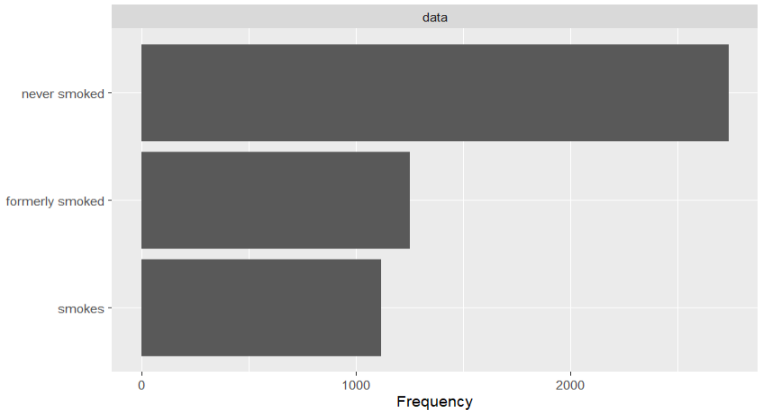


Figure 4.1.9 Before Imputation

To handle unknown values in the "smoking_status" column, we calculate the probabilities of the other categories (formerly smoked, never smoked, smoked) based on the existing data. These probabilities are then used to impute the unknown values in the "smoking_status" column Lurie et al., (2022).



4.1.3.2 Data Normalization

Here the focus is on normalizing the 'bmi' and 'avg_glucose_level' columns in our dataset to help get rid of any extreme values that might skew our results. This is super important in preprocessing, especially when it comes to machine learning.

Normalizing our data limits the values to a uniform range of 0 to 1, which is really useful for feature standardization and making sure our models perform their best. Using the 'round' function keeps our numbers nice and precise.

Standardization is like giving all our variables a fair chance to contribute to the analysis, so none of them end up overpowering the others. This way, we can trust that our models will learn from everything equally and make accurate predictions. Basically, this step helps our machine learning algorithms understand the data better and gives us more reliable predictive models.

4.1.3.3 Data Encoding

In the data encoding phase of the dataset, the "fastDummies" library is utilized to employ one-hot encoding on the "work_type" and "smoking_status" features. This process generates new binary columns in the dataset. These binary columns represent unique categories, contributing to the interpretability of machine learning algorithms. To achieve this transformation, the "mutate()" function from the "dplyr" library is employed to modify the columns effectively. Categorical variables, often represented by distinct labels, are transformed into numerical counterparts using one-hot encoding. This transformation plays a critical role in machine learning tasks, considering that most machine learning models operate exclusively with numerical data (Reminho, 2021).

4.1.3.4 Correlation Plot

The ‘plot_correlation()’ from DataExplorer visualizes correlations in a dataset. It presents a heatmap where colour shades indicate the strength and direction of numeric variable correlations. This tool helps by revealing collinearities among predictors, facilitating feature selection for optimal model performance. Providing visual insights into potential relationships between predictors, enabling informed decisions about variable selection or exclusion. Enhancing the modelling process with a simple and easy-to-understand overview of data correlations.

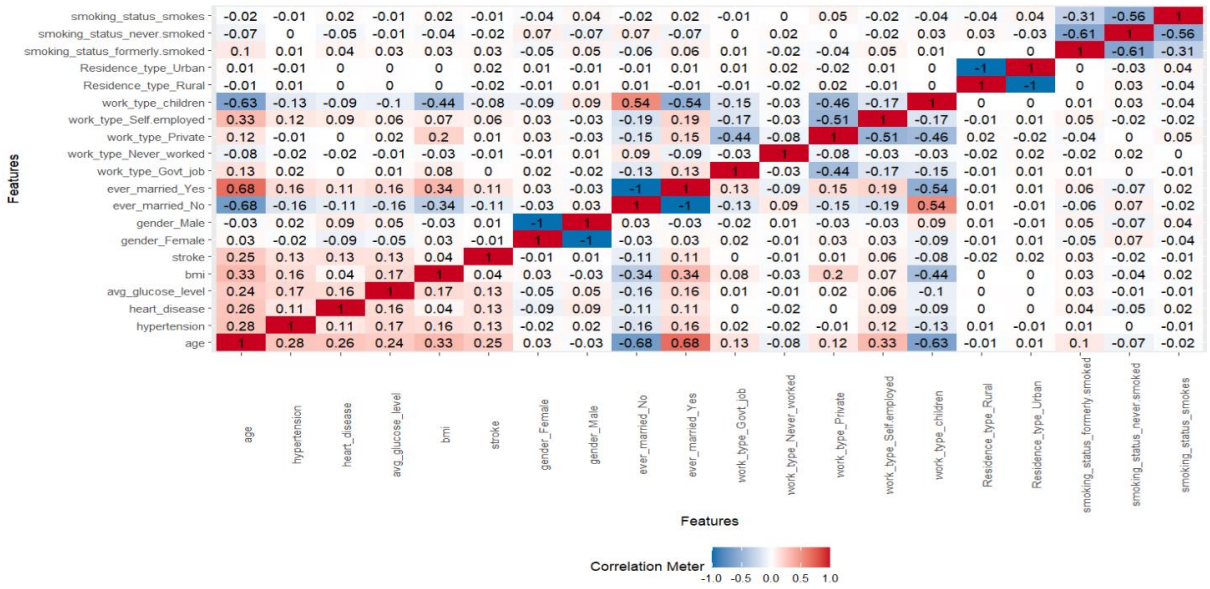


Figure 4.1.10 Correlation Plot 1

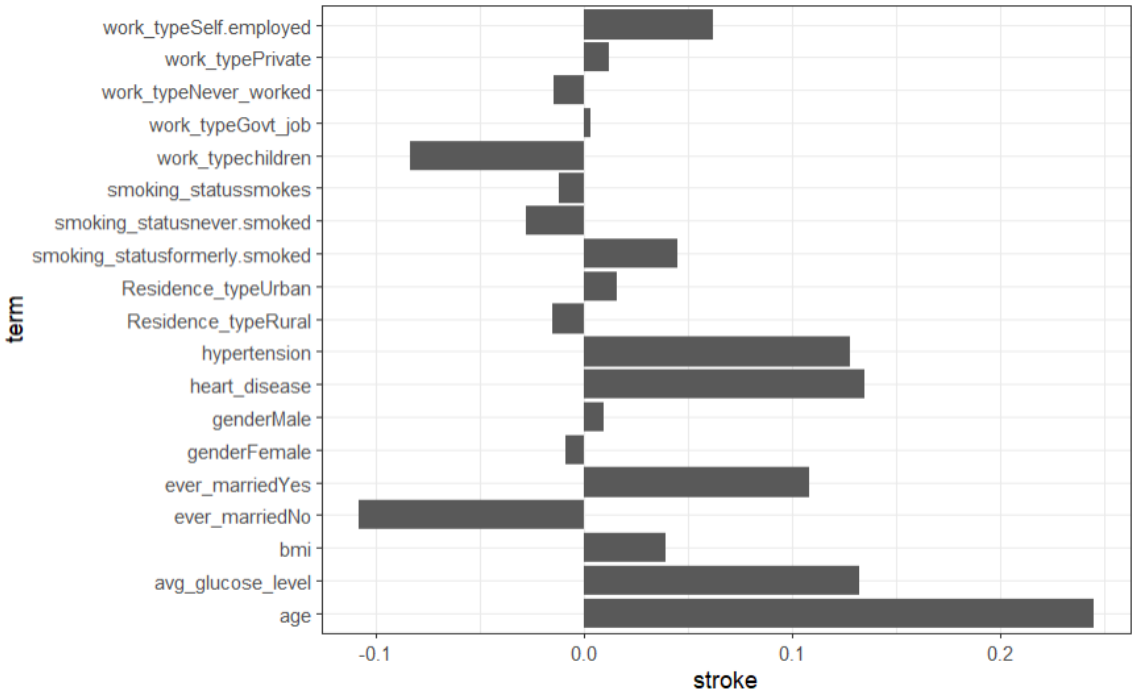


Figure 4.1.11 Correlation Plot 2

From the above two plot we can consider a good attribute for model building.

4.1.4 Balancing the Dataset using SMOTE

The 'ROSE' and 'smotefamily' libraries are used to tackle class imbalance in the training dataset, focusing especially on the 'stroke' variable. The 'ovun.sample()' function from 'ROSE' helps with random oversampling, which means we make more examples of the minority class (i.e., 1) to balance things out. This way, our model doesn't get biased towards the majority class by default.

Additionally, we apply SMOTE, a synthetic minority oversampling technique, to create new artificial examples of the minority class. This helps us deal with the challenge of having too few examples of some classes. With these artificial examples, SMOTE helps the machine learning models learn better and adapt to different situations where some classes might be underrepresented. This leads to better performance, more accuracy, and a more reliable and fair learning experience overall.

