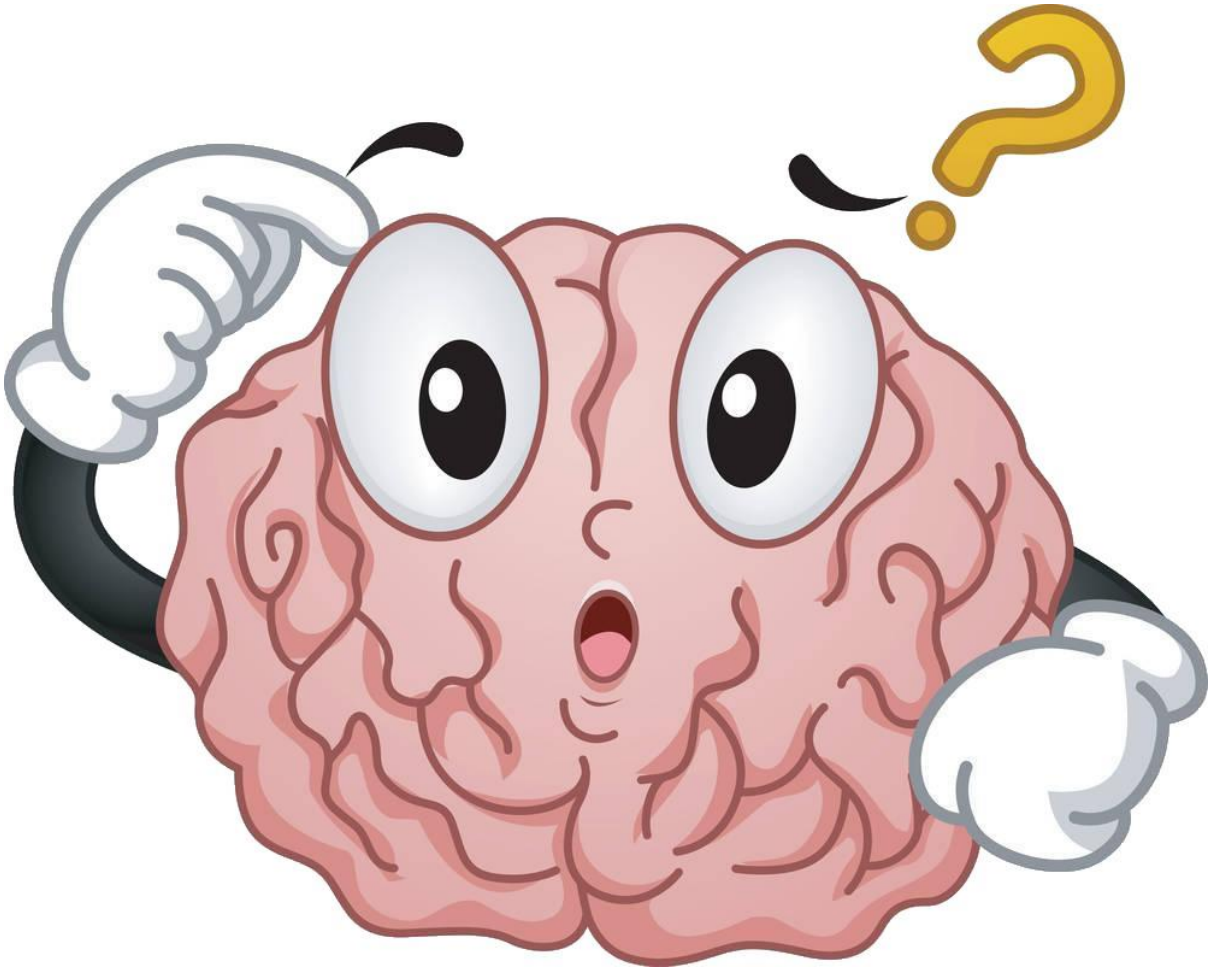
BRAIN STROKE PREDICTION



SUBJECT: CS634-DATA MINING

GUIDED BY: PROF. CHEN HSEU

SUBMITTED BY: ROHAN KHANNA (RK868) & PARTH PANARA (PMP24)

INTRODUCTION:

Stroke, a debilitating and potentially life-threatening neurological event, is a major global health concern. The timely identification and prediction of stroke risk factors are critical for preventive healthcare and effective patient management. Data mining techniques offer a powerful toolset to analyse large datasets and uncover hidden patterns that can aid in stroke prediction. This project aims to leverage the potential of data mining in the context of brain stroke prediction, with the ultimate goal of improving early diagnosis and patient outcomes.

Stroke occurs when the blood supply to a part of the brain is interrupted or reduced, leading to brain cell damage. Early diagnosis and intervention are pivotal to minimizing the long-term effects of stroke, making it imperative to develop accurate prediction models. By applying data mining methods to a diverse range of patient data, including medical history, lifestyle factors, and clinical measurements, this project seeks to identify key risk factors and build predictive models that can assist healthcare professionals in assessing stroke risk for individual patients. According to the World Health Organization (WHO), stroke is the 2nd leading cause of death globally, responsible for approximately 11% of total deaths.

ABSTRACT:

This data mining project addresses the critical issue of brain stroke prediction, a significant public health concern worldwide. Stroke is a devastating medical event that often leads to severe disability or even death. Early detection and risk assessment are essential to reduce the impact of strokes on individuals and healthcare systems. In this project, we apply advanced data mining techniques to analyse a comprehensive dataset comprising medical records, lifestyle data, and clinical measurements.

Our objective is to develop accurate predictive models that can identify individuals at risk of experiencing a stroke. By exploring and processing this extensive dataset, we aim to uncover hidden patterns and relationships among various risk factors, including age, gender, hypertension, diabetes, smoking habits, and glucose levels. Through the data mining algorithms, we intend to create a predictive tool that healthcare professionals can utilize to assess stroke risk more effectively.

This project contributes to the advancement of medical science by harnessing the potential of data mining to enhance stroke prediction and prevention. Ultimately, our research may empower healthcare providers to take proactive measures in managing stroke risk factors, leading to improved patient outcomes and a reduced burden on healthcare systems.

DATASET DESCRIPTION:

This dataset is used to predict whether a patient is likely to get a stroke based on input parameters like gender, age, various diseases, and smoking status. Each row in the data provides relevant information about the patient. The dataset consists of 5110 entries.

Attribute Information from the Kaggle Website:
1) id: unique identifier
2) gender: "Male", "Female" or "Other"
3) age: age of the patient
4) hypertension: 0 if the patient doesn't have hypertension, 1 if the patient has hypertension

5) heart_disease: 0 if the patient doesn't have any heart diseases, 1 if the patient has a heart disease
6) ever_married: "No" or "Yes"
7) work_type: "children", "Govt_jov", "Never_worked", "Private" or "Self-employed"
8) Residence_type: "Rural" or "Urban"
9) avg_glucose_level: average glucose level in blood
10) BMI: body mass index
11) smoking_status: "formerly smoked", "never smoked", "smokes" or "Unknown"*
12) stroke: 1 if the patient had a stroke or 0 if not

*Note: "Unknown" in smoking_status means that the information is unavailable for this patientNumerical data comprises attributes such as age, average glucose level, and body mass index (BMI).

- Categorical data encompasses characteristics like gender, hypertension, heart disease, marital status, occupation type, residential area, and smoking status.

- NOTE: All categorical data are nominal. "Unknown" in smoking_status means that the information is unavailable for this patient

DATASET LINK:

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

MACHINE LEARNING ENVIRONMENT SETUP AND LIBRARY IMPORTS:

1. **Data Manipulation and Visualization Libraries (Source: General Data Analysis and Visualization):**
    - **import pandas as pd**: Imports the pandas library and assigns it the alias 'pd' for handling data structures like DataFrames.
    - **import numpy as np**: Imports the NumPy library and assigns it the alias 'np' for numerical operations and array handling.
    - **import matplotlib.pyplot as plt**: Imports the Matplotlib library for creating static, interactive, and animated visualizations.
    - **import seaborn as sns**: Imports the Seaborn library for statistical data visualization based on Matplotlib.
2. **Data Preprocessing and Evaluation Metrics (Source: Scikit-learn and Imbalanced-Learn):**
    - **from sklearn.preprocessing import StandardScaler**: Imports the StandardScaler class from scikit-learn for standardizing feature variables.
    - **from sklearn.model_selection import train_test_split**: Imports the train_test_split function for splitting data into training and testing sets.
    - **from imblearn.over_sampling import SMOTE**: Imports the Synthetic Minority Over-sampling Technique (SMOTE) for handling class imbalance in classification tasks.
    - **from sklearn.metrics import classification_report, accuracy_score, confusion_matrix**: Imports metrics for evaluating classification models, including accuracy, confusion matrix, and classification report.
    - **from sklearn.metrics import roc_auc_score, roc_curve, precision_score, recall_score, f1_score**: Imports additional metrics related to receiver operating characteristic (ROC) curves and precision-recall curves.
    - **import warnings**: Imports the warnings module to handle warning messages during code execution.
    - **from scipy.stats import boxcox**: Imports the boxcox function from SciPy for performing Box-Cox transformations on data.
3. **Machine Learning Models (Source: Scikit-learn, LightGBM, XGBoost):**
    - **from sklearn.tree import DecisionTreeClassifier**: Imports the DecisionTreeClassifier for building decision tree-based classification models.
    - **from sklearn.svm import SVC**: Imports the Support Vector Classification (SVC) algorithm for building support vector machine models.
    - **from sklearn.naive_bayes import GaussianNB**: Imports the Gaussian Naive Bayes algorithm for probabilistic classification.

- **from sklearn.linear_model import LogisticRegression**: Imports the Logistic Regression algorithm for binary and multiclass classification.
- **from sklearn.ensemble import RandomForestClassifier**: Imports the RandomForestClassifier for building ensemble models based on decision trees.
- **from sklearn.neighbors import KNeighborsClassifier**: Imports the KNeighborsClassifier for building k-nearest neighbors classification models.
- **from lightgbm import LGBMClassifier**: Imports the LGBMClassifier for gradient boosting-based classification.
- **from xgboost import XGBClassifier**: Imports the XGBClassifier for extreme gradient boosting-based classification.

4. **Model Interpretability (Source: Scikit-learn):**
    - **from sklearn.inspection import permutation_importance**: Imports the permutation_importance function for evaluating feature importance through permutation-based methods.
    - **from sklearn.metrics import plot_confusion_matrix**: Imports the plot_confusion_matrix function for visualizing the confusion matrix of a classifier.

