Stroke Prediction Group -2
Individual Report

Introduction:

The study aims to predict the likelihood of a stroke recurrence using a publicly available stroke prediction dataset.

Data Description

The “healthcare-dataset-stroke-data” is a stroke prediction dataset from Kaggle that contains 5110 observations (rows) with 12 attributes (columns). Each observation corresponds to one patient, and the attributes are variables about the health status of each patient. In particular, the categorical variables are id, gender, hypertension (yes/no), heart disease (yes/no), marital status, work type (children, government job, never worked, private, self-employed), residence type (urban, rural), smoking status (formerly smoked, never smoked, smokes), and stroke history. As for quantitative variables we have one’s age, BMI, and average glucose level.

Data Cleansing

The data cleansing process helps to ensure that the dataset used for training the machine learning model is of high quality, which in turn increases the likelihood of developing a robust and accurate model. It's essential to carefully examine the data, identify any issues or inconsistencies, and address them appropriately before moving forward with the modeling process.
The data includes NA values and unknown values, and the following steps were taken for data cleansing:

1. Handling missing values (N/A): The dataset contains missing values (N/A) in the BMI column, with 201 missing values out of 5109 rows. Missing values can lead to incorrect or biased results when training a machine learning model. In this case, the missing values are filled with the mean of the BMI column. Using the mean is a common method for imputing missing values in numerical columns, as it helps to preserve the overall distribution of the data while minimizing the impact of missing values on the model. However, it's important to note that this approach may not always be appropriate, as it could introduce bias or skew the data distribution in certain cases.

2. Removing irrelevant columns: The 'id' column is removed from the dataset because it doesn't provide any meaningful information for the classification task. Including such columns in the analysis can lead to an unnecessarily complex model and can negatively impact the model's performance. By removing irrelevant columns, we can focus on the features that are important for predicting the target variable, resulting in a more accurate and efficient model.

Label Encoding Categorical Values

Label encoding is a process of converting categorical variables into numerical values so that machine learning algorithms can work with the data more effectively. Machine Learning algorithms, including the Random Forest classifier used in this project, require numerical inputs, and cannot directly handle categorical data. By converting the categorical variables into numerical values, we can ensure that the model can process the features and make predictions based on them.
In this case, label encoding is applied to the following categorical columns: 'gender', 'ever_married', 'work_type', 'Residence_type', and 'smoking_status'. The LabelEncoder from the scikit-learn library is used to perform the encoding. For each column, the LabelEncoder assigns a unique integer value to each distinct category in the column, effectively mapping the categorical values to numerical values. After applying label encoding, you can see that the unique values in each categorical column have been replaced with numerical values, making the data more suitable for input to the machine learning model.

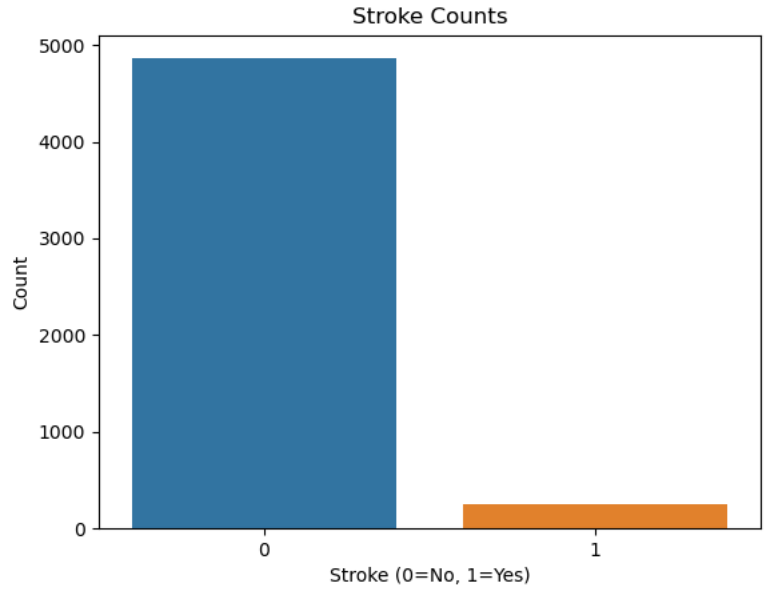
However, it is essential to consider the limitations of label encoding. One issue with label encoding is that it introduces an arbitrary ordering of categories, which might not reflect their true relationship. For example, assigning 0 to 'never smoked' and 1 to 'formerly smoked' in the 'smoking_status' column implies that 'formerly smoked' is greater than 'never smoked', which might not be accurate.

An alternative approach to handle categorical data is one-hot encoding, which creates binary (0 or 1) columns for each category in the original column, avoiding the ordinal relationship imposed by label encoding. However, one-hot encoding can significantly increase the number of columns in the dataset and potentially lead to a higher computational cost.

To overcome this limitation, my teammate used one-hot encoding, which creates binary (0 or 1) features for each category in a categorical variable. One of the most common ways to perform one-hot encoding is by using the pandas' library's get_dummies () function. This function creates a new data frame with binary columns for each category/label present in the categorical variable. Each row will have a 1 in the column corresponding to the category it belongs to and a 0 for the other categories.

EDA

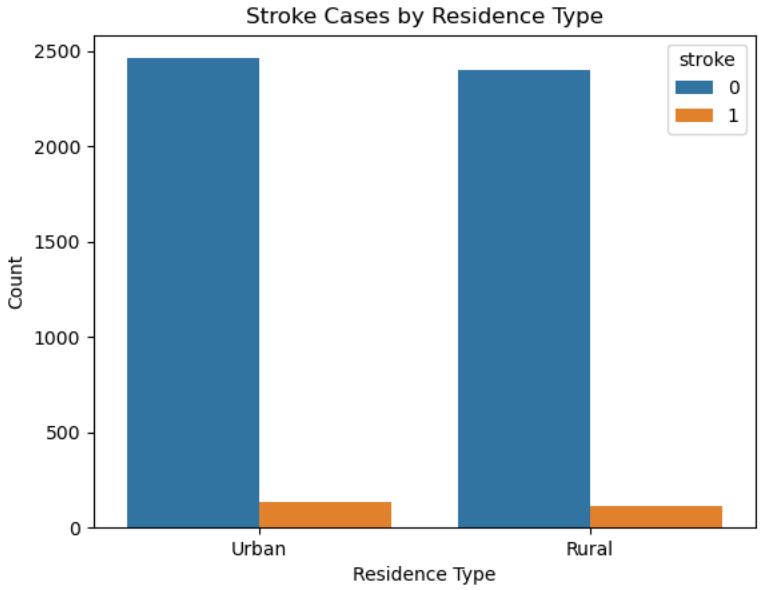
Target feature - **Stroke**: First, the value counts of the stroke variable are printed, showing the number of instances with and without a stroke. A bar chart is plotted to visualize the distribution of stroke cases (0 = No Stroke, 1 = Stroke). This helps in identifying any class imbalance in the dataset.




```
Value count in the stroke :
 0    4861
1     249
Name: stroke, dtype: int64
```