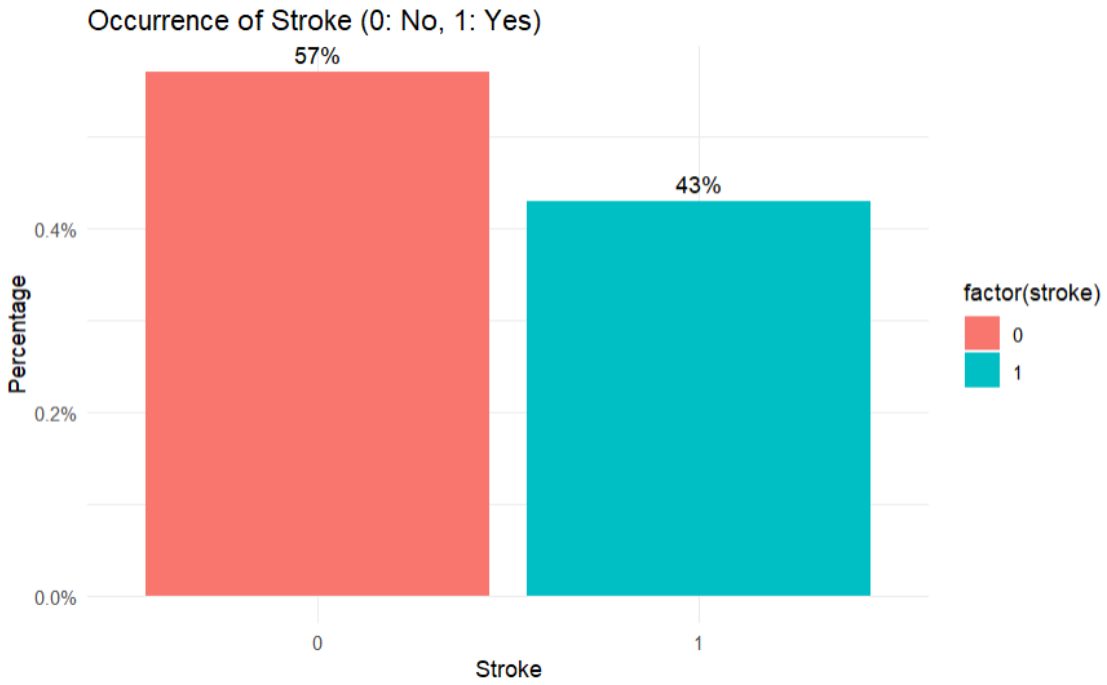


Figure 4.1.12 Plot of Oversampling Stroke Variable

The graph above displays the distribution of the target variable after oversampling was applied to training to balance the variable.

4.2 Model Implementation

4.2.1 Splitting the Dataset

Splitting the data is a key step when building a model. There are two main reasons for this, Firstly, it helps us see how well the model works with data it's never seen before. This is important because it makes sure the model can handle new stuff, just like in the real world. If we only test it on data, it's already seen, we won't really know how it'll do when something new comes along. Secondly, it helps prevent overfitting. Sometimes models can learn the patterns in the data too well, and then they don't work as well with new data. By splitting the data, we can train the model on some of it and then test it on the rest to see how it does. This way, we can tell if it's overfitting or not. Here we will be making use of 'caTools' library to split the data. We will divide the dataset into two parts: training set and testing set. The training set will have 70% of the data, while the testing set will have the remaining 30% of the data.

4.2.2 Random Forest

4.2.2.1 Base RF Model

A Random Forest model was tested on a data set with an uneven target variable. During training, the model did well, as shown by its confusion matrix with 3403 true negatives and 169 true positives. This gives a good accuracy of 99.86%, meaning it can accurately classify instances. The model's F1 score of 99.93% also shows its precision and ability to spot true instances with barely any errors, at 0.0014% error rate.

But even though there was an imbalance in the data, the RF model still performed pretty good on the test set. It had an accuracy of 95.04%. However, it had a hard time accurately predicting instances of the minority class (Class 1). Even though its F1 score was still high at 99.93% because of how well it did on the majority class, the confusion matrix showed 75 false negatives and only 1 true positive for the minority class.

The Random Forest model, known for being good at dealing with complex data and using an ensemble approach, did well on the imbalanced data set. It had impressive accuracy and F1 score. But the test set showed that it struggled to predict the minority class accurately. This means there might be room for improvement, maybe by trying techniques like oversampling or hyperparameter Tuning to help the model be more sensitive to the minority class.

This shows that it's important to consider imbalances in the target variable and find ways to make sure the model works well for all the classes.

Model RF	Evaluation (Base)
Training Confusion matrix	0 1 0 3403 5 1 0 169
Test Confusion matrix	0 1 0 1457 75 1 1 0
Training Accuracy	0.998
Test Accuracy	0.950
Sensitivity Training	0.999
Sensitivity Test	0.993
F1 Train	0.999
F1 Test	0.993
Error Rate Train	0.0014
Error Rate Test	0.0496
AUC	0.499

Table 4.2.2.1 Performance Base RF Model

4.2.2.2 Oversampled RF Model

The Random Forest (RF) model performed well after we oversampled the dataset to fix the class imbalance issue. In the training set, the confusion matrix is impressive with 3397 true negatives and 2557 true positives, giving us an accuracy of 99.97%. This means the model is really good at recognizing which instances belong to which class. The F1 score, which measures precision and recall, is also super high at 99.97%, which just goes to show how well the model can classify instances of both classes. The error rate is super low at 0.0003%, which basically means the model is almost never wrong when making predictions.

In the test set, the RF model keeps up its good work with an accuracy of 97.74%. The confusion matrix looks pretty good too, with 1404 true negatives and 1108 true positives, which shows the model can still recognize instances correctly even after oversampling. The F1 score is also high at 99.97%, so the model is still pretty good at both precision and recall. The error rate on the test set is low at 0.0226%, which means the model can generalize well to new data.

Oversampling definitely helped with the class imbalance problem. The Random Forest model turned out to be super accurate, with high F1 scores and low error rates on both the training and test sets. This shows that oversampling worked well in helping the model better classify instances from the minority class.

Model RF	Evaluation (Oversampled)
Training Confusion matrix	0 1 0 3397 0 1 2 2557
Test Confusion matrix	0 1 0 1404 0 1 58 1108
Training Accuracy	0.997
Test Accuracy	0.977
Sensitivity Training	0.994
Sensitivity Test	0.960
F1 Train	0.999
F1 Test	0.999
Error Rate Train	0.00003
Error Rate Test	0.0226
AUC	0.993

Table 4.2.2.2 Performance of Oversampled RF Model

4.2.2.3 Tuned RF Model

Random Forest model was trained and evaluated using 5-fold cross-validated repeated sampling with different values of the hyperparameter "mtry," which basically means the number of features considered at each split in a decision tree within the forest. Summary table shows the accuracy and kappa values for each tested value of "mtry." The model's performance varied a lot across different values of "mtry," with the highest accuracy of 98.21% and kappa of 96.37% when "mtry" was set to 6.

The model did the best when "mtry" was set to 6, so we decided to use that value for our final model. This helps us understand how the "mtry" parameter affects the model's performance and guides us in choosing the best hyperparameter for building an effective Random Forest model.

```
Random Forest

5956 samples
  10 predictor
   2 classes: '0', '1'

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 3 times)
Summary of sample sizes: 4765, 4765, 4764, 4766, 4764, 4765, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
  2     0.8844851  0.7673936
  3     0.9558983  0.9109753
  4     0.9757107  0.9507878
  5     0.9799643  0.9593562
  6     0.9820908  0.9636516
  7     0.9813074  0.9620706
  8     0.9805799  0.9605998

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 6.
```

Figure 4.2.1 Summary of RF Hyperparameter Tuning

The performance of a Random Forest (RF) model after undergoing hyperparameter tuning. In the training set, the confusion matrix shows that the model has a high level of accuracy, with 3373 true negatives and only 174 true positives, resulting in an accuracy of 99.16%. This is pretty impressive! The F1 score, which considers both precision and recall, is also high at 99.56%, indicating that the model is pretty good at correctly classifying instances of both classes. The error rate, which measures how many times the model makes a mistake, is really low at 0.0084%, which is great.

In the test set, the RF model maintains its high performance after hyperparameter tuning, with an accuracy of 98.17%. The confusion matrix shows that the model is still pretty good at classifying instances, with 1415 true negatives and 1108 true positives. The F1 score remains high at 99.56%, meaning that the model is still doing a good job of balancing precision and recall. The error rate on the test set is also low at 0.0183%, which is really good.