SUMMARY OF THE STEPS TO BE TAKEN AND ANALYSING THE PROCESS FOR THE DATA MINING:

Loading the Dataset:

- The dataset, named **stroke_data**, is loaded from a CSV file using the pandas library.
- We have attached the screenshot of the first five rows of dataset with it's output below.

```
# Print the first 5 rows of the dataset
print("\nFirst 5 rows of the dataset:\n")
display(stroke_data.head())
```

```
First 5 rows of the dataset:
```

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	9046	Male	67.0	0	1	Yes	Private	Urban	228.69	36.6	formerly smoked	1
1	51676	Female	61.0	0	0	Yes	Self-employed	Rural	202.21	NaN	never smoked	1
2	31112	Male	80.0	0	1	Yes	Private	Rural	105.92	32.5	never smoked	1
3	60182	Female	49.0	0	0	Yes	Private	Urban	171.23	34.4	smokes	1
4	1665	Female	79.0	1	0	Yes	Self-employed	Rural	174.12	24.0	never smoked	1

Statistical Analysis of Numerical Attributes:

- The code utilizes the **describe()** method to generate statistical summaries for all numerical attributes in the dataset. This includes count, mean, standard deviation, minimum values, all quartiles, and maximum values.
- The output is displayed using the **display()** function in the code.

Dataset Shape and Attribute Information:

- The code prints the shape of the dataset (number of rows: 5110 and columns: 12).
- It then uses the **info()** method to display information about the data types and the number of non-null values for each attribute.

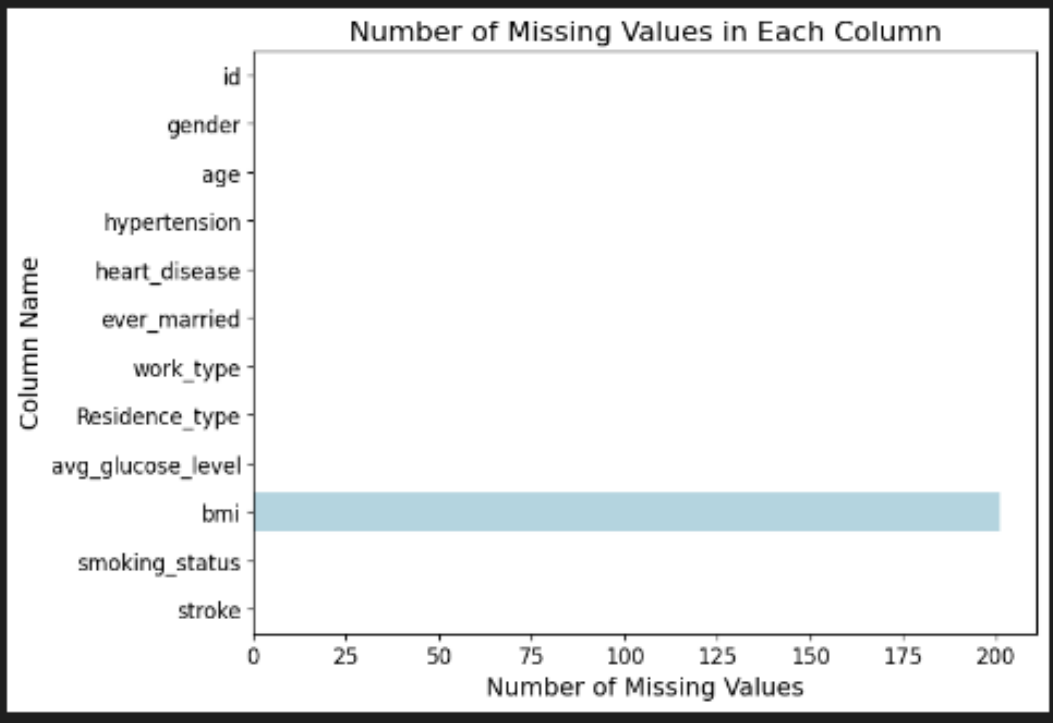
DATA PREPROCESSING AND CLEANING:

Missing Values Analysis:

- This analysis provides insights into the presence and distribution of missing values, aiding in data preprocessing and cleaning decisions. It is crucial for deciding on appropriate strategies for handling missing values during the data cleaning process.
- The code focuses on detecting missing values, with specific attention to the 'bmi' column
- Objective: Identify the extent of missing data in the 'bmi' attribute.
- Output: Revealed 201 missing values in the 'bmi' column.
- The below attached screenshot shows its output with code.

```python
# Printing the number of N/A values in each column
null_values = stroke_data.isna().sum()
print("Number of missing values in each column:\n", null_values)

# Graphical representation of the na values present in the attribute - bar graph
fig, ax = plt.subplots(figsize=(8,6))
sns.barplot(x=null_values.values, y=null_values.index, color='lightblue')
ax.set_title('Number of Missing Values in Each Column', fontsize=16)
ax.set_xlabel('Number of Missing Values', fontsize=14)
ax.set_ylabel('Column Name', fontsize=14)
ax.tick_params(axis='both', which='major', labelsize=12)
plt.show()
```

```
Number of missing values in each column:
 id                   0
gender                0
age                   0
hypertension          0
heart_disease         0
ever_married          0
work_type             0
Residence_type        0
avg_glucose_level     0
bmi                 201
smoking_status        0
stroke                0
dtype: int64
```



BMI Attribute Analysis:

- This analysis provides a quick insight into the extent of missing data in the 'bmi' column, aiding in decision-making for handling missing values during data preprocessing.
- Graphical representations of the BMI attribute aid in understanding its distribution and outliers:
- Objective: Visualize BMI distribution and identify outliers.

- Observations:
  Distribution is right-skewed.
  Outliers detected in the boxplot.

Outlier Detection and Handling:

The code employs the Interquartile Range (IQR) method to identify and handle outliers in the 'bmi' column:

- Objective: Detect and handle outliers in the 'bmi' attribute.
- Observations: 110 outliers identified.
- It is important to note that BMI is a crucial factor in determining the risk of stroke, and missing values can lead to inaccurate predictions. Therefore, it is efficient way to impute the missing values using appropriate techniques before performing any analysis or modeling on the data.

Attribute Analysis and Outlier Detection

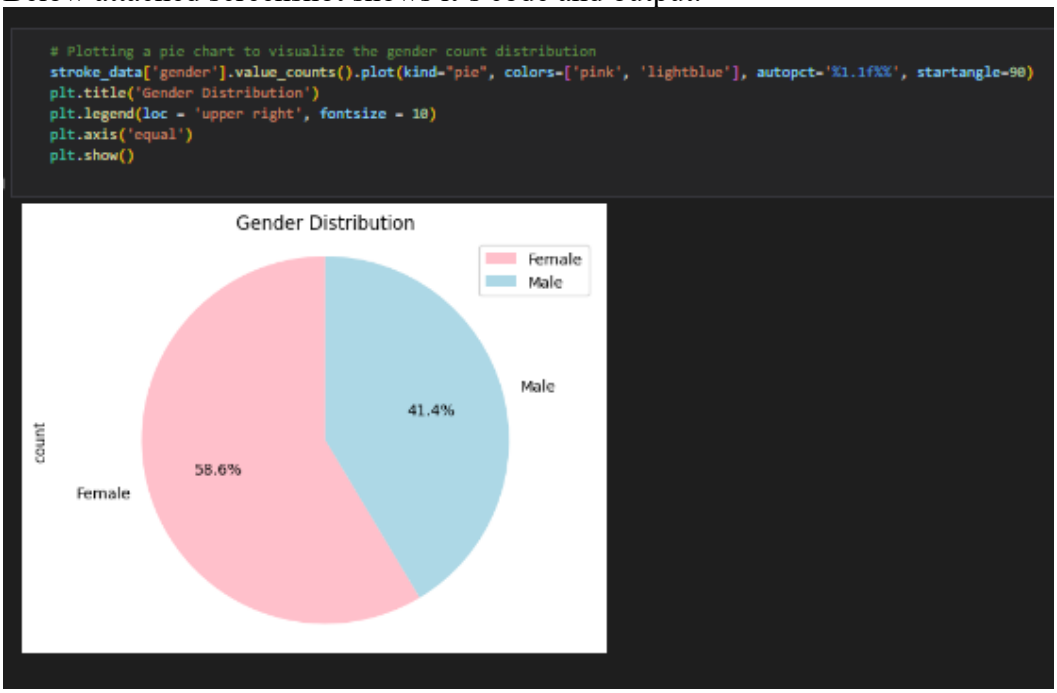
ID Analysis:

- The 'id' column, being unique for each row, holds no significance for analysis and is dropped for efficiency.

Gender Analysis:

- Distribution of values in the 'gender' column is checked.
- 'Other' instances are replaced with 'Male' to simplify and reduce dimensionality.
- Below attached screenshot shows it's code and output.

```
# Plotting a pie chart to visualize the gender count distribution
stroke_data['gender'].value_counts().plot(kind="pie", colors=['pink', 'lightblue'], autopct='%1.1f%%', startangle=90)
plt.title('Gender Distribution')
plt.legend(loc = 'upper right', fontsize = 10)
plt.axis('equal')
plt.show()
```



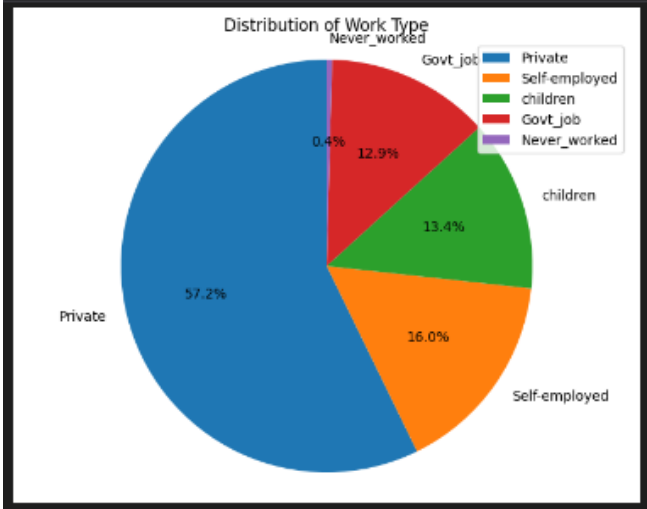
Hypertension Analysis:

- The 'hypertension' column is analyzed for the distribution of values.
- A bar graph illustrates the count of instances with and without hypertension.