Residence Type: The value counts of the Residence_type variable are printed, and a bar chart is plotted to show the distribution of stroke cases by residence type (Urban or Rural).

```
Value of count of residence-
 Urban    2596
Rural    2514
Name: Residence_type, dtype: int64
```



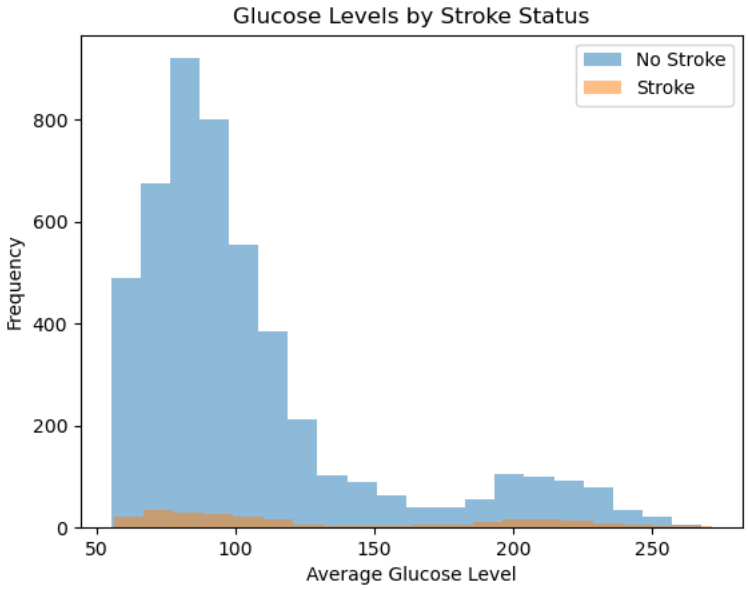
Observation:
This attribute is of no use. As we can see there is not much difference in both attribute values. Maybe we must discard it.