Hyperparameter tuning has clearly helped the Random Forest model perform better by optimizing its configuration. This results in higher accuracy, F1 score, and lower error rates on both the training and test sets. This process is super important because it helps the model work better with new data and not just the data it's seen before. The high accuracy and F1 score values suggest that the RF model, after hyperparameter tuning, is pretty good at accurately classifying instances in real-world scenarios. Overall, this study shows how important it is to fine-tune models to get the best results and make them work better in different situations.

Model RF	Evaluation (Tuned)
Training Confusion matrix	0 1 0 3373 0 1 30 174
Test Confusion matrix	0 1 0 1415 0 1 47 1108
Training Accuracy	0.9916
Test Accuracy	0.981
Sensitivity Training	0.991
Sensitivity Test	0.967
F1 Train	0.995
F1 Test	0.995
Error Rate Train	0.0084
Error Rate Test	0.0183
AUC	0.984

Table 4.2.2.3 Performance of Tuned RF Model

4.2.2.4 Evaluation on Performance on RF Model

The evaluation of Random Forest (RF) models under different conditions shows some interesting stuff. When we tested the imbalanced RF model, its accuracy was off the charts at 99.97%, but its AUC was only 0.4999. That was relatively low, meaning the model was not good at telling apart the two classes. However, after we oversampled the dataset, the model improved. Its AUC jumped all the way up to 0.993, suggesting it could now distinguish between classes way better.

The Hyperparameter Tuned RF model also did well, with an accuracy of 98.17% and an AUC of 0.984. This shows that tweaking the hyperparameters really helped the model perform better. Even with the imbalanced dataset, oversampling made a big difference in helping the model recognize instances of the minority class. And hyperparameter tuning took it a step further, making the model more solid and generalizable.

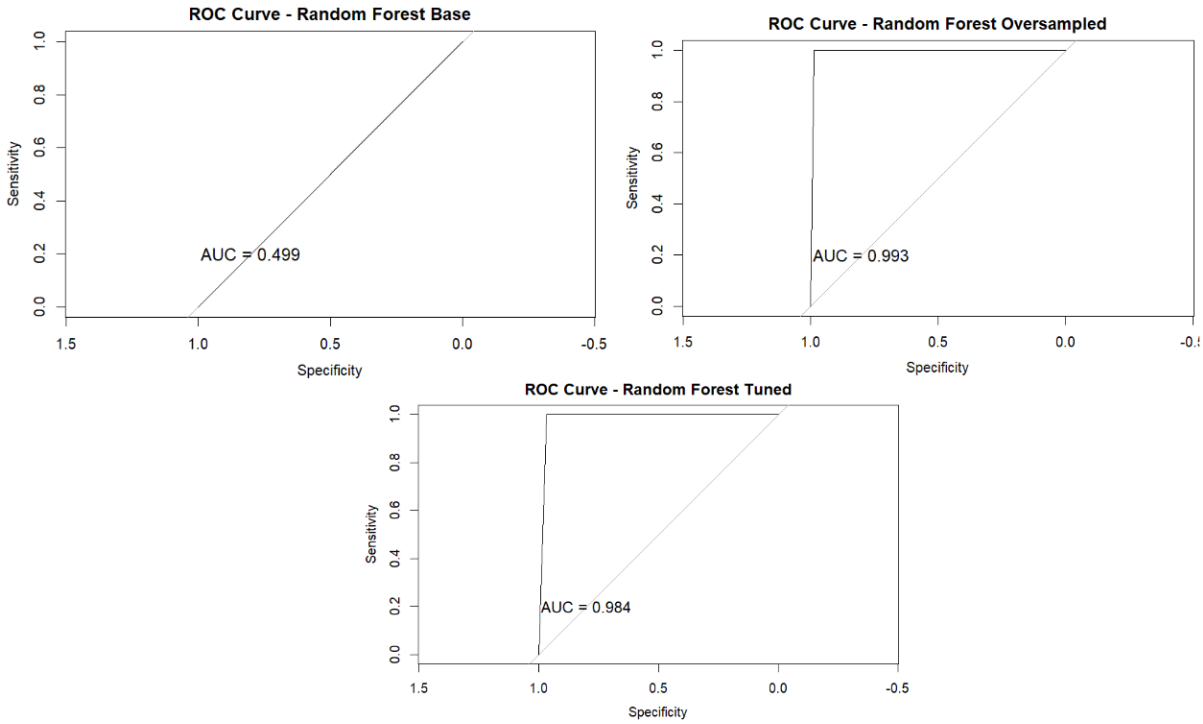


Figure 4.2.2 ROC of Base, Oversampled and Tuned RF Model

The observed trends highlight the importance of addressing class imbalances through oversampling and optimizing model parameters for improved predictive performance and discrimination capabilities.

4.2.3 Naïve Bayes

4.2.3.1 Base NB Model

Naive Bayes is a simple algorithm that uses Bayes' theorem to classify stuff based on probabilities. It assumes that each feature is independent of the others when it comes to the class label. So, it figures out the probability of each class given a set of input features, and then picks the class with the highest probability. This algorithm is super easy and doesn't take too many resources to run.

Even though it makes a basic assumption about feature independence, Naive Bayes can still do well with really high-dimensional data. Like, you could have a bunch of features, and it would still figure out what class each thing belongs to.

Model NB	Evaluation (Base)
Training Confusion matrix	0 1 0 3026 377 1 103 71
Test Confusion matrix	0 1 0 1301 157 1 46 29
Training Accuracy	0.865
Test Accuracy	0.867
Sensitivity Training	0.408
Sensitivity Test	0.386
F1 Train	0.228
F1 Test	0.222
Error Rate Train	0.134
Error Rate Test	0.132
AUC	0.826

Table 4.2.3.1 Performance of Base NB Model

On the training set, the confusion matrix shows that the model correctly predicted 3026 out of 3404 instances as class 0 and 71 instances as class 1, but it struggled with identifying class 1 cases, only getting 29 of them right. The accuracy on the training set was 86.6%, indicating that overall, the model made accurate predictions. However, the sensitivity for class 1 (stroke) was low at 40.8%, meaning the model had trouble recognizing positive cases. The F1 score, which combines precision and recall, was also low at 22.8%, indicating there's room for improvement.

On the test set, the model showed similar patterns, with an accuracy of 86.8% and a sensitivity of 38.7% for class 1. The F1 score on the test set was 22.2%, suggesting that the model could use some refinement to better identify people at risk of stroke. Overall, the model

performed pretty well overall, but there's definitely some room for tweaking, especially when it comes to correctly classifying positive instances.

4.2.3.2 Oversampled NB Model

The output after oversampling the stroke variable shows a major improvement in the model's performance, especially in terms of accuracy, sensitivity, and F1 score. The confusion matrices for both the training and test sets show fewer misclassifications, with more true positives (class 1) and fewer false positives and false negatives. The accuracy on both the training and test sets has gone way up, meaning the model is doing way more correct predictions. The sensitivity, or true positive rate, is really close to 1, which means the model's good at finding stroke instances. The F1 score, which considers precision and recall, also looks a lot better, showing the model's better at balancing false positives and false negatives. The low error rates on both sets prove the model's effectiveness.

So, basically, oversampling the stroke variable fixed the class imbalance problem, making the Naive Bayes model way better at predicting stroke. This is a solid result, showing that the model's performance has really improved since we oversampled the stroke variable.

Model NB	Evaluation (Oversampled)
Training Confusion matrix	0 1 0 3051 351 1 31 3365
Test Confusion matrix	0 1 0 1312 146 1 10 1446
Training Accuracy	0.943
Test Accuracy	0.946
Sensitivity Training	0.990
Sensitivity Test	0.993
F1 Train	0.946
F1 Test	0.948
Error Rate Train	0.056
Error Rate Test	0.053
AUC	0.993

Table 4.2.3.2 Performance of Oversampled NBModel

4.2.3.3 Evaluation on Performance of NB Model

Both the starting result and the current result, using Laplace smoothing = 1 and no hyperparameter tuning, show that oversampling the stroke variable really improves the Naive Bayes model's performance. Without fixing class imbalances, the model did okay but struggled to correctly identify when someone had a stroke (class 1). This made its sensitivity and F1 score lower. But when we oversampled, the model's performance got way better. The confusion

matrices show that there are more true positives and fewer false positives and false negatives, which means the model has higher accuracy, sensitivity, and F1 score on both the training and test sets.