Work-Type Analysis:

- The 'work_type' column is analyzed to understand the distribution of different work types.
- A pie chart visually represents the count of instances for each work type category in the dataset.
- Below attached screenshot shows it’s code and output.

```
# Graphical representation of the value counts of the work-type attribute
plt.figure(figsize=(8,6))
plt.pie(stroke_data['work_type'].value_counts(), labels=stroke_data['work_type'].value_counts().index, autopct='%1.1f%%', startangle=90)
plt.title('Distribution of Work Type')
plt.axis('equal')
plt.legend(loc = 'upper right', fontsize = 10)
plt.show()
```

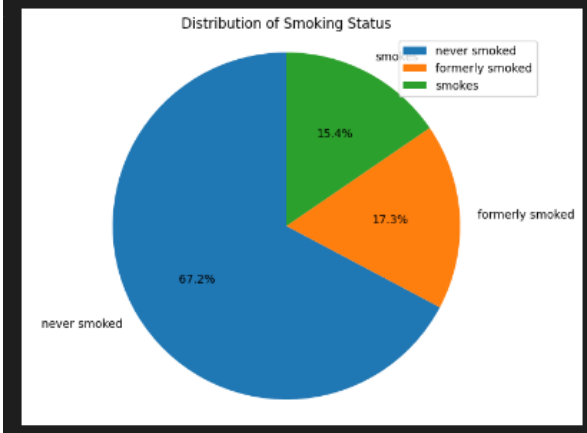


Smoking Status Analysis:

- The 'smoking_status' column is examined for the distribution of smoking statuses.
- 'Unknown' values are replaced with the mode, and the updated distribution is visualized using a pie chart.
- Below attached screenshot shows it’s code and output.

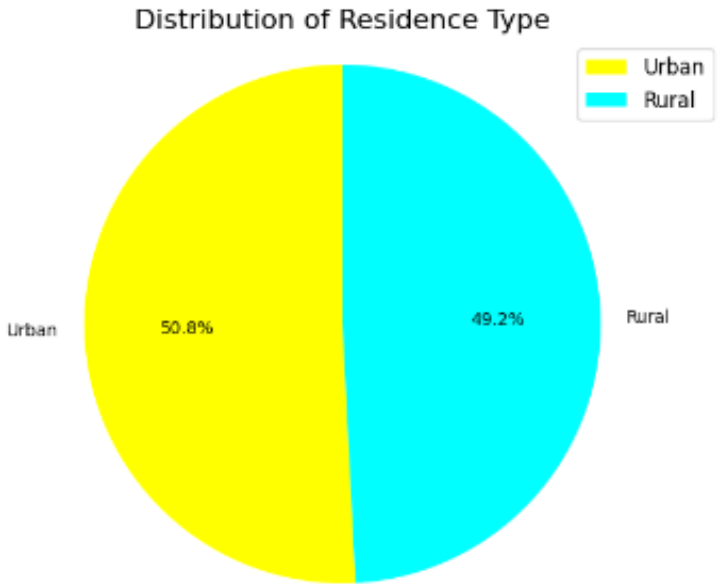
```
# Replace 'Unknown' values in the 'smoking_status' column with the mode
stroke_data['smoking_status'] = stroke_data['smoking_status'].replace('Unknown', stroke_data['smoking_status'].mode()[0])

# Print the value counts of the 'smoking_status' column after replacing 'Unknown' values with the mode
print("Value counts of 'smoking_status' column after replacing 'Unknown' values with the mode:", stroke_data['smoking_status'].value_counts())

# Visualize the distribution of smoking status using a pie chart
plt.figure(figsize=(8,6))
plt.pie(stroke_data['smoking_status'].value_counts(), labels=stroke_data['smoking_status'].value_counts().index, autopct='%1.1f%%', startangle=90)
plt.title('Distribution of Smoking Status')
plt.axis('equal')
plt.legend(loc = 'upper right', fontsize = 10)
plt.show()
```

```
Value counts of 'smoking_status' column after replacing 'Unknown' values with the mode: smoking_status
never smoked       3436
formerly smoked     885
smokes              789
Name: count, dtype: int64
```



Residence Type Analysis:

- The 'Residence_type' column is analyzed to understand the distribution of residence types.
- A pie chart visually represents the count of instances for each residence type category in the dataset.
- Below attached screenshot shows the distribution of residence type from our dataset.

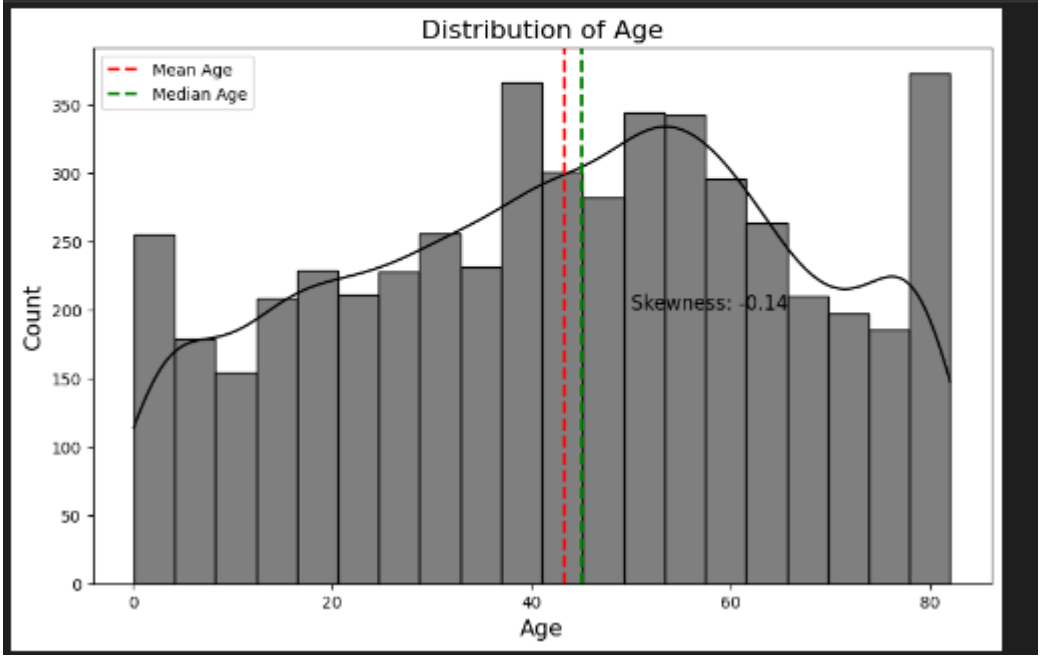


Age Analysis:

- The distribution of ages in the dataset is visualized using a histogram and boxplot.

- Mean and median lines provide insights into the central tendency and skewness of the age data.
- Below attached screenshot shows it’s code and output from our dataset.

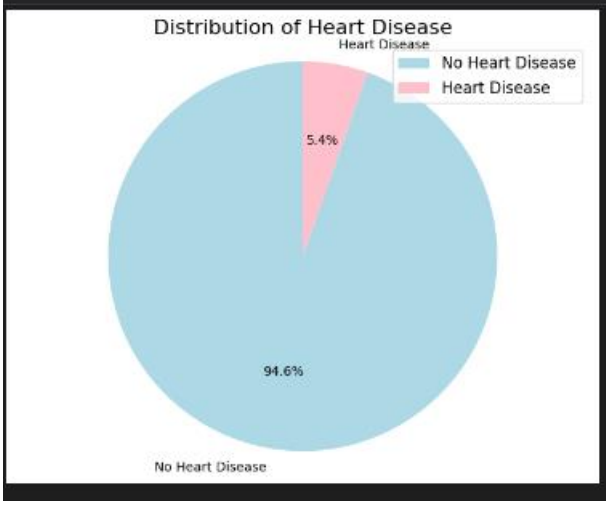
```
# Graphical representation fo the data in age column
plt.figure(figsize=(10,6))
sns.histplot(data=stroke_data['age'], color='black', bins=20,kde=True)
plt.xlabel('Age', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.title('Distribution of Age', fontsize=16)
plt.axvline(stroke_data['age'].mean(), color='red', linestyle='dashed', linewidth=2, label='Mean Age')
plt.axvline(stroke_data['age'].median(), color='green', linestyle='dashed', linewidth=2, label='Median Age')
plt.legend()
plt.text(50, 200, 'Skewness: {:.2f}'.format(stroke_data['age'].skew()), fontsize=12)
plt.show()
```



Heart Disease Analysis:

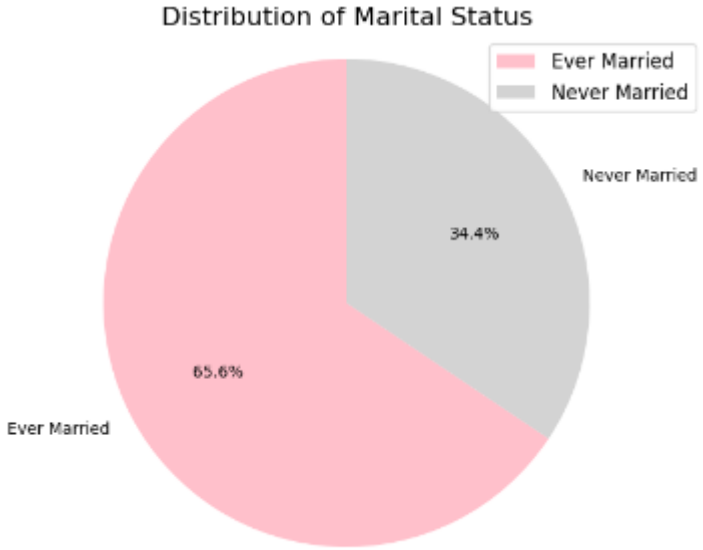
- The 'heart_disease' column is analyzed for the distribution of instances with and without heart disease.
- A pie chart visually represents the count of instances for each category in the dataset.
- Below attached screenshot shows its code and output from our dataset.

```
# Create a pie chart of the 'heart_disease' column
plt.figure(figsize=(8,6))
plt.pie(stroke_data['heart_disease'].value_counts(), labels=['No Heart Disease', 'Heart Disease'], autopct='%1.1f%%', startangle=90, colors=['lightblue', 'pink'])
plt.title('Distribution of Heart Disease', fontsize=16)
plt.axis('equal')
plt.legend(loc='upper right', fontsize=12)
plt.show()
```



Ever Married Analysis:

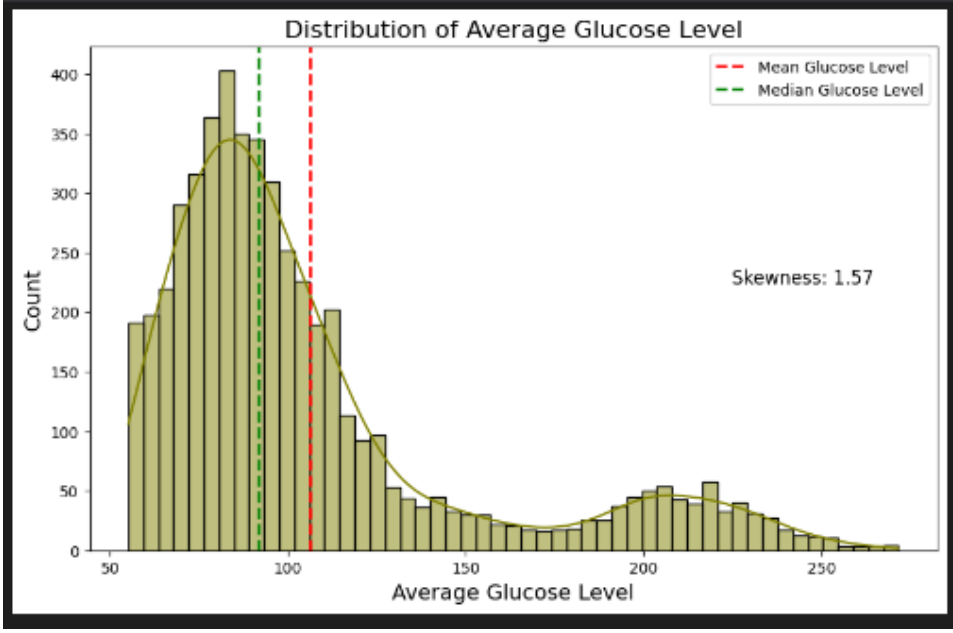
- The 'ever_married' column is analyzed to understand the distribution of individuals who have and haven't been married.
- A pie chart visually represents the count of instances for each category in the dataset. As, we have shown below the distribution from our dataset by implementing code.



Average Glucose Level Analysis:

- The distribution of average glucose levels in the dataset is visualized using a histogram and boxplot.
- Outliers are detected using the Interquartile Range (IQR) method, and the distribution is visualized with outliers highlighted.
- The below mentioned screenshot shows the distribution from dataset by implementing code.

```
# Graphical representation of the average glucose level using a histogram
plt.figure(figsize=(10,6))
sns.histplot(data=stroke_data['avg_glucose_level'], color='olive', kde=True)
plt.xlabel('Average Glucose Level', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.title('Distribution of Average Glucose Level', fontsize=16)
plt.axvline(stroke_data['avg_glucose_level'].mean(), color='red', linestyle='dashed', linewidth=2, label='Mean Glucose Level')
plt.axvline(stroke_data['avg_glucose_level'].median(), color='green', linestyle='dashed', linewidth=2, label='Median Glucose Level')
plt.legend()
plt.text(225, 224, 'Skewness: {:.2f}'.format(stroke_data['avg_glucose_level'].skew()), fontsize=12)
plt.show()
```



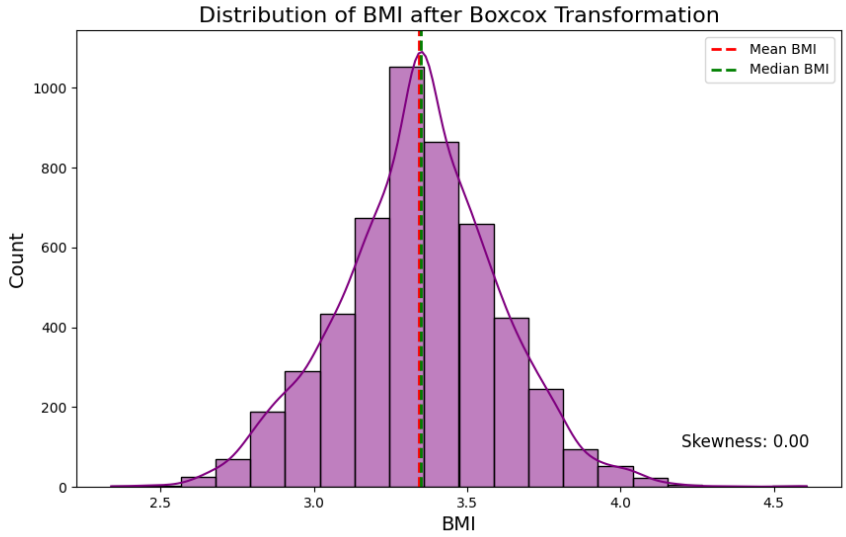
In short, the analysis of additional attributes provides a holistic understanding of the dataset. Outlier detection methods enhance data quality, ensuring reliable insights for further analysis or modeling.

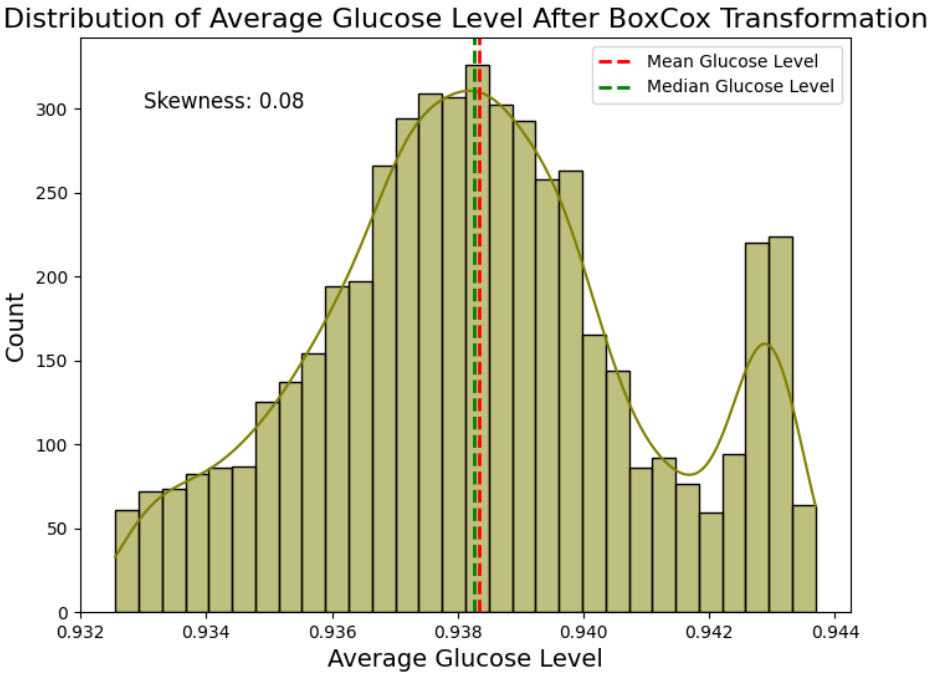
Skewness of Feature Variables:

- The skewness analysis reveals that both the 'bmi' and 'avg_glucose_level' columns exhibit right skewness, indicating a tail to the right in their distributions.
- Specifically, the 'bmi' column has a skewness of 1.06, while the 'avg_glucose_level' column has a skewness of 1.57. These numerical values quantify the extent of skewness, providing valuable information about the asymmetry in the respective data distributions. Understanding skewness is essential for making informed decisions during the modeling process, as it can impact the performance and interpretability of machine learning algorithms.
- To address the right skewness observed in the 'bmi' and 'avg_glucose_level' columns, we explore transformation techniques such as Box-Cox and log transformations.

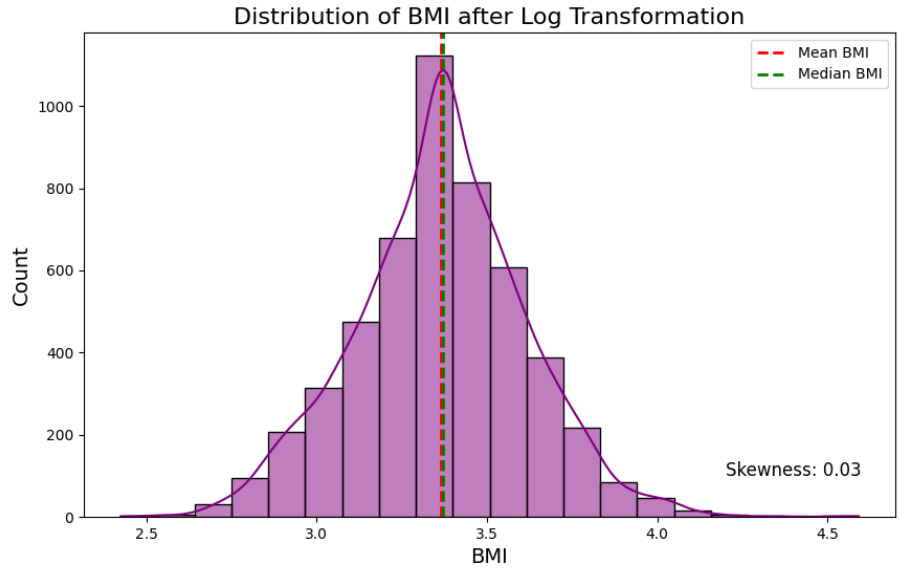
```
# Log transformation of the 'avg_glucose_level' column and 'bmi' column

#stroke_data['bmi'] = np.log1p(stroke_data['bmi'])
#stroke_data['avg_glucose_level'], _ = boxcox(stroke_data['avg_glucose_level'])

# Box-Cox transformation of the 'avg_glucose_level' column and 'bmi' column

#stroke_data['bmi'], _ = boxcox(stroke_data['bmi'])
#stroke_data['avg_glucose_level'] = np.log1p(stroke_data['avg_glucose_level'])
```