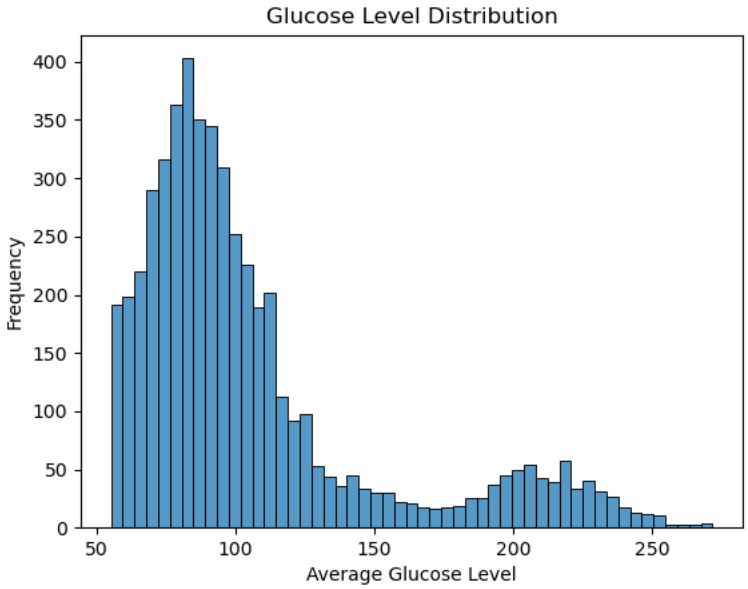
Average Glucose Level

Value and Count

```
Name: avg_glucose_level, Length: 3979, dtype: int64
```

Plot -Glucose level with stroke

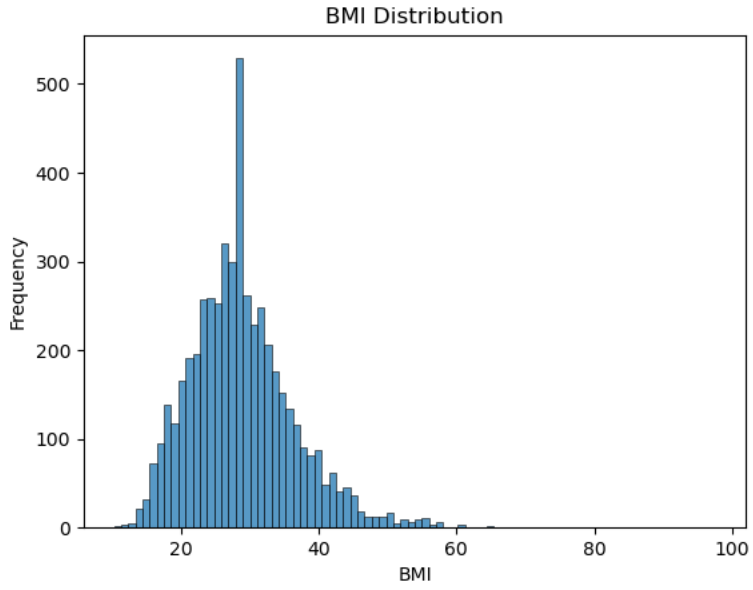




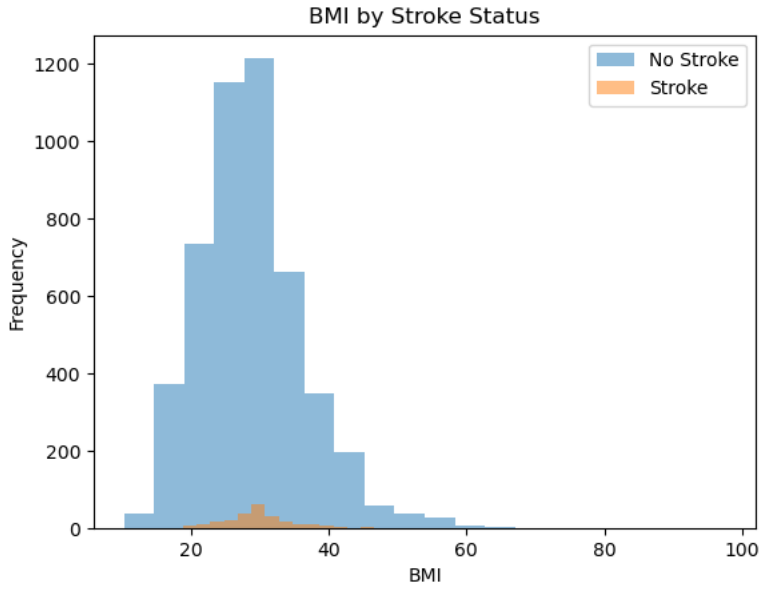
Observation: From the above graph, we can see that people having a stroke have an average glucose level of more than 100.

BMI

It has a null value of 201 and the count is 419.



BMI With stroke

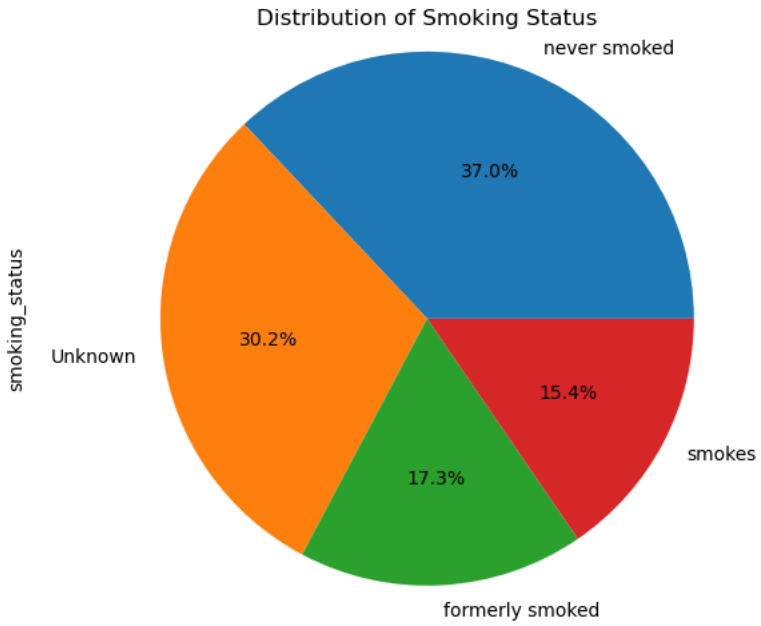


Smoking Status

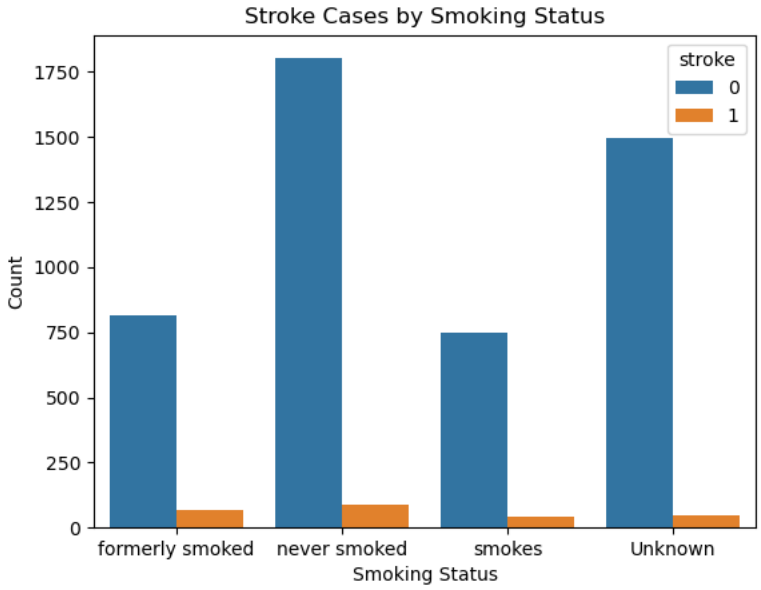
Value count

```
Value of count of smoking status-
 never smoked       1892
Unknown             1544
formerly smoked      885
smokes               789
Name: smoking_status, dtype: int64
```

Count plot:

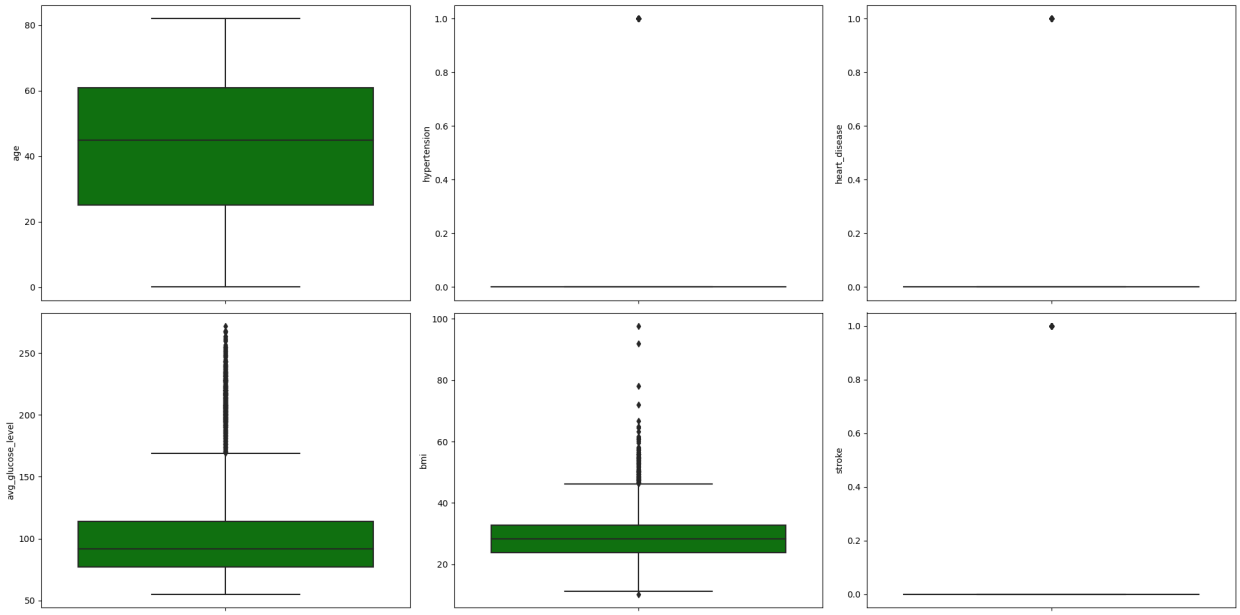


Smoke and Stroke:

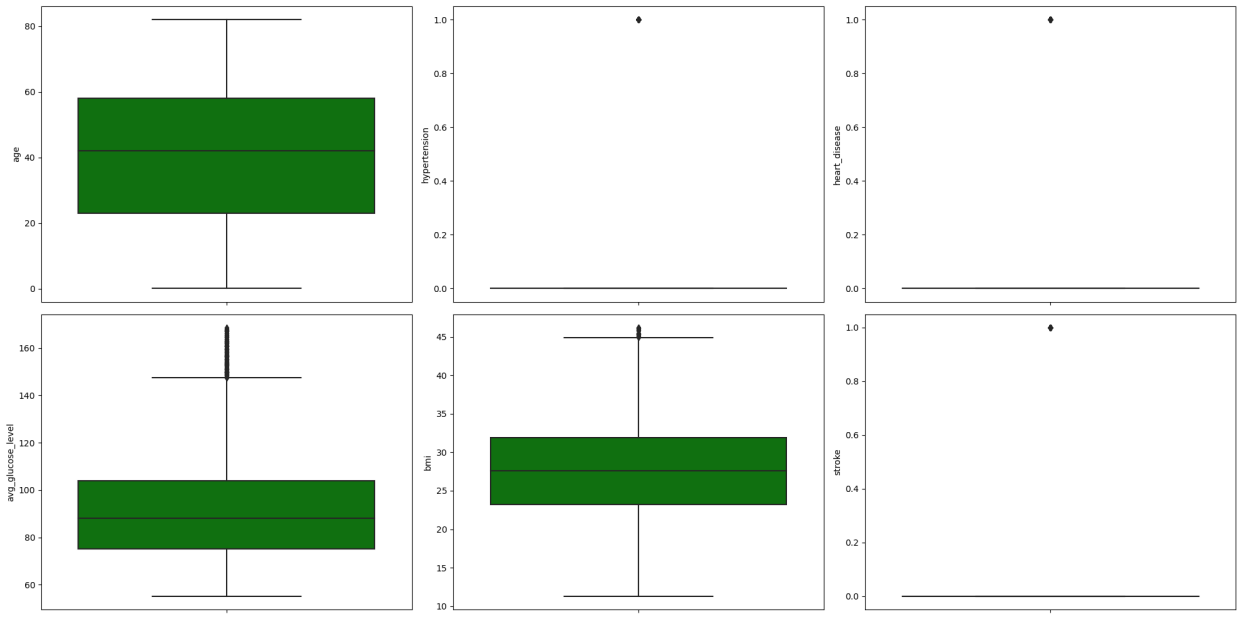


Observation: As per these plots, we can see there is not much difference in the chances of stroke irrespective of smoking status.

Outliers – In this, we identify and treat outliers in the dataset, specifically for the 'avg_glucose_level' and 'bmi' variables. Outliers can have a significant impact on the performance of certain machine learning models, as they might skew the data distribution and affect the model's ability to make accurate predictions. By treating the outliers, the data becomes more representative of the general population and less influenced by extreme values. This can help improve the performance of machine learning models, as they can better capture the underlying patterns in the data.



After Outliers



Data Sampling for Imbalanced Classification

The output column "stroke" has a value of either 1 or 0, where 0 indicates no stroke risk and 1 indicates a stroke risk. In the dataset, the occurrence of 0 in the output column (stroke) is greater than the occurrence of 1. There are 129 rows with a value of 1 in the stroke column and 3383 rows with a value of 0. To enhance the accuracy, data preprocessing was applied to balance the data. There are 3389 patients with a stroke value equal to 0 and 129 patients with a stroke value equal to 1. The ratio of observations in ‘stroke’ is about 1:20, which is highly imbalanced. To resolve this problem, we under-sample the majority class, the class of stroke value being 1, to

achieve a balanced class distribution. To resolve this problem, the SMOTE (Synthetic Minority Over-sampling Technique) was used to balance the data. The balance of the output column in the dataset after applying the SMOTE technique was as follows:

Class counts before SMOTE oversampling:
Class 0: 3389 Class 1: 129
Class counts after SMOTE oversampling:
Class 0: 3389 Class 1: 3389

Preprocessing:

Before building a predictive model, the dataset was preprocessed to ensure its accuracy and efficiency. The preprocessing stage involved cleaning and preparing the data for model development. Missing values were filled, and string literals were converted to integer values through label encoding.
Before building a predictive model, it is crucial to preprocess the dataset to ensure the model's accuracy and efficiency. The preprocessing stage involves cleaning and preparing the data for model development. In this case, the dataset used has twelve characteristics, and the column 'id' is omitted as it does not contribute to the model's construction. Next, the dataset is checked for missing values and any detected missing values are filled. For example, in this case, the missing values in the column 'BMI' are filled with the mean of the column. After handling missing values, the next step is to convert the string literals in the dataset to integer values that can be understood by the computer. This process is called label encoding, and in this case, the dataset has five columns with string data type. All string values are encoded, and the entire dataset is transformed into numerical values. The dataset used for stroke prediction is imbalanced, with 5110 rows in total, 249 rows indicating the possibility of a stroke, and 4861 rows confirming the absence of a stroke. Training a machine-learning model using such an imbalanced dataset can result in high accuracy, but other accuracy measures such as precision and recall may be lacking. To ensure the model's effectiveness, it is necessary to address the imbalance in the data. The SMOTE (Synthetic Minority Over-sampling Technique) is used for this purpose.