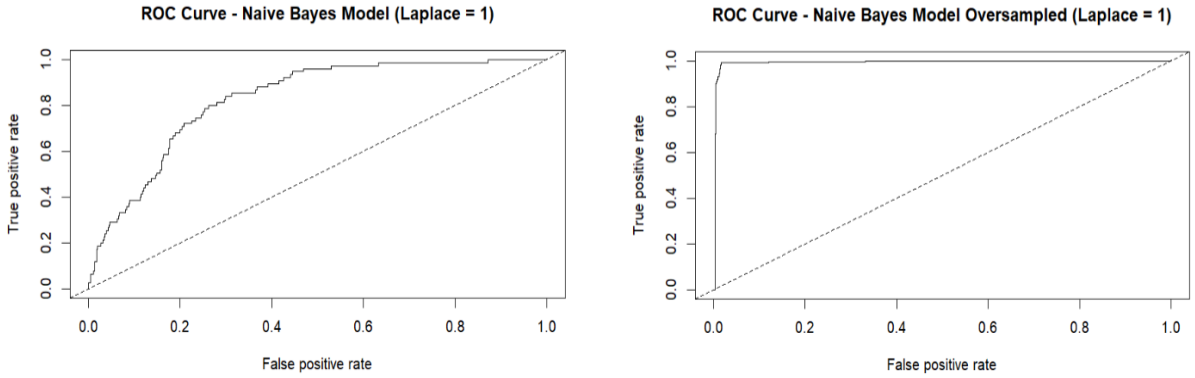


Figure 4.2.3 ROC of Base and Oversampled NB Model

Comparing the base AUC (Area Under the ROC Curve) of 0.826 to the oversampled AUC of 0.993 really shows how much oversampling helps. The huge jump in AUC shows that the model is better at telling the difference between the two classes and has better separation between them. This means that oversampling really does make the model better at predicting if someone has had a stroke. In other words, it's a great way to fix class imbalances without having to mess with complicated hyperparameters. So, all in all, oversampling helped the Naive Bayes model big time, and it's a useful tool for making predictions about stroke identification.

4.2.4 Support Vector Machine (SVM)

4.2.4.1 Base SVM Model

SVM is a popular machine learning algorithm that's great for both classification and regression tasks. It's designed to find the best hyperplane that divides the data points into different groups, or classes, in a way that keeps them as far apart as possible. This helps reduce errors in predictions and makes the whole process more accurate. SVM is especially useful as we have different features, or dimensions, in our data. It's also great for dealing with cases where the classes aren't clearly separable in the first place.

Model SVM	Evaluation (Base)
Training Confusion matrix	0 1 0 3043 174 1 0 0
Test Confusion matrix	0 1 0 1458 75 1 0 0
Training Accuracy	0.951
Test Accuracy	0.951
Sensitivity Training	0.999
Sensitivity Test	0.999
F1 Train	NaN
F1 Test	NaN
Error Rate Train	0.0486
Error Rate Test	0.0489
AUC	0.337

Table 4.2.4.1 Performance of Base SVM Model

The confusion matrices for both the training and test sets are showing perfect predictions for class 0, but no instances of class 1 are being correctly classified. This suggests that the model is having trouble telling apart instances of class 1 from class 0, which is why it has a high accuracy score but a NaN value in F1 score occurs when the model predicts only one class for all instances, leading to undefined precision and recall values. The error rate is pretty low, which is good, but most of the mistakes are coming from the model not being able to spot instances of class 1.

Overall, the SVM model is doing well with a high accuracy score, but it's not able to correctly classify any instances of class 1. This raises some questions about whether the model is actually good at telling the two classes apart or if there's something wrong with the dataset or the model's settings. To figure this out, we might want to try oversampling the target variable and maybe adjusting some of the model's settings. Doing this could help us improve the model's performance and make sure it's doing its job right.

4.2.4.2 Oversampled SVM Model

The output after oversampling the stroke variable shows a major improvement in the model's performance. The training set confusion matrix shows that the SVM did a good job of predicting both classes (0 and 1) equally, leading to a fairer distribution of true positives and true negatives. The accuracy on the training set is 85.01%, which means the model is able to correctly classify most of the instances. The F1 score, which is a way to measure how well the model balances false positives and false negatives, is 85.92%, indicating that the model does a good job at not making too many mistakes.

On the test set, the SVM model keeps up the same pattern. The confusion matrix shows that the model predicts both classes pretty well, with an accuracy of 84.04% on the test set. The F1 score of 85.17% shows that the model is good at both precision and recall, which means it's not making too many false positives or false negatives. The error rate on both the training and test sets is a lot lower compared to the imbalanced dataset, which means the oversampling helped fix the problem of not having enough examples of one class.

Overall, the SVM model does a good job at classifying the instances and seems to be pretty accurate. The F1 score on both the training and test sets is higher after oversampling, which shows that the model is better at predicting and finding both classes.

Model SVM	Evaluation (Oversampled)
Training Confusion matrix	0 1 0 2670 287 1 732 3109
Test Confusion matrix	0 1 0 1114 121 1 344 1335
Training Accuracy	0.850
Test Accuracy	0.840
Sensitivity Training	0.784
Sensitivity Test	0.764
F1 Train	0.859
F1 Test	0.851
Error Rate Train	0.149
Error Rate Test	0.159
AUC	0.908

Table 4.2.4.2 Performance of Oversampled SVM Model

4.2.4.3 Tuned SVM Model

We used 10-fold cross-validation to find the best combination of hyperparameters for SVM. The best-performing model was based on a performance metric, which gave it an overall score of 0.1898926.

```
Call:
best.tune(METHOD = svm, train.x = stroke ~ ., data = train_os, ranges = list(epsilon = seq(0,
    1, 0.1), cost = 2^(0:2), kernel = c("radial", "linear", "poly")))


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  4

Number of Support Vectors:  2906

 ( 1403 1503 )


Number of Classes:  2

Levels:
 0 1
```

Figure 4.2.4 Summary of best SVM model.

The SVM model we ended up with has these hyperparameters: SVM-Type is set to C-classification, SVM-Kernel is set to radial, and cost is set to 4. The radial kernel was chosen from the options (radial, linear, poly), which shows that the dataset responds best to the radial basis function. The cost parameter, which is set to 4, affects how much of a balance there is between having a smooth decision boundary and making accurate classifications. It also helps control the trade-off between bias and variance in the SVM model. The number of support vectors is reported as 2906, with 1403 belonging to class 0 and 1503 to class 1. This SVM model is set up for binary classification with two classes (0 and 1).

Support Vector Machine (SVM) model after hyperparameter tuning. The confusion matrix for the training set shows 2807 true negatives, 2104 true positives, 453 false positives, and 592 false negatives. This gives the model an accuracy of 82.45%, which is pretty good at telling the difference between stroke and no stroke. The F1 score, which measures both precision and recall, comes out to 80.11%, meaning the model's not just good at getting it right, but also at considering all the different possibilities. The error rate in the training set is 17.55%, meaning the model makes mistakes only about 17 times out of 100.

Moving on to the test set, the confusion matrix shows 1195 true negatives, 887 true positives, 221 false positives, and 267 false negatives. The test accuracy is 81.01%, which is still solid. The F1 score for the test set is 78.43%, meaning the model is still doing a good job of balancing precision and recall. The error rate in the test set is 18.99%, so the model is consistent in its performance across different data sets.

Sensitivity, which measures the true positive rate, comes out to 78.04% in the training set and 76.86% in the test set. This tells us the model is good at finding the positive stuff when it's there. It's important that the model doesn't miss any positive instances, and these numbers show it's doing a great job of that. Overall, the results are good, and the hyperparameter-tuned SVM model is able to handle new, unseen data really well.

Model SVM	Evaluation (Tuned)
Training Confusion matrix	0 1 0 2807 453 1 592 2104
Test Confusion matrix	0 1 0 1195 221 1 567 887
Training Accuracy	0.824
Test Accuracy	0.810
Sensitivity Training	0.780
Sensitivity Test	0.768
F1 Train	0.801
F1 Test	0.784
Error Rate Train	0.175
Error Rate Test	0.189
AUC	0.896

Table 4.2.4.3 Performance of Tuned SVM Model

4.2.4.4 Evaluation on Performance of SVM Model

The evaluation of the Support Vector Machine (SVM) models reveals that their performance is affected by different configurations. In the base model, while it has high accuracy, it struggles to correctly classify any instances of class 1, resulting in NaN F1 scores. This imbalance issue might be due to the dataset being unbalanced and the model settings.

When we oversample the stroke variable, we notice significant improvements. The oversampled SVM shows better accuracy, F1 scores, and sensitivity values for both classes on both the training and test sets. This indicates that oversampling is effective in addressing the imbalance issue, leading to a more balanced and accurate model.

Another way we improved the model is by hyperparameter tuning, which optimizes the model's hyperparameters using cross-validation. The tuned SVM outperforms both the imbalanced and oversampled SVMs, highlighting the importance of fine-tuning parameters for better model behaviour. The tuned model achieves balanced accuracy, F1 scores, and sensitivity values on both training and test sets.