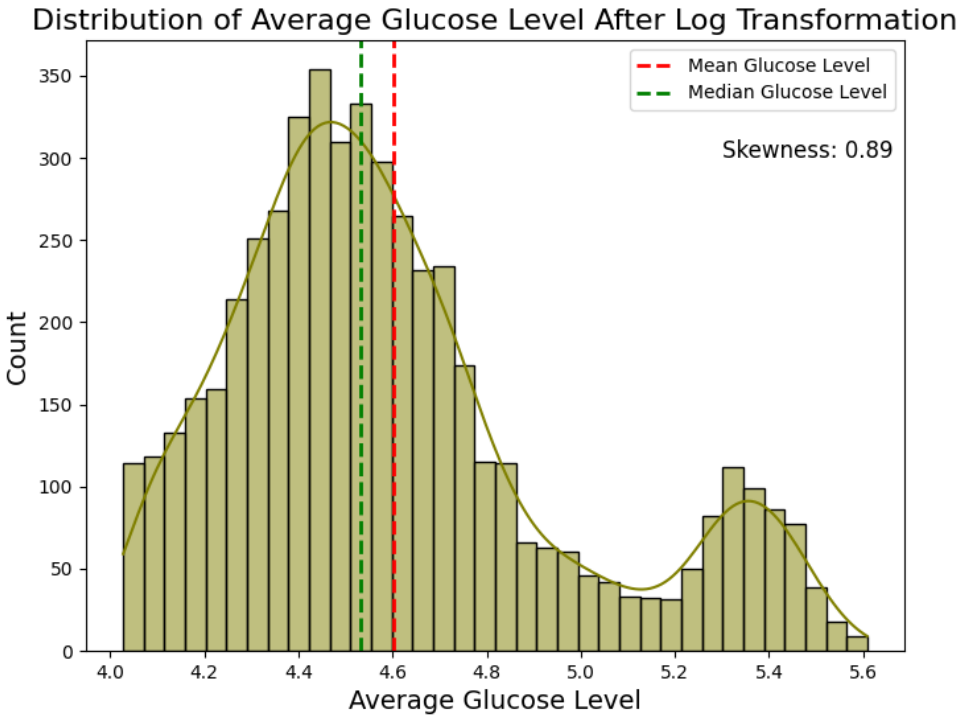
- Box-Cox Transformation: Graphical representation of BMI and Average Glucose Level Columns after the box-cox transformation.





- Log Transformation : Graphical representation of BMI and Average Glucose Level Columns after the log transformation.





Target Variable: Stroke Analysis

- The 'stroke' column is analyzed to understand the distribution of instances with and without stroke.
- The result provides the count of instances for each category in the dataset.

Percentage of People Who Had a Stroke: 4.87%

Percentage of People Who Did Not Have a Stroke: 95.13%

Skewness of Target Variable

Our target variable is highly skewed, with only around 5% of the instances experiencing a stroke. This presents a challenge when building a predictive model, as the model may be biased towards the majority class and perform poorly on the minority class.

To address this issue, we will need to perform necessary transformations to improve the representation of the minority class in our dataset. This may include techniques such as oversampling, undersampling, or generating synthetic samples using techniques such as SMOTE.

By improving the representation of the minority class, we can build a more accurate and robust predictive model that can identify individuals who are at risk of experiencing a stroke.

One Hot Encoding

One-hot encoding is a preprocessing technique applied to categorical columns, creating new binary columns for each unique value. This expands the feature space by converting categorical variables into a binary matrix. The shape of the dataset is printed to confirm the number of new columns created after one-hot encoding.

Oversampling

SMOTE (Synthetic Minority Over-sampling Technique) is applied to oversample the minority class (instances with stroke) and balance the dataset. This involves generating synthetic samples for the minority class to address the class imbalance. The shape and class distribution of the original and resampled datasets are printed for comparison.

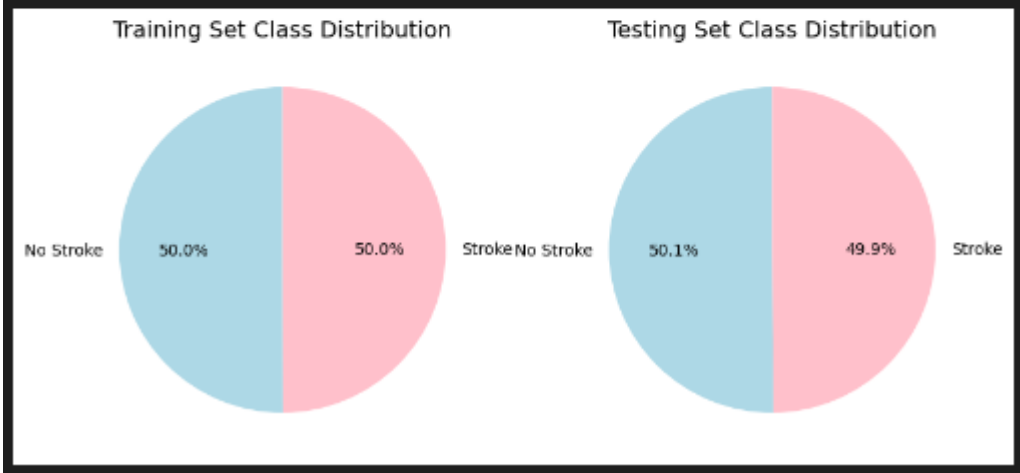
- **Stroke Distribution:**
    - Percentage of people who had a stroke: 4.87%
    - Percentage of people who did not have a stroke: 95.13%
    - Shape of encoded dataset: (5110, 15)
    - Shape of original dataset: (5110, 14)
    - Number of instances with stroke in the original dataset: 249
    - Shape of resampled dataset: (9722, 14)
    - Number of instances with stroke in the resampled dataset: 4861

Train Test Split

- The dataset is split into training and testing sets with a specified test size.
- The class distribution for each set is printed and visualized using pie charts.
- Split the data into training and testing sets in 80-20 split.
- The below attached screenshot shows the the code and its output from our dataset.

```python
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42)

# Print the number of instances in each class for the training and testing sets
train_counts = pd.Series(y_train).value_counts()
test_counts = pd.Series(y_test).value_counts()
print("Training set class distribution:\n", train_counts)
print("\nTesting set class distribution:\n", test_counts)

# Visualize the class distribution of the training and testing sets
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))
axs[0].pie(train_counts, labels=['No Stroke', 'Stroke'], autopct='%1.1f%%', startangle=90, colors=['lightblue', 'pink'])
axs[0].set_title('Training Set Class Distribution', fontsize=14)
axs[1].pie(test_counts, labels=['No Stroke', 'Stroke'], autopct='%1.1f%%', startangle=90, colors=['lightblue', 'pink'])
axs[1].set_title('Testing Set Class Distribution', fontsize=14)
plt.show()
```

```
Training set class distribution:
 stroke
1    3891
0    3886
Name: count, dtype: int64

Testing set class distribution:
 stroke
0    975
1    970
Name: count, dtype: int64
```



Feature Scaling

- In the feature scaling process, StandardScaler is employed to standardize the features in both the training and testing sets.
- The fit_transform() method is utilized on the training set, allowing the scaler to fit to the training data and simultaneously transform it.
- Subsequently, the transform() method is applied to the testing set, ensuring that the same scaling transformation used on the training set is consistently implemented.
- This standardization practice is crucial as it ensures that all features are brought to a uniform scale, preventing any particular feature from disproportionately influencing the model and enhancing the convergence of algorithms.

MODEL TRAINING

Decision Tree

- A Decision Tree model is a supervised machine learning algorithm used for both classification and regression tasks. It creates a tree-like structure where each internal node represents a decision based on a feature, each branch represents an outcome of that decision, and each leaf node represents a class label or regression value.

  1. Model Creation and Training: An instance of the DecisionTreeClassifier is created with a random state set to 42 for reproducibility. This classifier is then trained using standardized training data (X_train_std and y_train) via the fit method.

  2. Model Evaluation and Prediction: The trained Decision Tree model is used to predict outcomes on standardized testing data (X_test_std), generating predictions labeled as y_dt. Additionally, class probabilities for the testing data are computed using predict_proba.

  3. Performance Metrics Calculation: The model's predictive performance is assessed using various evaluation metrics:

    - Accuracy: The accuracy of the model's predictions on the testing dataset is calculated. It achieved the value of 93.06%.
    - Area Under the ROC Curve (AUC): This metric quantifies the model's ability to discriminate between classes by plotting the Receiver Operating Characteristic (ROC) curve and measuring the area under it. It achieved the value of 0.93.
    - F1 Score: It measures the balance between precision and recall and is useful in assessing the model's accuracy when dealing with imbalanced classes. It achieved the value of 0.93
    - Precision: The proportion of correctly predicted positive instances out of all instances predicted as positive. It achieved the value of 0.92
    - Recall (Sensitivity): The proportion of correctly predicted positive instances out of all actual positive instances. It achieved the value of 0.95
    - Sensitivity and Specificity: Sensitivity is the true positive rate, while specificity measures the true negative rate, aiding in understanding the model's performance on each class.

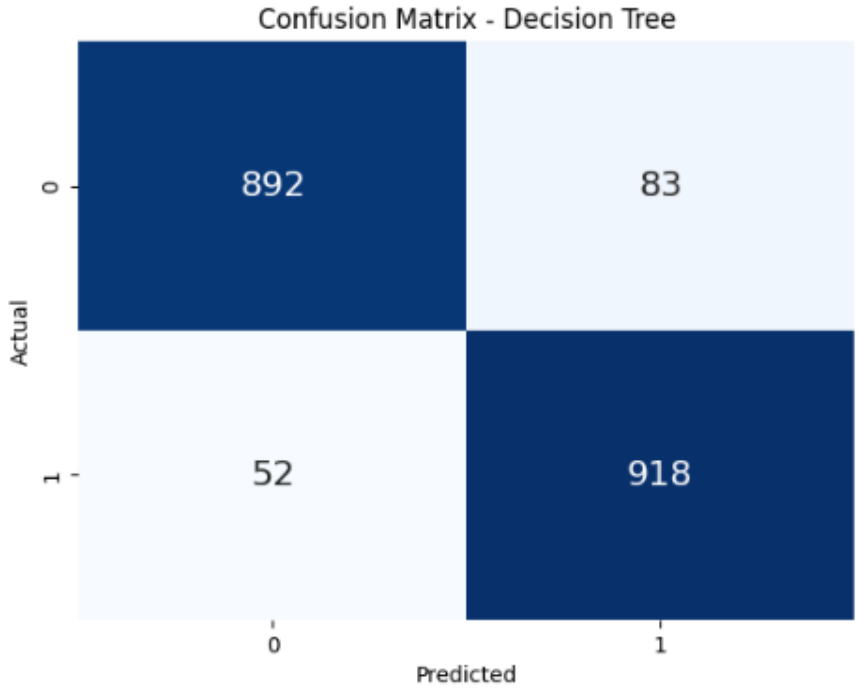
  4. Model Performance Summary: The performance metrics, including accuracy, AUC, F1 score, precision, recall, sensitivity, and specificity, are printed to provide a comprehensive assessment of the Decision Tree model's effectiveness in classification.