Model Algorithms:

The study evaluated the performance of several commonly used machine learning methods including Random Forest, XGBoost, and Gradient Boosting. These algorithms were selected based on their popularity and effectiveness in solving similar problems in the medical field. The accuracy, precision, and recall of the model are then evaluated using the testing data to determine the most effective algorithm.

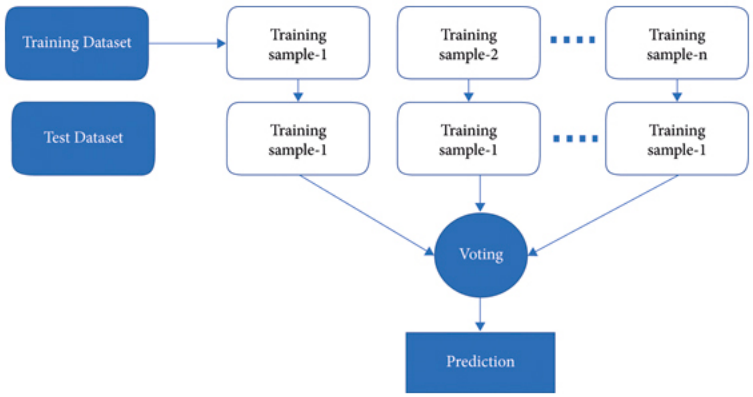
1) Random Forest

2) Gradient Boosting

3) XGBoost

Random Forest

The chosen classification algorithm for this project is the Random Forest (RF) classifier. Random Forests are ensemble learning methods that consist of numerous independent decision trees, each trained on a random subset of the data. The core idea behind this approach is to create a "forest" of decision trees that work together to produce more accurate and stable predictions than any single decision tree could achieve on its own. During the training phase, each decision tree is constructed by selecting a random sample of the data and applying a learning algorithm to generate a tree structure. This random sampling introduces diversity in the decision trees, which helps reduce overfitting and improve the generalization ability of the model. Once the decision trees are trained, the RF classifier combines their outputs through a process called "voting" to arrive at the final prediction. In this specific case, each decision tree must vote for one of the two output classes (stroke or no stroke). The class that receives many votes from the decision trees is then chosen as the final prediction.
The strength of the RF classifier lies in its ability to leverage the collective knowledge of multiple decision trees, effectively reducing the impact of individual tree weaknesses, such as biases or overfitting. This results in a more robust and accurate model that can better handle the complexity and nuances of the data, ultimately leading to improved stroke prediction performance. A block diagram of random forest classification is shown in Figure.



The versatility of the Random Forest is one of its most appealing characteristics. It can be employed for both regression and classification tasks, and the relative importance of input features is easily discernible. Additionally, this method is advantageous because the default hyperparameters often yield clear predictions.
Comprehending the hyperparameters is crucial since there are only a few to consider in the first place. Overfitting is a common issue in machine learning; however, it is rarely a problem with the Random Forest classifier. If there is an adequate number of trees in the forest, the classifier is unlikely to overfit the model.

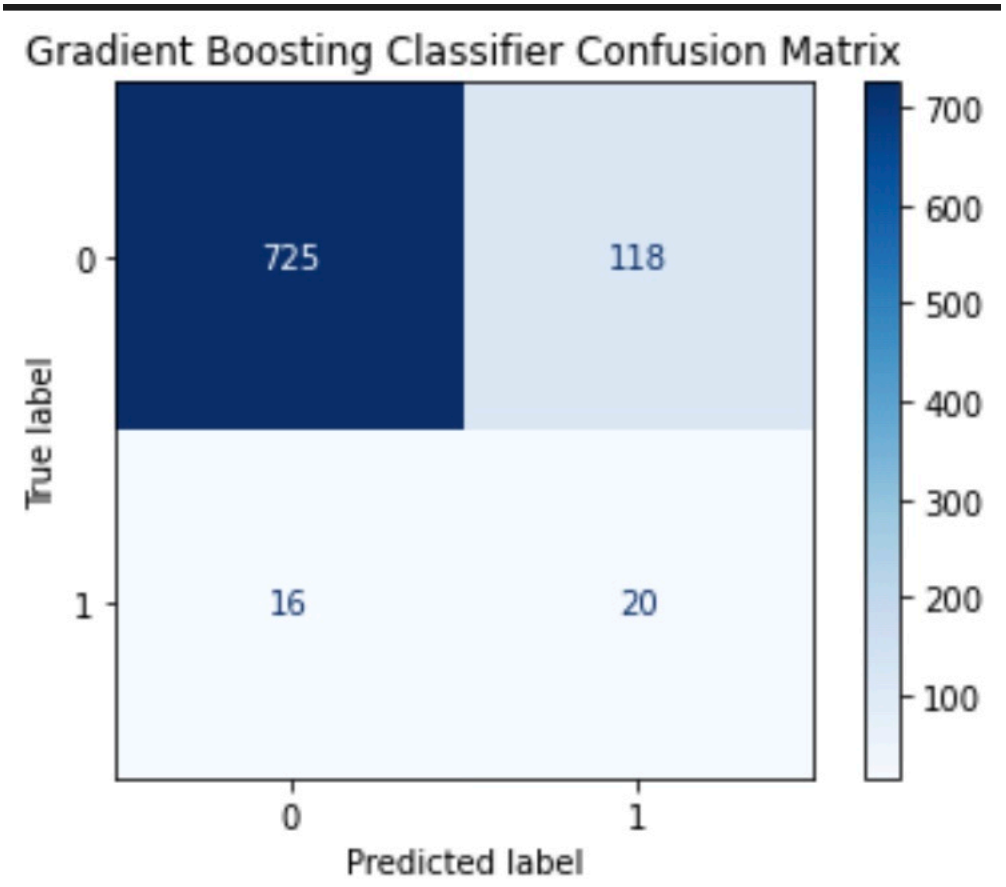
By constructing a diverse ensemble of decision trees and aggregating their predictions, the Random Forest classifier can achieve high performance and stability across a wide range of tasks. This adaptability, combined with its inherent resistance to overfitting and ease of interpretation, makes the Random Forest classifier a powerful and popular choice in the field of machine learning.