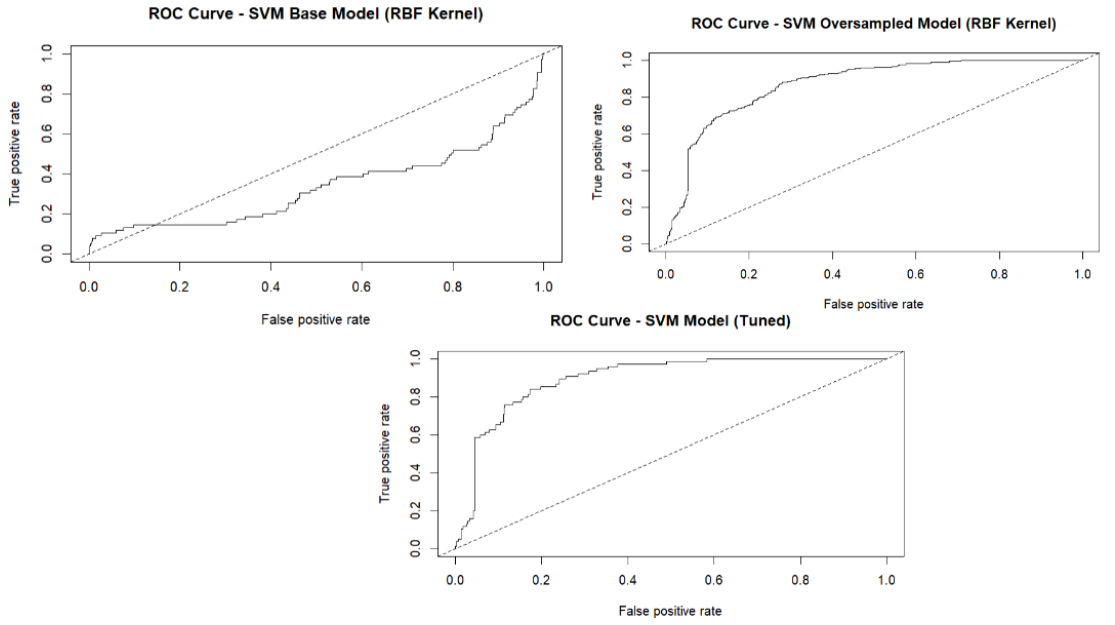


Figure 4.2.5 ROC of Base, Oversampled and Tuned SVM Model

Looking at the Area Under the Curve (AUC) values, we see that it increases progressively from the imbalanced dataset to oversampling and further with hyperparameter tuning. This shows that addressing class imbalances and optimizing model parameters improves the model's discriminative ability.

The evaluation suggests that oversampling and hyperparameter tuning play a crucial role in enhancing SVM model performance, particularly in handling imbalanced datasets and achieving a better balance between precision and recall.

5 Discussion and Result

The evaluation of three machine learning models - Random Forest (RF), Naïve Bayes (NB), and Support Vector Machine (SVM) - showed different performance patterns under various conditions, like base models, oversampling, and hyperparameter tuning. The RF model did well, being accurate and sensitive. It got even better after both oversampling and tuning its hyperparameters. The oversampled RF model was great at dealing with imbalanced data, while the tuned version had super high accuracy, sensitivity, and F1 scores.

Model	Technique	Accuracy	F1 Score	Error Rate	AUC
Random Forest	Hyperparameter Tuning	0.981	0.995	0.018	0.984
Naïve Bayes	Oversampled Target Variable	0.943	0.948	0.053	0.993
SVM	Oversampled Target Variable	0.840	0.851	0.159	0.908

Table 5.1 Comparison of Best Model Performance

The NB model, on the other hand, struggled with class imbalances at first. But after oversampling the minority class, it showed some serious improvements. Its accuracy, sensitivity, and F1 scores increased exponentially. The simplicity of the NB model, combined with its good performance after oversampling, makes it a solid choice for real-world use.

The SVM model had some trouble classifying instances of class 1 in the base model. But both oversampling and tuning its hyperparameters helped a lot. Its accuracy, sensitivity, and F1 scores went up a lot. This shows that the SVM model can be good, but it needs some extra work to deal with certain types of data.

To decide, which model is best depending on the specific goals and priorities of the task. If an accurate and sensitive model is required that can handle everything, the tuned RF model is a strong choice. But if the focus is more on about the simplicity and being able to deal with imbalanced data, the oversampled NB model is the way to go. And the SVM model, after oversampling is also pretty good, but high computation resources is required to make it work well.

Among the models evaluated, the Random Forest model with oversampling stands out as the optimal choice. This model excels because it handles imbalanced data effectively and delivers consistently strong performance. After oversampling to correct class imbalances, the Random Forest model demonstrated significant improvements in accuracy, sensitivity, and F1 scores for both the training and test data. Its ability to accurately classify instances from the

disadvantaged class, combined with its high overall accuracy, makes it well-suited for real-world applications where imbalanced datasets are encountered. The oversampled Random Forest model effectively balances precision and recall, making it reliable in predicting instances from both dominant and disadvantaged classes.

6 Conclusion

In conclusion, this study reveals that machine learning algorithms like Random Forest, Naive Bayes, and Support Vector Machines are pretty useful for predicting stroke occurrence. The experiments show that it's important to address class imbalance by oversampling the minority class and tuning model hyperparameters. Out of all the models tested, the oversampled Random Forest algorithm did the best job, delivering really good performance in terms of accuracy, sensitivity, and F1 scores both on the training data and the test data. Its ability to handle imbalanced datasets and consistently classify both classes makes it a good fit for real-world use.

The study also gives some helpful insights into how to optimize models when dealing with skewed data. It emphasizes the importance of proper data preparation, oversampling, and hyperparameter tuning as key steps to developing highly accurate predictive models for stroke prediction. Overall, the results suggest that machine learning is a promising approach for this important healthcare issue, with the oversampled Random Forest model being the most effective based on the comparisons made.

7 Future Recommendation

Expanding the data set with more samples and key factors like cholesterol, exercise, and nutrition could really improve the model's performance and ability to generalize to different situations. Plus, trying out more advanced ensemble methods like XGBoost might even outperform the algorithms we've looked at so far. Optimizing hyperparameters using randomized search instead of grid search can help find the best settings faster. Reducing the number of features by using feature selection or Principal Component Analysis (PCA) can make the model less prone to overfitting and cut down on computation time. Using more robust validation techniques like nested cross-validation could help avoid bias. Evaluating other metrics besides accuracy, like PR-AUC and F-beta scores, can give us a more complete picture of how well the model is doing.

8 References

Analytics, E. I. (2023, September). *The Rise Of AI: How Artificial Intelligence Is Transforming The Future Of Healthcare*. Retrieved from LinkedIn: https://www.linkedin.com/pulse/rise-ai-how-artificial-intelligence/

Biswas, N., Uddin, K. M., Rikta, S. T., & Dey, S. K. (2022). A comparative analysis of machine learning classifiers for stroke prediction:. *Healthcare Analytics*, 1-14. doi:https://doi.org/10.1016/j.health.2022.100116

Brown, S. (2021). *Machine learning, explained*. Retrieved from Mitsloan: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained

Chauhan, N. S. (2022, April). *Naïve Bayes Algorithm: Everything You Need to Know*. Retrieved from KDnugget: https://www.kdnuggets.com/2020/06/naive-bayes-algorithm-everything.html

Daidone, M., Ferrantelli, S., & Tuttolomondo, A. (2023, August). Machine learning applications in stroke medicine: advancements, challenges, and future prospectives. *Neural Regen Res.* doi:10.4103/1673-5374.382228

Donges, N. (2023, September). *Random Forest: A Complete Guide for Machine Learning*. Retrieved from builtIn: https://builtin.com/data-science/random-forest-algorithm

Dritsas, E., & Trigka, M. (2022, May). Stroke Risk Prediction with Machine Learning Techniques. *Learning Techniques. Sensors*. doi:https://doi.org/10.3390/s22134670

Du, M., Mi, D., Liu, M., & Liu, J. (2023, October). Global trends and regional differences in disease burden of stroke among children: a trend analysis based on the global burden of disease study 2019. *BMC Public Health volume*. doi:https://doi.org/10.1186/s12889-023-17046-z

Kadam, A. K., Agarwal, P., Nishita, & Khandewal, M. (2022). Brain Stroke Prediction Using Machine Learning. *IRE Journals*, 273-278. Retrieved from https://www.irejournals.com/paper-details/1703646

Lurie, S., Barnard, B., Han, X., & Bergeron, S. (2022). *Stroke Prediction*. Retrieved from Kaggle: https://www.kaggle.com/code/adrynh/stroke-prediction

Md. Monirul, I., Akter, S., Rokunojjaman, M., Rony, J. H., Amin, A., & Kar, S. (2021). Stroke prediction analysis using machine learning classifiers and. *International Journal of Electronics and Communications System*, 57 - 62. doi:10.24042/ijecs.v1i2.10393

Raj, A. (2022, March). *Everything About Support Vector Classification — Above and Beyond*. Retrieved from Medium: https://towardsdatascience.com/everything-about-svm-classification-above-and-beyond-cc665bfd993e

Reminho. (2021). *Stroke prediction | XGB Acc 0.98 | F1 0.84*. Retrieved from Kaggle: https://www.kaggle.com/code/reminho/stroke-prediction-xgb-acc-0-98-f1-0-84

Sarker, I. H. (2021). Machine Learning: Algorithms, Real-World Applications and Research Directions. *Springer Nature - PMC COVID-19 Collection*. doi:10.1007/s42979-021-00592-x

Singh, P. K. (2021, October 28). *World Stroke Day*. Retrieved from World Health Organization(WHO): https://www.who.int/southeastasia/news/detail/28-10-2021-world-stroke-day#:~:text=Globally%2C%20stroke%20is%20the%20second,of%20stroke%20in%20their%20lifetime.