  5. Classification Report: The classification_report function generates a detailed report illustrating the precision, recall, F1 score, and support for each class in the dataset, offering a comprehensive overview of the model's performance across different classes.

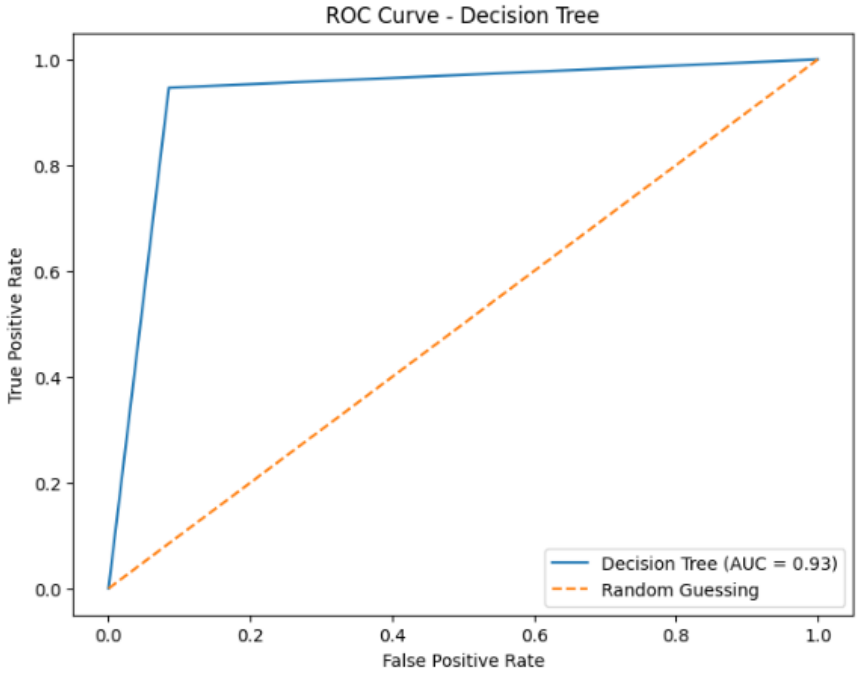
- The below code has been implemented.

```
# Create an instance of the DecisionTreeClassifier class with a random state of 42
model_dt = DecisionTreeClassifier(random_state=42)

# Fit the decision tree model to the standardized training data
model_dt.fit(X_train_std, y_train)

# Use the trained model to make predictions on the standardized testing data
y_dt = model_dt.predict(X_test_std)

# Use the trained model to predict the class probabilities for the standardized testing data
y_prob_dt = model_dt.predict_proba(X_test_std)

# Calculate the accuracy of the decision tree model on the testing data
accuracy_dt = accuracy_score(y_test, y_dt)

# Calculate the area under the ROC curve for the decision tree model on the testing data
auc_dt = roc_auc_score(y_test, y_prob_dt[:, 1])

# Calculate the F1 score for the decision tree model on the testing data
f1_dt = f1_score(y_test, y_dt)

# Calculate the precision for the decision tree model on the testing data
precision_dt = precision_score(y_test, y_dt)

# Calculate the recall for the decision tree model on the testing data
recall_dt = recall_score(y_test, y_dt)

# Print the accuracy, AUC, F1 score, precision, and recall for the decision tree model
print("Decision Tree Model")
print("Accuracy: {:.2f}%".format(accuracy_dt * 100))
print("AUC: {:.2f}".format(auc_dt))
print("F1 score: {:.2f}".format(f1_dt))
print("Precision: {:.2f}".format(precision_dt))
print("Recall: {:.2f}".format(recall_dt))

# Calculate the false positive rate and true positive rate for the decision tree model
fpr_dt, tpr_dt, thresholds_dt = roc_curve(y_test, y_prob_dt[:, 1])

# Calculate the sensitivity and specificity of the decision tree model
sensitivity_dt, specificity_dt = sens_specs_calculator(y_test, y_dt)

# Print the sensitivity and specificity of the decision tree model
print("Sensitivity: {:.2f}%".format(sensitivity_dt * 100))
print("Specificity: {:.2f}%".format(specificity_dt * 100))

# Print the classification report for the decision tree model
print("\nClassification Report:\n", classification_report(y_test, y_dt))
```

- Visual Representation of Classification Performance: The code generates a heatmap that visually represents the confusion matrix for the Decision Tree model. This matrix showcases the model's predictive performance by displaying the true positive, true negative, false positive, and false negative predictions.
    - Detailed Assessment of Model Performance: The heatmap with annotations allows for a comprehensive evaluation of the Decision Tree model's classification performance. It aids in assessing how well the model predicts different classes and provides insights into potential misclassifications, offering a clear overview of its strengths and weaknesses in classification tasks.
    - The below snapshot shows our output from code.



- Performance Assessment via ROC Curve: This code generates a Receiver Operating Characteristic (ROC) curve for the Decision Tree model. The curve illustrates the trade-off between the True Positive Rate (Sensitivity) and False Positive Rate (1 - Specificity) across different threshold values. It visually represents the model's performance in distinguishing between the positive and negative classes.
    - Model's Discrimination Ability The plotted ROC curve provides a succinct overview of the Decision Tree model's classification performance, accompanied by the Area under the Curve (AUC) value. The AUC quantifies the model's ability to separate the classes, with higher AUC values indicating better discrimination and a superior predictive model. The below snapshot shows our output from code.



Logistic Regression

- Logistic Regression is a statistical method used for binary classification problems, where the aim is to predict the probability of a particular outcome. Despite its name containing "regression," it's a classification algorithm
- Logistic Regression is a supervised learning algorithm utilized for binary classification tasks, predicting the probability of a categorical outcome or assigning observations to one of two classes (0 or 1).
- It models the relationship between a dependent binary variable and one or more independent variables by estimating the probability using the logistic function, also known as the sigmoid function. This function ensures that the output lies between 0 and 1, representing probabilities.
    - The implemented code of a Logistic Regression classifier using the LBFGS solver to predict outcomes in binary classification scenarios. Here's a breakdown of the process:
    - Model Configuration:
        - Solver Selection: The Logistic Regression model is instantiated with the LBFGS solver via LogisticRegression(solver='lbfgs', random_state=42). The LBFGS solver is an optimization algorithm suitable for logistic regression and often utilized for large datasets due to its efficiency in handling multi-class problems and convergence speed.
    - Model Training and Prediction:
        - Training the Model: The model is trained using the standardized training data (X_train_std and y_train) via the fit() method: model_lr.fit(X_train_std, y_train).
        - Making Predictions: Once trained, the model is utilized to predict outcomes for the standardized testing data (X_test_std) and to estimate class probabilities using predict() and predict_proba() methods

respectively: y_lr = model_lr.predict(X_test_std) and y_prob_lr = model_lr.predict_proba(X_test_std).

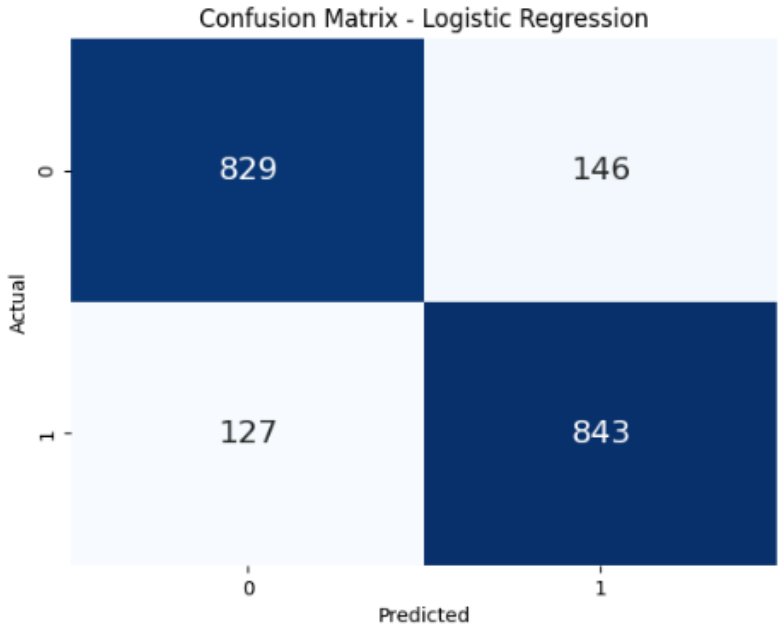
- Performance Evaluation:
  - Evaluation Metrics: Various performance metrics are calculated to evaluate the model's performance on the testing data:
  - Accuracy: It achieved the value of 85.96%
  - AUC: It achieved the value of 85.96%
  - F1 Score: It achieved the value of 0.86
  - Precision: It achieved the value of 0.85
  - Recall: It achieved the value of 0.87
  - Sensitivity and Specificity: It achieved the value of 86.91% and 85.03% respectively.
  - Metric Display: The computed metrics are displayed through a set of printed statements, offering an overall assessment of the model's predictive capability.
- Visual Representation:
  - ROC Curve: Additionally, an ROC (Receiver Operating Characteristic) curve is generated using roc_curve() and visualized with matplotlib. This graphical representation showcases the trade-off between true positive rate and false positive rate, aiding in assessing the model's discriminative ability across various thresholds.

- The below code has been implemented.