Gradient Boosting Classifier

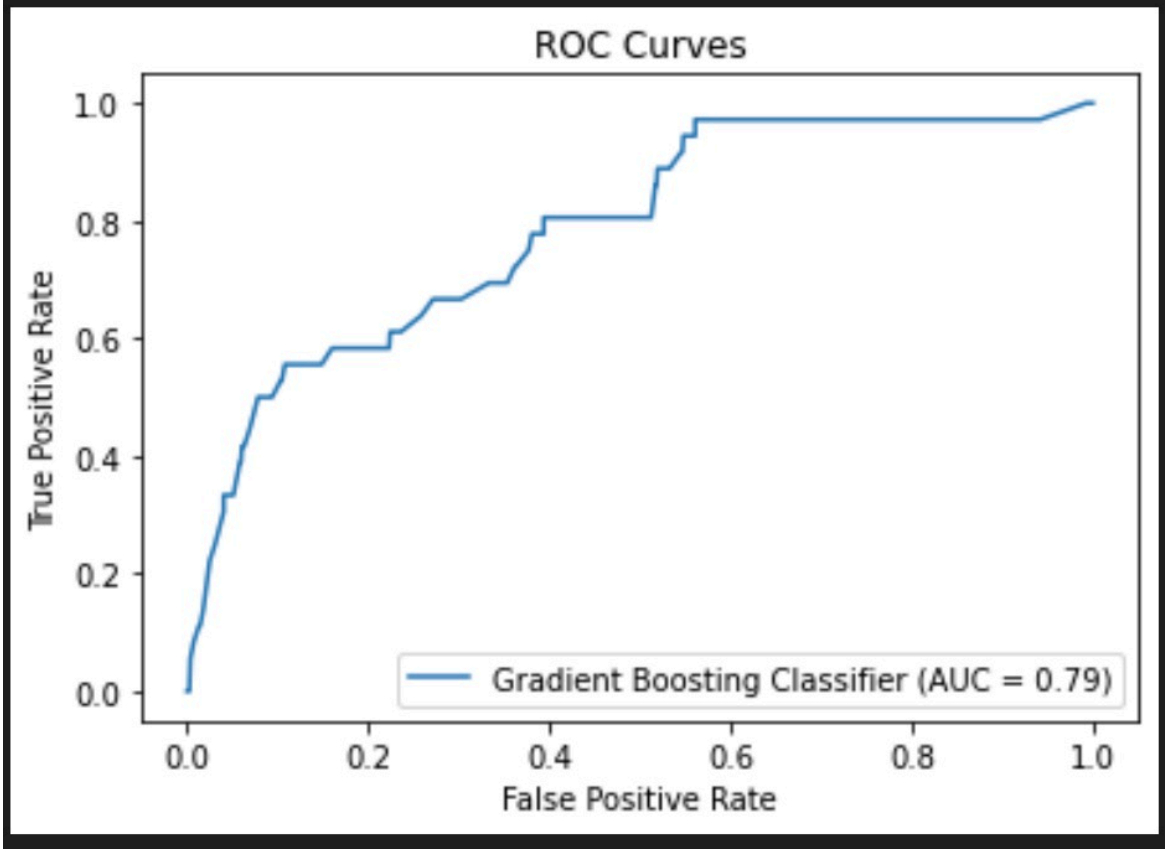
Gradient boosting is one of the variants of ensemble methods where you create multiple weak models and combine them to get better performance.
The gradient boosting algorithm is one of the most powerful algorithms in the field of machine learning. As we know that the errors in machine learning algorithms are broadly classified into two categories i.e., Bias Error and Variance Error. As gradient boosting is one of the boosting algorithms it is used to minimize the bias error of the model.
Gradient boosting algorithm can be used for predicting not only continuous target variable (as a Regressor) but also categorical target variable (as a Classifier). When it is used as a regressor, the cost function is Mean Square Error (MSE) and when it is used as a classifier then the cost function is Log loss.

Gradient boosting is a highly robust technique for developing predictive models. It applies to several risk functions and optimizes the accuracy of the model's prediction. It also resolves multicollinearity problems where the correlations among the predictor variables are high.

Gradient Boosting Classifier

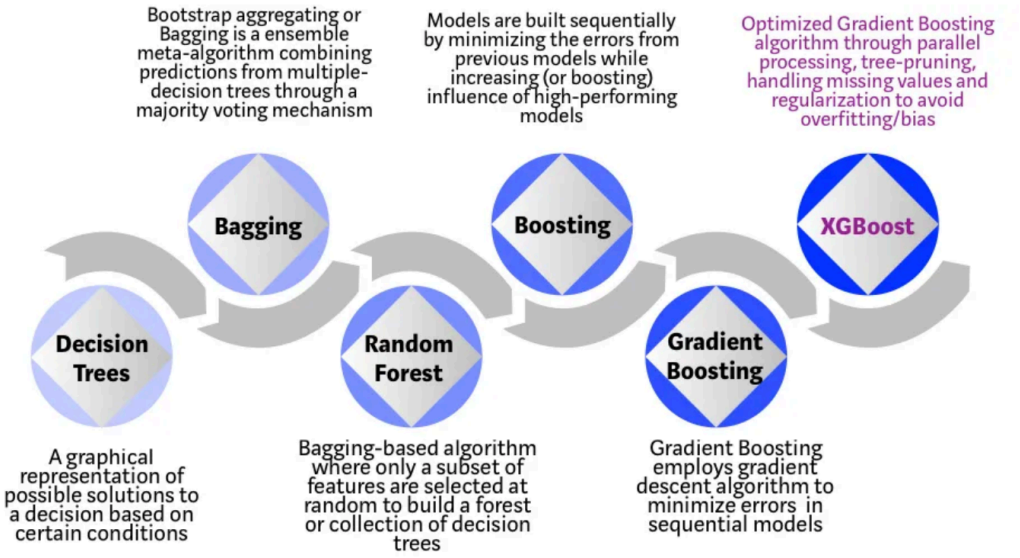

Gradient Boosting Classifier Confusion Matrix
725
118
16
20
0
1
True label
0
1
Predicted label
700
600
500
400
300
200
100

```
Accuracy-Gradient Boosting Classifier
: 0.8475540386803185
Gradient Boosting Classifier confusion matrix-
 [[725 118]
 [ 16  20]]
Gradient Boosting Classifier Classification report
              precision    recall  f1-score   support

           0       0.98      0.86      0.92       843
           1       0.14      0.56      0.23        36

    accuracy                           0.85       879
   macro avg       0.56      0.71      0.57       879
weighted avg       0.94      0.85      0.89       879
```



XGBOOST

XGBoost algorithm is an extended version of the gradient boosting algorithm. It is basically designed to enhance the performance and speed of a Machine Learning model.
XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient-boosting framework. In prediction problems involving unstructured data (images, text, etc.) artificial neural networks tend to outperform all other algorithms or frameworks. However, when it comes to small-to-medium structured/tabular data, decision tree-based algorithms are considered best-in-class right now.