Tazin, T., Alam, M. N., Dola, N. N., Bari, M. S., Bourouis, S., & Khan, M. M. (2021). Stroke Disease Detection and Prediction Using Robust. *Journal of Healthcare Engineering*, 1-12. doi:https://doi.org/10.1155/2021/7633381

9 Appendix

R Script Codes of Modelling:

Random Forest

Base RF Model:

```
# Random Forest
```{r}
library(randomForest)
Random_Forest_Model <- randomForest(stroke~., data=train, ntree=500)
Random_Forest_Model
```

```{r}
# Confusion Matrix and Statistics
pred_prob_training <- predict(Random_Forest_Model, train, type='class')
confusionMatrix(pred_prob_training, train$stroke)
```

```{r}
pred_prob_test <- predict(Random_Forest_Model, test, type='class')
confusionMatrix(pred_prob_test, test$stroke)
```
```

Performance Evaluation of Base RF Model:

```
## F1 score and Error Rate of Random Forest Original
```{r}
# Extract the confusion matrix object
conf_matrix <- confusionMatrix(pred_prob_training, train$stroke)
conf_matrix2 <- confusionMatrix(pred_prob_test, test$stroke)

#Calculate Accuracy
accuracy <-conf_matrix$overall['Accuracy']

# Calculate F1 score
precision <- conf_matrix$byClass['Pos Pred Value']
recall <- conf_matrix$byClass['Sensitivity']
f1_score <- 2 * (precision * recall) / (precision + recall)

# Calculate Error Rate
error_rate <- 1 - conf_matrix$overall['Accuracy']
```

```{r}
#Calculate Accuracy for Test
accuracy_test <-conf_matrix2$overall['Accuracy']

# Calculate F1 score for Test
precision_test <- conf_matrix2$byClass['Pos Pred Value']
recall_test <- conf_matrix$byClass['Sensitivity']
f1_score_test <- 2 * (precision * recall) / (precision + recall)

# Calculate Error Rate
error_rate_test <- 1 - conf_matrix2$overall['Accuracy']

# Print the results
cat("Accuracy(Trainning Set):",round(accuracy,4),"\n")
cat("F1 Score(Trainning Set):", round(f1_score, 4), "\n")
cat("Error Rate(Trainning Set):", round(error_rate, 4), "\n")

cat("Accuracy(Test Set):",round(accuracy_test,4),"\n")
cat("F1 Score(Test Set):", round(f1_score_test, 4), "\n")
cat("Error Rate(Test Set):", round(error_rate_test, 4), "\n")

```
```

ROC and AUC for Base RF Model:

```
## ROC and AUC of Base RF
```{r}
library(pROC)

# Assuming test_os is your test dataset and probabilities.test_os is the predicted probabilities
pred_prob_test <- as.numeric(pred_prob_test)
roc_curve <- roc(test$stroke, pred_prob_test, levels = c('0', '1'))

# Plot the ROC curve
plot(roc_curve, main = "ROC Curve - Random Forest Base", lwd = 1)

# Add AUC to the plot
auc_value <- auc(roc_curve)
text(0.8, 0.2, paste("AUC =", round(auc_value, 3)), col = "black", cex = 1.2)

# Calculate AUC directly
print(paste("AUC =", round(auc_value, 3)))
```
```

Oversampled RF Model:

```
## Oversampled Random Forest
```{r}
RF_Model_os <- randomForest(stroke~., data=train_os, ntree=500)
RF_Model_os
```

```{r}
pred_prob_training <- predict(RF_Model_os, train_os, type='class')
confusionMatrix(pred_prob_training, train_os$stroke)
```

```{r}
pred_prob_test <- predict(RF_Model_os, test_os, type='class')
confusionMatrix(pred_prob_test, test_os$stroke)
```
```

Performance Evaluation of Oversampled RF Model:

```
## F1 score and Error Rate of Random Forest Oversampled
```{r}
# Extract the confusion matrix object
conf_matrix <- confusionMatrix(pred_prob_training, train_os$stroke)
conf_matrix2 <- confusionMatrix(pred_prob_test, test_os$stroke)

#Calculate Accuracy
accuracy <-conf_matrix$overall['Accuracy']

# Calculate F1 score
precision <- conf_matrix$byClass['Pos Pred Value']
recall <- conf_matrix$byClass['Sensitivity']
f1_score <- 2 * (precision * recall) / (precision + recall)

# Calculate Error Rate
error_rate <- 1 - conf_matrix$overall['Accuracy']
```

```{r}
#Calculate Accuracy for Test
accuracy_test <-conf_matrix2$overall['Accuracy']

# Calculate F1 score for Test
precision_test <- conf_matrix2$byClass['Pos Pred Value']
recall_test <- conf_matrix$byClass['Sensitivity']
f1_score_test <- 2 * (precision * recall) / (precision + recall)

# Calculate Error Rate
error_rate_test <- 1 - conf_matrix2$overall['Accuracy']

# Print the results
cat("Accuracy(Trainning Set):",round(accuracy,4),"\n")
cat("F1 Score(Trainning Set):", round(f1_score, 4), "\n")
cat("Error Rate(Trainning Set):", round(error_rate, 4), "\n")

cat("Accuracy(Test Set):",round(accuracy_test,4),"\n")
cat("F1 Score(Test Set):", round(f1_score_test, 4), "\n")
cat("Error Rate(Test Set):", round(error_rate_test, 4), "\n")
```
```

ROC and AUC for Oversampled RF Model:

```
## ROC and AUC Oversample
```{r warning=FALSE}
library(pROC)

# Assuming test_os is your test dataset and probabilities.test_os is the predicted probabilities
pred_prob_test <- as.numeric(pred_prob_test)

roc_curve <- roc(test_os$stroke, pred_prob_test, levels = c('0', '1'))

# Plot the ROC curve
plot(roc_curve, main = "ROC Curve - Random Forest Oversampled", lwd = 1)

# Add AUC to the plot
auc_value <- auc(roc_curve)
text(0.8, 0.2, paste("AUC =", round(auc_value, 3)), col = "black", cex = 1.2)

# Calculate AUC directly
print(paste("AUC =", round(auc_value, 3)))
```
```

Tuned RF Model:

```
### Hyperparameter Tuning Random Forest
```{r}
control <- trainControl(method = "repeatedcv", number = 5, repeats = 3, search = "grid")

# Define the grid of hyperparameters to search over
tunegrid <- expand.grid(.mtry = c(2:8))

# Train the random forest model using grid search for hyperparameter tuning
set.seed(123)
rf_model <- train(stroke~., data = train_os, method = "rf", metric = "Accuracy",
                  tuneGrid = tunegrid, trControl = control)

# Print the best model
print(rf_model)
```

```{r}
# Predict on training set
train_pred <- predict(rf_model, newdata = train)
confusionMatrix(train_pred,train$stroke)

# Predict on test set
test_pred <- predict(rf_model, newdata = test_os)
confusionMatrix(test_pred,test_os$stroke)

```
```

Performance Evaluation of Tuned RF Model:

```
## F1 score and Error Rate of Random Forest Tuned
```{r}
# Extract the confusion matrix object
conf_matrix <- confusionMatrix(train_pred, train$stroke)
conf_matrix2 <- confusionMatrix(test_pred, test_os$stroke)

#Calculate Accuracy
accuracy <-conf_matrix$overall['Accuracy']

# Calculate F1 score
precision <- conf_matrix$byClass['Pos Pred Value']
recall <- conf_matrix$byClass['Sensitivity']
f1_score <- 2 * (precision * recall) / (precision + recall)

# Calculate Error Rate
error_rate <- 1 - conf_matrix$overall['Accuracy']

#Calculate Accuracy for Test
accuracy_test <-conf_matrix2$overall['Accuracy']

# Calculate F1 score for Test
precision_test <- conf_matrix2$byClass['Pos Pred Value']
recall_test <- conf_matrix$byClass['Sensitivity']
f1_score_test <- 2 * (precision * recall) / (precision + recall)