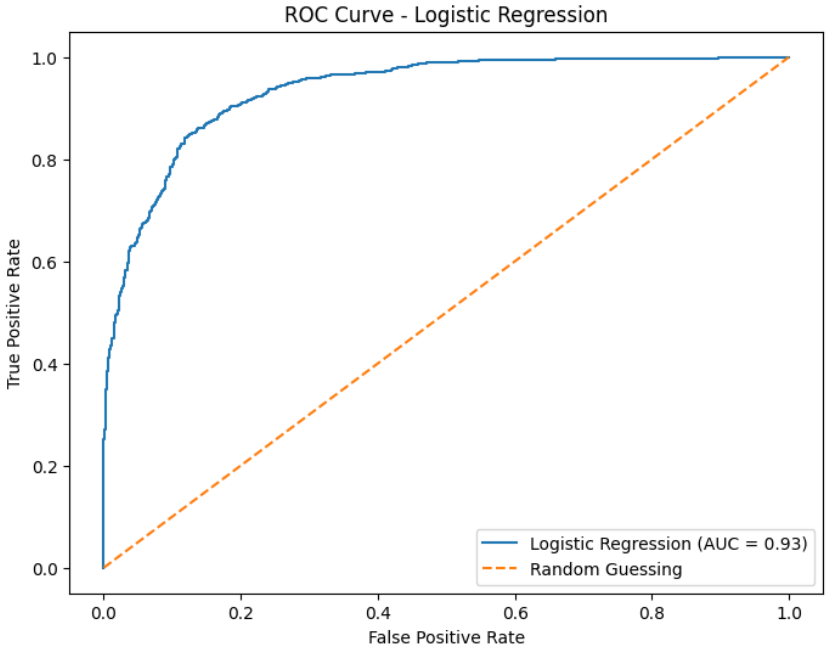
```python
# Create an instance of the logistic regression classifier with a lbfgs solver and random state of 42
model_lr = LogisticRegression(solver='lbfgs', random_state=42)

# Fit the logistic regression model to the standardized training data
model_lr.fit(X_train_std, y_train)

# Use the trained model to make predictions on the standardized testing data
y_lr = model_lr.predict(X_test_std)

# Use the trained model to predict the class probabilities for the standardized testing data
y_prob_lr = model_lr.predict_proba(X_test_std)

# Calculate the accuracy of the logistic regression model on the testing data
accuracy_lr = accuracy_score(y_test, y_lr)

# Calculate the area under the ROC curve for the logistic regression model on the testing data
auc_lr = roc_auc_score(y_test, y_prob_lr[:, 1])

# Calculate the F1 score for the logistic regression model on the testing data
f1_lr = f1_score(y_test, y_lr)

# Calculate the precision for the logistic regression model on the testing data
precision_lr = precision_score(y_test, y_lr)

# Calculate the recall for the logistic regression model on the testing data
recall_lr = recall_score(y_test, y_lr)

# Print the accuracy, AUC, F1 score, precision, and recall for the logistic regression model
print("Logistic Regression Model")
print("Accuracy: {:.2f}%".format(accuracy_lr * 100))
print("AUC: {:.2f}".format(auc_lr))
print("F1 score: {:.2f}".format(f1_lr))
print("Precision: {:.2f}".format(precision_lr))
print("Recall: {:.2f}".format(recall_lr))

# Calculate the false positive rate and true positive rate for the logistic regression model
fpr_lr, tpr_lr, thresholds_lr = roc_curve(y_test, y_prob_lr[:, 1])

# Calculate the sensitivity and specificity of the logistic regression model
sensitivity_lr, specificity_lr = sens_specs_calculator(y_test, y_lr)

# Print the sensitivity and specificity of the logistic regression model
print("Sensitivity: {:.2f}%".format(sensitivity_lr * 100))
print("Specificity: {:.2f}%".format(specificity_lr * 100))

# Print the classification report for the logistic regression model
print("\nClassification Report:\n", classification_report(y_test, y_lr))
```

- Visualizing Classification Performance of Confusion Matrix:
    - The confusion matrix heatmap vividly displays the model's classification performance by showcasing true positive, true negative, false positive, and false negative predictions.

    - Granular Assessment of Predictions:Through visual annotations in the heatmap cells, the confusion matrix provides a granular understanding of the model's accuracy in predicting actual classes versus predicted classes, aiding in the assessment of model reliability and error patterns. The below snapshot shows our output from code.



- Model Discrimination Assessment with ROC Curve: The ROC curve visually demonstrates the model's ability to discriminate between classes by displaying the trade-off between true positive rate and false positive rate across various threshold settings.
    - AUC Metric Representation: The Area Under the Curve (AUC) score, highlighted in the ROC plot, quantifies the model's performance, offering a single numerical value summarizing its ability to distinguish between classes. The below snapshot shows our output from code.

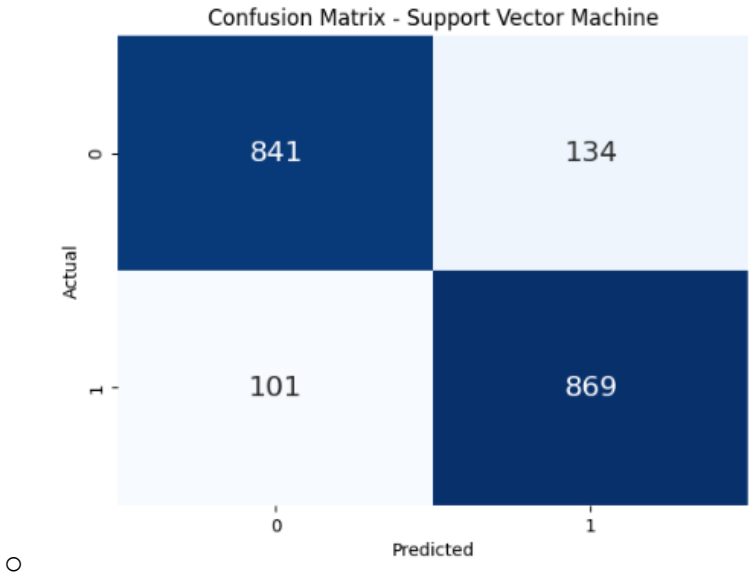


Support Vector Machine (SVM)

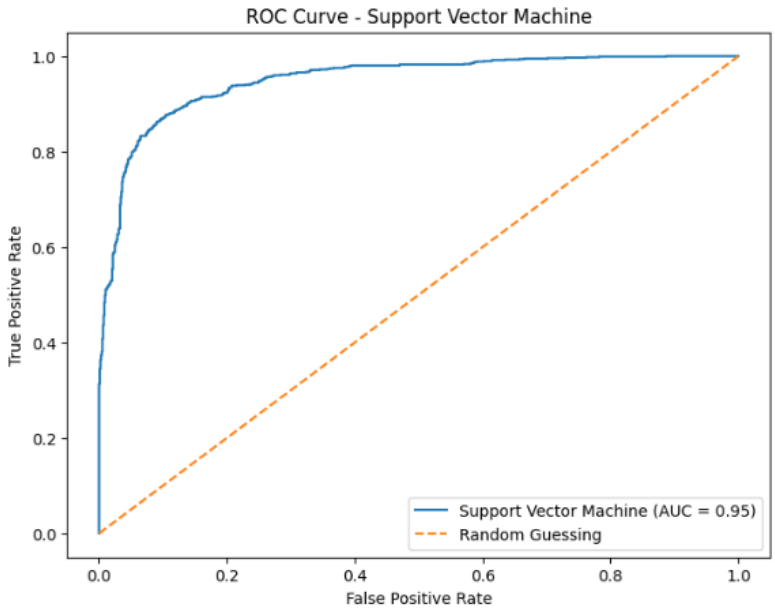
- It's a powerful supervised learning algorithm used for classification tasks.
- We have used an instance of the SVM classifier with a radial basis function (RBF) kernel, probability estimates enabled, and a random state of 42. It then fits the SVM model to the standardized training data, predicts on the standardized testing data, and calculates various evaluation metrics.
- The fit method: The fit method is applied to train the SVM model on the standardized training data (X_train_std and y_train).
- The trained model is used to predict the target values for the standardized testing data (X_test_std) using the predict method.
- Probability estimates are calculated for the testing data using the predict_proba method to evaluate the area under the ROC curve (AUC).
- Performance Metrics Computation:
    - Various performance metrics are calculated to evaluate the model's effectiveness on the test set:
    - Accuracy: The proportion of correctly predicted instances. It achieved the value of 87.92%
    - AUC (Area Under the ROC Curve): A metric assessing the model's ability to distinguish between classes. It achieved the value of 0.95
    - F1 Score: The harmonic mean of precision and recall, providing a balance between the two metrics. It achieved the value of 0.88
    - Precision: The proportion of true positive predictions among all positive predictions. It achieved the value of 0.87
    - Recall (or Sensitivity): The proportion of true positives correctly identified by the model. It achieved the value of 0.90
    - Specificity: The proportion of true negatives correctly identified by the model. It achieved the value of 86.26%
- The below code has been implemented.

```python
# Create an instance of the SVM classifier with a radial basis function (RBF) kernel, probability estimates enabled and a random state of 42
model_svm = SVC(kernel='rbf', probability=True, random_state=42)

# Fit the SVM model to the standardized training data
model_svm.fit(X_train_std, y_train)

# Use the trained model to make predictions on the standardized testing data
y_svm = model_svm.predict(X_test_std)

# Use the trained model to predict the class probabilities for the standardized testing data
y_prob_svm = model_svm.predict_proba(X_test_std)

# Calculate the accuracy of the SVM model on the testing data
accuracy_svm = accuracy_score(y_test, y_svm)

# Calculate the area under the ROC curve for the SVM model on the testing data
auc_svm = roc_auc_score(y_test, y_prob_svm[:, 1])

# Calculate the F1 score for the SVM model on the testing data
f1_svm = f1_score(y_test, y_svm)

# Calculate the precision for the SVM model on the testing data
precision_svm = precision_score(y_test, y_svm)

# Calculate the recall for the SVM model on the testing data
recall_svm = recall_score(y_test, y_svm)

# Print the accuracy, AUC, F1 score, precision, and recall for the SVM model
print("Support Vector Machine Model")
print("Accuracy: {:.2f}%".format(accuracy_svm * 100))
print("AUC: {:.2f}".format(auc_svm))
print("F1 score: {:.2f}".format(f1_svm))
print("Precision: {:.2f}".format(precision_svm))
print("Recall: {:.2f}".format(recall_svm))

# Calculate the false positive rate and true positive rate for the SVM model
fpr_svm, tpr_svm, thresholds_svm = roc_curve(y_test, y_prob_svm[:, 1])

# Calculate the sensitivity and specificity of the SVM model
sensitivity_svm, specificity_svm = sens_specs_calculator(y_test, y_svm)

# Print the sensitivity and specificity of the SVM model
print("Sensitivity: {:.2f}%".format(sensitivity_svm * 100))
print("Specificity: {:.2f}%".format(specificity_svm * 100))

# Print the classification report for the SVM model
print("\nClassification Report:\n", classification_report(y_test, y_svm))
```

- Visualization of SVM Model's Confusion Matrix: This code generates a heatmap representing the confusion matrix for the Support Vector Machine (SVM) model. The heatmap offers a visual representation of the model's performance, displaying the counts or proportions of true positive, true negative, false positive, and false negative predictions. Annotations within the heatmap cells indicate specific numeric values, allowing quick interpretation of the SVM model's classification accuracy and misclassifications.
    - Comprehensive Performance Assessment: The heatmap provides an intuitive and comprehensive way to evaluate the SVM model's classification performance. It offers insights into the model's ability to correctly classify instances and identifies potential misclassifications or errors in predictions, aiding in the assessment of the model's effectiveness. The below snapshot shows our output from code.