Each step of the evolution of tree-based algorithms can be viewed as a version of the interview process.

Decision Tree: Every hiring manager has a set of criteria such as education level, number of years of experience, and interview performance. A decision tree is analogous to a hiring manager interviewing candidates based on his or her own criteria.

Bagging: Now imagine instead of a single interviewer, now there is an interview panel where each interviewer has a vote. Bagging or bootstrap aggregating involves combining inputs from all interviewers for the final decision through a democratic voting process.

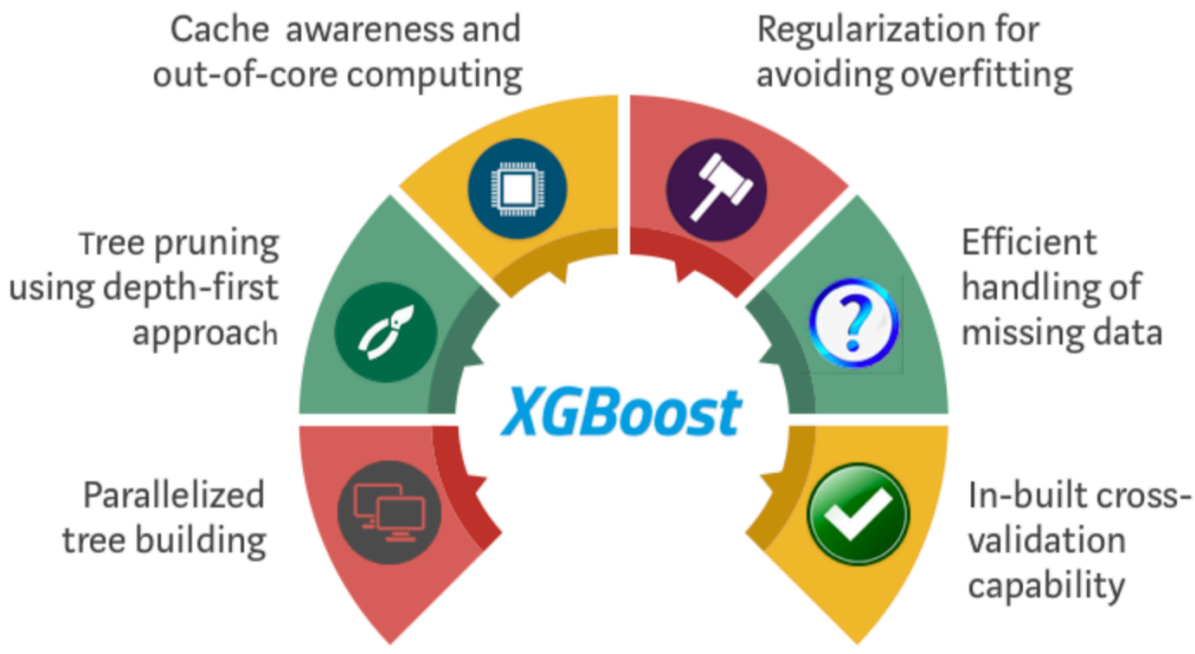
Random Forest: It is a bagging-based algorithm with a key difference wherein only a subset of features is selected at random. In other words, every interviewer will only test the interviewee on certain randomly selected qualifications (e.g., a technical interview for testing programming skills and a behavioral interview for evaluating non-technical skills).

Boosting: This is an alternative approach where each interviewer alters the evaluation criteria based on feedback from the previous interviewer. This 'boosts' the efficiency of the interview process by deploying a more dynamic evaluation process.

Gradient Boosting: A special case of boosting where errors are minimized by a gradient descent algorithm e.g., the strategy consulting firms leverage by using case interviews to weed out less qualified candidates.

XGBoost: Think of XGBoost as gradient boosting on 'steroids' (well it is called 'Extreme Gradient Boosting' for a reason!). It is a perfect combination of software and hardware optimization techniques to yield superior results using fewer computing resources in the shortest amount of time.

XGBoost and Gradient Boosting Machines (GBMs) are both ensemble tree methods that apply the principle of boosting weak learners (CARTs generally) using the gradient descent architecture. However, XGBoost improves upon the base GBM framework through systems optimization and algorithmic enhancements.



Parallelization: XGBoost employs a parallelized implementation for sequential tree building. This is achieved by interchanging the order of the nested loops used to construct base learners: the outer loop enumerating leaf nodes of a tree, and the inner loop calculating features. The original nesting order hinders parallelization, as the outer loop cannot begin until the computationally demanding

inner loop is completed. By initializing through a global scan of all instances and sorting with parallel threads, the loop order switch enhances algorithmic performance by counterbalancing parallelization overheads.

Tree Pruning: In contrast to the GBM framework's greedy stopping criterion, which depends on the negative loss criterion at the point of the split, XGBoost uses the 'max_depth' parameter first and begins pruning trees backward. This 'depth-first' approach significantly improves computational performance.

Hardware Optimization: XGBoost is designed to efficiently utilize hardware resources. It achieves cache awareness by allocating internal buffers in each thread to store gradient statistics. Additional enhancements, such as 'out-of-core' computing, optimize disk space usage when handling large data frames that do not fit into memory.