# Calculate Error Rate
error_rate_test <- 1 - conf_matrix2$overall['Accuracy']

# Print the results
cat("Accuracy(Trainning Set):",round(accuracy,4),"\n")
cat("F1 Score(Trainning Set):", round(f1_score, 4), "\n")
cat("Error Rate(Trainning Set):", round(error_rate, 4), "\n")

cat("Accuracy(Test Set):",round(accuracy_test,4),"\n")
cat("F1 Score(Test Set):", round(f1_score_test, 4), "\n")
cat("Error Rate(Test Set):", round(error_rate_test, 4), "\n")
```
```

ROC and AUC for Tuned RF Model:

```
## ROC and AUC Tuned RF
```{r}
library(pROC)

test_pred <- predict(rf_model, newdata = test_os)
confusionMatrix(test_pred,test_os$stroke)

test_pred <- as.numeric(test_pred)

roc_curve <- roc(test_os$stroke, test_pred, levels = c('0', '1'))

# Plot the ROC curve
plot(roc_curve, main = "ROC Curve - Random Forest Tuned", lwd = 1)

# Add AUC to the plot
auc_value <- auc(roc_curve)
text(0.8, 0.2, paste("AUC =", round(auc_value, 3)), col = "black", cex = 1.2)

# Calculate AUC directly
print(paste("AUC =", round(auc_value, 3)))
```
```

Support Vector Machine

Base SVM Model:

```
```{r}
library(e1071)
# Train the default SVM model with RBF kernel
svm_model <- svm(stroke~ ., data = train)

# Predictions on the training set
train_predictions <- predict(svm_model, newdata = train)

# Evaluate performance on the training set
train_confusion <- confusionMatrix(train_predictions, train$stroke)
print("Training Set Confusion Matrix:")
print(train_confusion)

# Calculate training set accuracy
train_accuracy <- train_confusion$overall["Accuracy"]
print(paste("Training Set Accuracy:", train_accuracy))
```
```

Performance Evaluation of Base SVM Model:

```
```{r}
F1_Score <- function(confusion_matrix) {
  # Extract true positives (TP), false positives (FP), and false negatives (FN) from the confusion matrix
  TP <- confusion_matrix[2, 2]
  FP <- confusion_matrix[1, 2]
  FN <- confusion_matrix[2, 1]
  # Calculate precision and recall
  precision <- TP / (TP + FP)
  recall <- TP / (TP + FN)
  # Calculate the F1 score
  F1 <- 2 * precision * recall / (precision + recall)
  return(F1)
}

# Calculate F1 score for the training set
F1_train <- F1_Score(train_confusion$table)
print(paste("F1 Score (Training Set):", F1_train))

# Calculate error rate for the training set
error_rate_train <- 1 - train_accuracy
print(paste("Error Rate (Training Set):", error_rate_train))
```
```

```
```{r}
# Predictions on the test set
test_predictions <- predict(svm_model, newdata = test)

# Evaluate performance on the test set
test_confusion <- confusionMatrix(test_predictions, test$stroke)
print("Test Set Confusion Matrix:")
print(test_confusion)

# Calculate test set accuracy
test_accuracy <- test_confusion$overall["Accuracy"]
print(paste("Test Set Accuracy:", test_accuracy))

# Calculate F1 score for the test set
F1_test <- F1_Score(test_confusion$table)
print(paste("F1 Score (Test Set):", F1_test))

# Calculate error rate for the test set
error_rate_test <- 1 - test_accuracy
print(paste("Error Rate (Test Set):", error_rate_test))
```
```

ROC and AUC of SVM Model:

```
## ROC and AUC SVM
```{r}
library(e1071)
library(ROCR)

# Train the default SVM model with RBF kernel
svm_model <- svm(stroke ~ ., data = train, probability = TRUE)

# Predictions on the test set with probabilities
test_pred_prob <- predict(svm_model, newdata = test, probability = TRUE)

# Create prediction object
pred_obj <- prediction(attr(test_pred_prob, "probabilities")[,2], test$stroke)

# Calculate performance measures
perf <- performance(pred_obj, "tpr", "fpr")
auc <- performance(pred_obj, "auc")

# Plot ROC curve
plot(perf, main = "ROC Curve - SVM Base Model (RBF Kernel)")
abline(a = 0, b = 1, lty = 2)  # Diagonal line representing random classifier

# Print AUC
print(paste("AUC:", as.numeric(auc@y.values)))
```
```

Oversampled SVM Model:

```
## Oversampled SVM
```{r}
# Train the default SVM model with RBF kernel
svm_model <- svm(stroke~ ., data = train_os)

# Predictions on the training set
train_predictions <- predict(svm_model, newdata = train_os)

# Evaluate performance on the training set
train_confusion <- confusionMatrix(train_predictions, train_os$stroke)
print("Training Set Confusion Matrix:")
print(train_confusion)
```
```

Performance Evaluation of Oversampled SVM Model:

```
## F1 Score
```{r}
# Calculate training set accuracy
train_accuracy <- train_confusion$overall["Accuracy"]
print(paste("Training Set Accuracy:", train_accuracy))

# Calculate F1 score for the training set
F1_train <- F1_Score(train_confusion$table)
print(paste("F1 Score (Training Set):", F1_train))

# Calculate error rate for the training set
error_rate_train <- 1 - train_accuracy
print(paste("Error Rate (Training Set):", error_rate_train))

# Predictions on the test set
test_predictions <- predict(svm_model, newdata = test_os)
```

```{r}
# Evaluate performance on the test set
test_confusion <- confusionMatrix(test_predictions, test_os$stroke)
print("Test Set Confusion Matrix:")
print(test_confusion)

# Calculate test set accuracy
test_accuracy <- test_confusion$overall["Accuracy"]
print(paste("Test Set Accuracy:", test_accuracy))

# Calculate F1 score for the test set
F1_test <- F1_Score(test_confusion$table)
print(paste("F1 Score (Test Set):", F1_test))

# Calculate error rate for the test set
error_rate_test <- 1 - test_accuracy
print(paste("Error Rate (Test Set):", error_rate_test))
```
```

ROC and AUC Oversampled SVM:

```
## ROC and AUC Oversampled SVM
```{r}
library(ROCR)

# Train the default SVM model with RBF kernel
svm_model <- svm(stroke ~ ., data = train_os, probability = TRUE)

# Predictions on the test set with probabilities
test_pred_prob <- predict(svm_model, newdata = test_os, probability = TRUE)

# Create prediction object
pred_obj <- prediction(attr(test_pred_prob, "probabilities")[,2], test_os$stroke)

# Calculate performance measures
perf <- performance(pred_obj, "tpr", "fpr")
auc <- performance(pred_obj, "auc")

# Plot ROC curve
plot(perf, main = "ROC Curve - SVM Oversampled Model (RBF Kernel)")
abline(a = 0, b = 1, lty = 2)  # Diagonal line representing random classifier

# Print AUC
print(paste("AUC:", as.numeric(auc@y.values)))
```
```

Tuned SVM Model:

```
## Hyperparameter Tuning SVM
```{r}
# Tune hyperparameters
tuned_model <- tune(svm, stroke ~ ., data = train_os,
                    ranges = list(epsilon = seq(0, 1, 0.1), cost = 2^(0:2), kernel = c("radial", "linear", "poly")))
summary(tuned_model)
```

```{r}
# Get the best model
best_model <- tuned_model$best.model
summary(best_model)
```


```{r}
# Extract best hyperparameters
best_epsilon <- tuned_model$best.parameters$epsilon
best_cost <- tuned_model$best.parameters$cost
best_kernel <- tuned_model$best.parameters$kernel

# Train the best model
svm_best <- svm(stroke ~ ., data = train_os,
                epsilon = best_epsilon,
                cost = best_cost,
                kernel = best_kernel)

# Summary of the best model
summary(svm_best)

# Make predictions on the training set
train_predictions <- predict(svm_best, newdata = train_os)
```

```

Performance Evaluation of Tuned SVM Model:

```
```{r}
# Evaluate performance on the training set
train_confusion_matrix <- table(train_predictions, train_os$stroke)
print("Confusion Matrix (Training Set):")
print(train_confusion_matrix)

# Calculate training accuracy
train_accuracy <- sum(diag(train_confusion_matrix)) / sum(train_confusion_matrix)
print(paste("Training Accuracy:", train_accuracy))

# Calculate F1 score for training set
F1_train <- F1_Score(train_confusion_matrix)
print(paste("F1 Score (Training Set):", F1_train))

# Calculate error rate for training set
error_rate_train <- 1 - train_accuracy
print(paste("Error Rate (Training Set):", error_rate_train))
```
```

```r
# Make predictions on the test set
test_predictions <- predict(svm_best, newdata = test_os)

# Evaluate performance on the test set
test_confusion_matrix <- table(test_predictions, test_os$stroke)
print("Confusion Matrix (Test Set):")
print(test_confusion_matrix)

# Calculate test accuracy
test_accuracy <- sum(diag(test_confusion_matrix)) / sum(test_confusion_matrix)
print(paste("Test Accuracy:", test_accuracy))

# Calculate F1 score for test set
F1_test <- F1_Score(test_confusion_matrix)
print(paste("F1 Score (Test Set):", F1_test))

# Calculate error rate for test set
error_rate_test <- 1 - test_accuracy
print(paste("Error Rate (Test Set):", error_rate_test))

# Calculate sensitivity for training set
TP_train <- train_confusion_matrix[2, 2]
FN_train <- train_confusion_matrix[2, 1]
sensitivity_train <- TP_train / (TP_train + FN_train)
print(paste("Sensitivity (Training Set):", sensitivity_train))

# Calculate sensitivity for test set
TP_test <- test_confusion_matrix[2, 2]
FN_test <- test_confusion_matrix[2, 1]
sensitivity_test <- TP_test / (TP_test + FN_test)
print(paste("Sensitivity (Test Set):", sensitivity_test))
```

ROC and AUC Tuned SVM:

```r
## ROC and AUC for Hyperparameter SVM
```r
library(e1071)
library(ROCR)
svm_best <- svm(stroke ~ ., data = train_os,
                epsilon = best_epsilon,
                cost = best_cost,
                kernel = best_kernel,
                probability = TRUE)  # Enable probability estimates


# Compute ROC curve and AUC
test_pred_prob <- predict(svm_best, newdata = test, probability = TRUE)
pred_obj <- prediction(attr(test_pred_prob, "probabilities")[,2], as.numeric(test$stroke)) # Ensure target is numeric
perf <- performance(pred_obj, "tpr", "fpr")
auc <- performance(pred_obj, "auc")

# Plot ROC curve
plot(perf, main = "ROC Curve - SVM Model (Tuned)")
abline(a = 0, b = 1, lty = 2)  # Diagonal line representing random classifier

# Print AUC
print(paste("AUC:", as.numeric(auc@y.values)))
```
```

Naïve Bayes

Base NB Model:

```r
```r
library(naivebayes)

# Convert target variable to factor
train$stroke <- as.factor(train$stroke)
test$stroke <- as.factor(test$stroke)

# Train the Naive Bayes model on the training set
Naive_Bayes_model <- naive_bayes(x = train[, -10], # Excluding the target column
                                 y = train$stroke,
                                 laplace = 1)
print(Naive_Bayes_model)

# Predict on the training set
train_pred <- predict(Naive_Bayes_model, newdata = train[, -10], type = "class")

# Create a confusion matrix for the training set
train_confusion_matrix <- table(Actual = train$stroke, Predicted = train_pred)

# Display the confusion matrix for the training set
print("Confusion Matrix (Training Set):")
print(train_confusion_matrix)
```
```

Performance Evaluation of Base NB Model:

```r
# Calculate accuracy for the training set
train_accuracy <- sum(diag(train_confusion_matrix)) / sum(train_confusion_matrix)
print(paste("Accuracy (Training Set):", train_accuracy))

# Calculate sensitivity for the training set
TP_train <- train_confusion_matrix[2, 2]
FN_train <- train_confusion_matrix[2, 1]
sensitivity_train <- TP_train / (TP_train + FN_train)
print(paste("Sensitivity (Training Set):", sensitivity_train))

# Calculate F1 score for the training set
precision_train <- TP_train / sum(train_pred == "1")
F1_train <- (2 * precision_train * sensitivity_train) / (precision_train + sensitivity_train)
print(paste("F1 Score (Training Set):", F1_train))

# Calculate error rate for the training set
error_rate_train <- 1 - train_accuracy
print(paste("Error Rate (Training Set):", error_rate_train))
```

```r
# Predict on the test set
test_pred <- predict(Naive_Bayes_model, newdata = test[, -10], type = "class")

# Create a confusion matrix for the test set
test_confusion_matrix <- table(Actual = test$stroke, Predicted = test_pred)

# Display the confusion matrix for the test set
print("Confusion Matrix (Test Set):")
print(test_confusion_matrix)

# Calculate accuracy for the test set
test_accuracy <- sum(diag(test_confusion_matrix)) / sum(test_confusion_matrix)
print(paste("Accuracy (Test Set):", test_accuracy))

# Calculate sensitivity for the test set
TP_test <- test_confusion_matrix[2, 2]
FN_test <- test_confusion_matrix[2, 1]
sensitivity_test <- TP_test / (TP_test + FN_test)
specificity <-
print(paste("Sensitivity (Test Set):", sensitivity_test))

# Calculate F1 score for the test set
precision_test <- TP_test / sum(test_pred == "1")
F1_test <- (2 * precision_test * sensitivity_test) / (precision_test + sensitivity_test)
print(paste("F1 Score (Test Set):", F1_test))

# Calculate error rate for the test set
error_rate_test <- 1 - test_accuracy
print(paste("Error Rate (Test Set):", error_rate_test))
```

ROC and AUC for NB Model:

```
## ROC and AUC NB
```

```r
library(ROCR)

# Convert target variable to factor
train$stroke <- as.factor(train$stroke)
test$stroke <- as.factor(test$stroke)

# Train the Naive Bayes model on the training set
Naive_Bayes_model <- naive_bayes(x = train[, -10], # Excluding the target column
                                 y = train$stroke,
                                 laplace = 1)

# Predict probabilities on the test set
test_pred_prob <- predict(Naive_Bayes_model, newdata = test[, -10], type = "prob")

# Create prediction object
pred_obj <- prediction(test_pred_prob[, "1"], test$stroke)

# Calculate performance measures
perf <- performance(pred_obj, "tpr", "fpr")
auc <- performance(pred_obj, "auc")

# Plot ROC curve
plot(perf, main = "ROC Curve - Naive Bayes Model (Laplace = 1)")
abline(a = 0, b = 1, lty = 2)  # Diagonal line representing random classifier

# Print AUC
print(paste("AUC (Laplace = 1):", as.numeric(auc@y.values)))
```

Oversampled NB Model:

```r
# Convert target variable to factor
train_os$stroke <- as.factor(train_os$stroke)
test_os$stroke <- as.factor(test_os$stroke)

# Train the Naive Bayes model on the training set
Naive_Bayes_model <- naive_bayes(x = train_os[, -10], # Excluding the target column
                                 y = train_os$stroke,
                                 laplace = 1)

# Predict on the training set
train_pred <- predict(Naive_Bayes_model, newdata = train_os[, -10], type = "class")

# Create a confusion matrix for the training set
train_confusion_matrix <- table(Actual = train_os$stroke, Predicted = train_pred)

# Display the confusion matrix for the training set
print("Confusion Matrix (Training Set):")
print(train_confusion_matrix)
```

Performance Evaluation of Oversampled NB Model:

```r
## F1 score
# Calculate accuracy for the training set
train_accuracy <- sum(diag(train_confusion_matrix)) / sum(train_confusion_matrix)
print(paste("Accuracy (Training Set):", train_accuracy))

# Calculate sensitivity for the training set
TP_train <- train_confusion_matrix[2, 2]
FN_train <- train_confusion_matrix[2, 1]
sensitivity_train <- TP_train / (TP_train + FN_train)
print(paste("Sensitivity (Training Set):", sensitivity_train))

# Calculate F1 score for the training set
precision_train <- TP_train / sum(train_pred == "1")
F1_train <- (2 * precision_train * sensitivity_train) / (precision_train + sensitivity_train)
print(paste("F1 Score (Training Set):", F1_train))

# Calculate error rate for the training set
error_rate_train <- 1 - train_accuracy
print(paste("Error Rate (Training Set):", error_rate_train))
```

ROC and AUC for Oversampled NB Model:

```r
## ROC and AUC for Oversampled NB
library(ROCR)

# Convert target variable to factor
train_os$stroke <- as.factor(train_os$stroke)
test_os$stroke <- as.factor(test_os$stroke)

# Train the Naive Bayes model on the training set
Naive_Bayes_model <- naive_bayes(x = train_os[, -10], # Excluding the target column
                                 y = train_os$stroke,
                                 laplace = 1)

# Predict probabilities on the test set
test_pred_prob <- predict(Naive_Bayes_model, newdata = test_os[, -10], type = "prob")

# Create prediction object
pred_obj <- prediction(test_pred_prob[, "1"], test_os$stroke)

# Calculate performance measures
perf <- performance(pred_obj, "tpr", "fpr")
auc <- performance(pred_obj, "auc")

# Plot ROC curve
plot(perf, main = "ROC Curve - Naive Bayes Model Oversampled (Laplace = 1)")
abline(a = 0, b = 1, lty = 2)  # Diagonal line representing random classifier

# Print AUC
print(paste("AUC (Laplace = 1):", as.numeric(auc@y.values)))
```