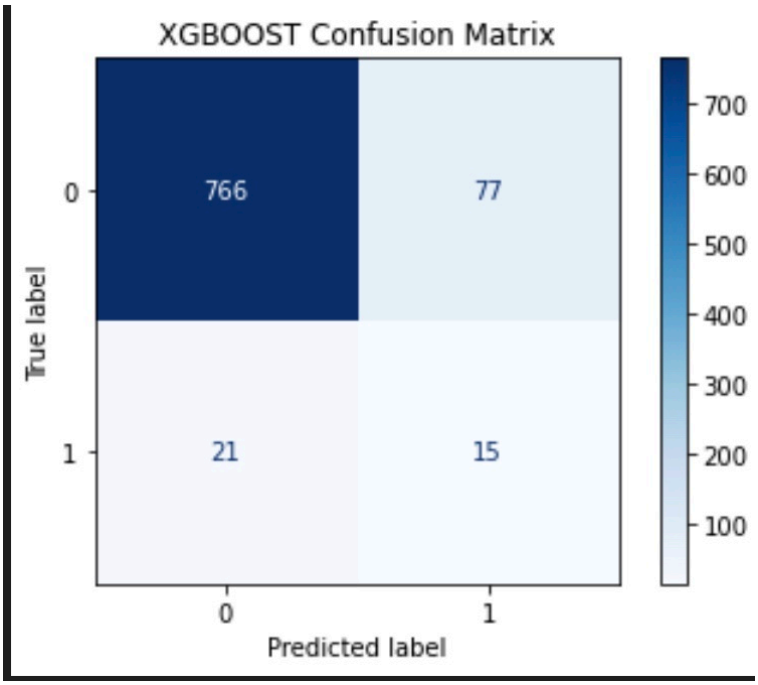
Algorithmic Enhancements:

Regularization: It penalizes more complex models through both LASSO (L1) and Ridge (L2) regularization to prevent overfitting.

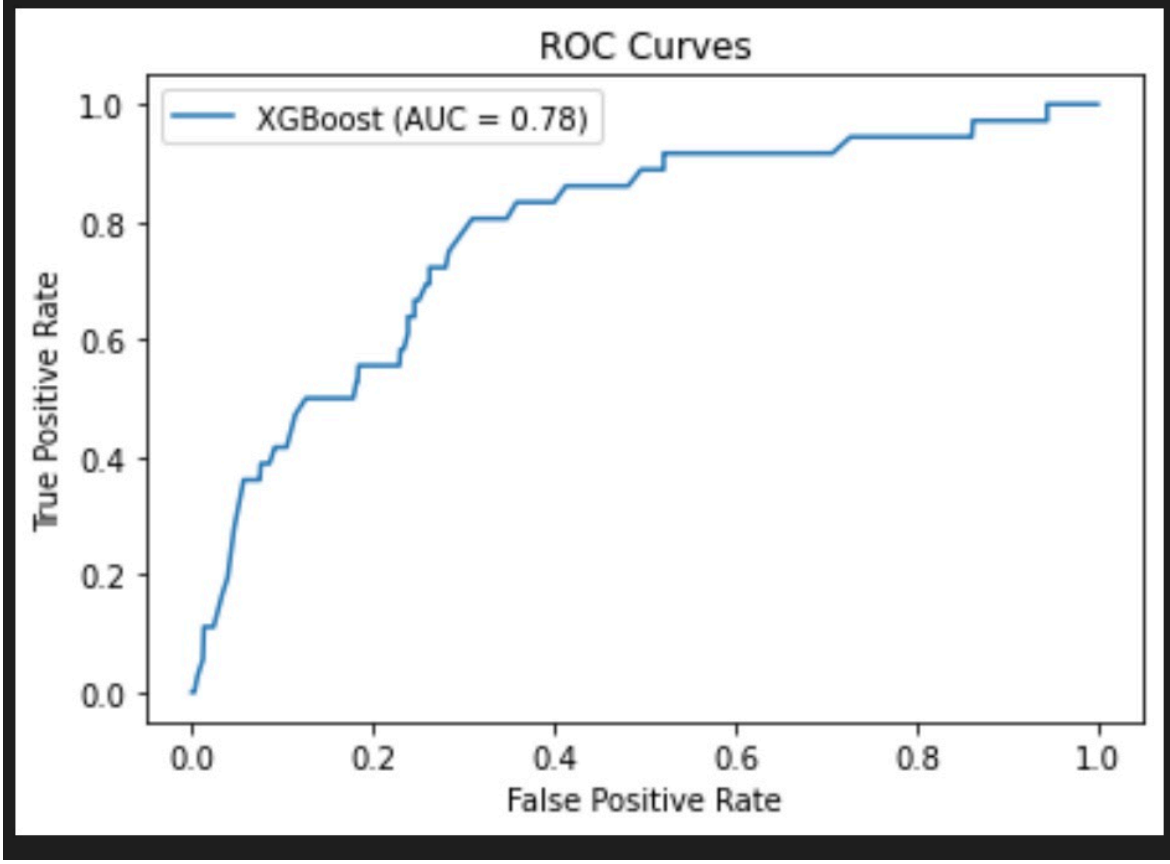
Sparsity Awareness: XGBoost naturally admits sparse features for inputs by automatically ‘learning’ the best missing value depending on training loss and handles different types of sparsity patterns in the data more efficiently.

Weighted Quantile Sketch: XGBoost employs the distributed weighted Quantile Sketch algorithm to effectively find the optimal split points among weighted datasets.

Cross-validation: The algorithm comes with a built-in cross-validation method at each iteration, taking away the need to explicitly program this search and to specify the exact number of boosting iterations required in a single


XGBOOST Confusion Matrix
766
77
21
15
0
1
True label
0
1
Predicted label
700
600
500
400
300
200
100

```
XGBOOST confusion matrix-
 [[766  77]
 [ 21  15]]
XGBOOST Classification report
              precision    recall  f1-score   support

           0       0.97      0.91      0.94       843
           1       0.16      0.42      0.23        36

    accuracy                           0.89       879
   macro avg       0.57      0.66      0.59       879
weighted avg       0.94      0.89      0.91       879

Accuracy-XGBOOST
: 0.888509670079636
```



Cross Validation Scores:

```
Random Forest Accuracy: 0.9057097969636316
Gradient Boosting Classifier Accuracy: 0.8488043316755963
XGBoost Accuracy: 0.9051180809873003
```

Final Output

Model Name	Precision	F1 Score	ROC	Accuracy	Recall
Random Forest	0.17	0.23	0.76	0.89	0.39
XGBoost	0.16	0.23	0.78	0.88	0.42
Gradient	0.14	0.23	0.79	0.84	0.56