- Evaluation of SVM Model's Discriminative Power: This code generates a Receiver Operating Characteristic (ROC) curve for the Support Vector Machine (SVM) model. The curve illustrates the trade-off between true positive rate (sensitivity) and false positive rate (1 - specificity) at various classification thresholds. The Area Under the Curve (AUC) value quantifies the model's ability to distinguish between classes, with a higher AUC indicating better discriminative power.
    - Assessment of Model Performance: The ROC curve visually demonstrates the SVM model's performance in distinguishing between positive and negative classes. It provides a comprehensive overview of the model's classification accuracy across different thresholds, aiding in determining an optimal threshold for classification based on the problem's specific requirements. The below snapshot shows our output from code.



Gaussian Naïve Bayes (GNB)

- It demonstrates the utilization and evaluation of a Support Vector Machine (SVM) classifier, a powerful supervised learning algorithm commonly used for classification tasks. The below steps shows the model implement, performance and its usefulness for the data.

1. Model Instantiation:
    - An SVM model is created using the `SVC` class from Scikit-Learn.
    - The kernel parameter is set to `''rbf'`, indicating the adoption of a radial basis function (RBF) kernel. This choice allows the SVM to handle non-linear decision boundaries effectively.
    - `probability` is set to `True`, enabling the model to provide class probability estimates along with predictions.
    - The parameter `random_state` is set to `42`, ensuring reproducibility in results by fixing the random seed.

2. Training and Prediction:
    - The model is trained on standardized training data (`X_train_std` and `y_train`) using the `fit` method.
    - Subsequently, the trained model is utilized to predict the target variable on standardized testing data (`X_test_std`) via the `predict` method.
    - Additionally, the model computes class probabilities for the test set using `predict_proba`.

3. Performance Metrics Computation:
    - Various performance metrics are calculated to evaluate the model's effectiveness on the test set:
    - Accuracy: The proportion of correctly predicted instances. It achieved the value of 66.32%
    - AUC (Area Under the ROC Curve): A metric assessing the model's ability to distinguish between classes. It achieved the value of 0.89
    - F1 Score: The harmonic mean of precision and recall, providing a balance between the two metrics. It achieved the value of 0.74
    - Precision: The proportion of true positive predictions among all positive predictions. It achieved the value of 0.60
    - Recall (or Sensitivity): The proportion of true positives correctly identified by the model. It achieved the value of 0.97
    - Specificity: The proportion of true negatives correctly identified by the model.
        - It achieved the value of 35.69%

4. Model Evaluation Output:
    - The code prints the computed values for accuracy, AUC, F1 score, precision, recall, sensitivity, and specificity.
    - Additionally, it generates a detailed classification report, which includes precision, recall, F1 score, and support for each class.

- This comprehensive evaluation helps in understanding the SVM model's predictive performance, its capability to classify instances accurately, and its proficiency in distinguishing between different classes within the dataset. The metrics provided offer an insightful overview of the model's strengths and areas for improvement, aiding in making informed decisions about its application and potential adjustments.

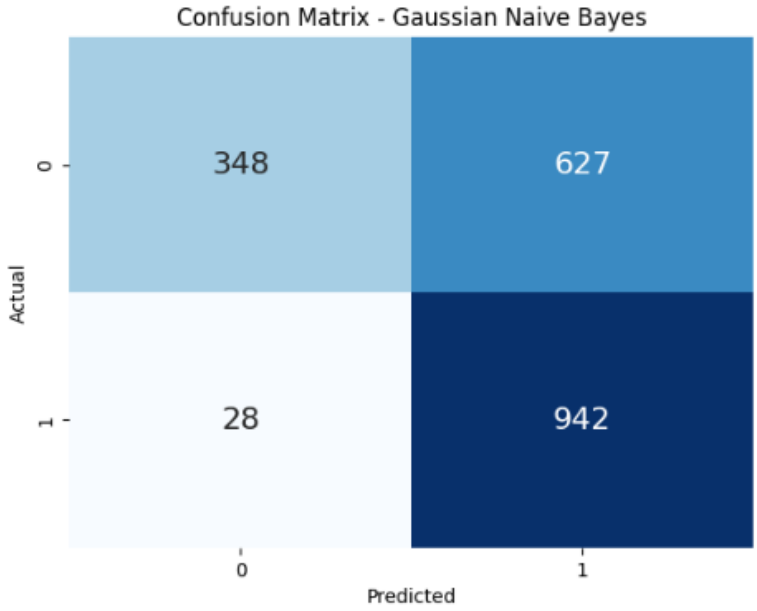
- The below code has been implemented.

```
# Create an instance of the Gaussian Naive Bayes (GNB) classifier
model_gnb = GaussianNB()

# Fit the GNB model to the standardized training data
model_gnb.fit(X_train_std, y_train)

# Use the trained model to make predictions on the standardized testing data
y_gnb = model_gnb.predict(X_test_std)

# Use the trained model to predict the class probabilities for the standardized testing data
y_prob_gnb = model_gnb.predict_proba(X_test_std)

# Calculate the accuracy of the GNB model on the testing data
accuracy_gnb = accuracy_score(y_test, y_gnb)

# Calculate the area under the ROC curve for the GNB model on the testing data
auc_gnb = roc_auc_score(y_test, y_prob_gnb[:, 1])

# Calculate the F1 score for the GNB model on the testing data
f1_gnb = f1_score(y_test, y_gnb)

# Calculate the precision for the GNB model on the testing data
precision_gnb = precision_score(y_test, y_gnb)

# Calculate the recall for the GNB model on the testing data
recall_gnb = recall_score(y_test, y_gnb)

# Print the accuracy, AUC, F1 score, precision, and recall for the GNB model
print("Gaussian Naive Bayes Model")
print("Accuracy: {:.2f}%".format(accuracy_gnb * 100))
print("AUC: {:.2f}".format(auc_gnb))
print("F1 score: {:.2f}".format(f1_gnb))
print("Precision: {:.2f}".format(precision_gnb))
print("Recall: {:.2f}".format(recall_gnb))

# Calculate the false positive rate and true positive rate for the GNB model
fpr_gnb, tpr_gnb, thresholds_gnb = roc_curve(y_test, y_prob_gnb[:, 1])

# Calculate the sensitivity and specificity of the GNB model
sensitivity_gnb, specificity_gnb = sens_specs_calculator(y_test, y_gnb)

# Print the sensitivity and specificity of the GNB model
print("Sensitivity: {:.2f}%".format(sensitivity_gnb * 100))
print("Specificity: {:.2f}%".format(specificity_gnb * 100))

# Print the classification report for the GNB model
print("\nClassification Report:\n", classification_report(y_test, y_gnb))
```

- Confusion Matrix Generation: The code computes a confusion matrix using the confusion_matrix function from Scikit-Learn, capturing the GNB model's performance by summarizing its predictive outcomes.

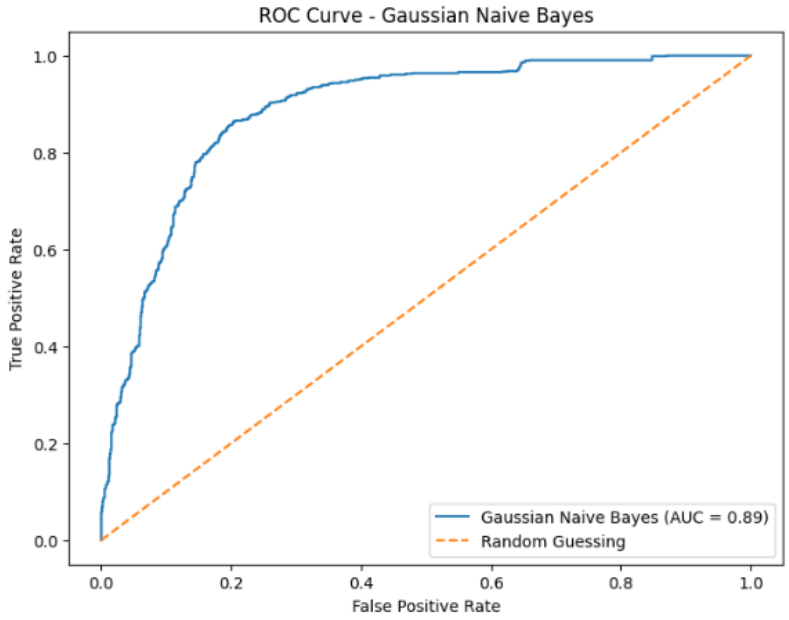
    1. Heatmap Visualization: Utilizing Seaborn's sns.heatmap, a heatmap representation of the confusion matrix is created, offering a visual depiction of the GNB model's classification accuracy and error rates.

        - Matrix Annotations**:** The heatmap is annotated with numeric values, aiding in better interpretation by displaying the counts of true positive, true negative, false positive, and false negative predictions within the cells. The below snapshot shows our output from code.



- ROC Curve Generation: The code generates a Receiver Operating Characteristic (ROC) curve to assess the performance of the Gaussian Naive Bayes (GNB) model.

    1. Model Evaluation: The curve illustrates the trade-off between true positive rate (sensitivity) and false positive rate (1 - specificity), offering insights into the GNB model's classification threshold.
    2. AUC Calculation: The Area Under the Curve (AUC) value is displayed in the plot's legend, representing the GNB model's overall performance in distinguishing between classes.
    3. Random Guessing Baseline: The dashed line represents the performance of a random classifier, aiding in the visualization of the model's effectiveness compared to random chance.
        - Visualization Clarity: The graph, with labeled axes and a title, provides a clear and concise visual summary of the GNB model's discriminatory power, aiding in quick model evaluation. The below snapshot shows our output from code.



Random Forest (RF):

- The implemented code demonstrates building a Random Forest classifier, evaluating its performance using multiple metrics, and assessing its predictive power on the test data. The below steps shows the model implement, performance and its usefulness for the data.

  1. Model Initialization:Initializes a Random Forest Classifier setting 100 trees, using entropy as the criterion for splitting nodes, and a specific random state (42) for reproducibility.

  2. Model Training: Fits the Random Forest model to standardized training data (`X_train_std`, `y_train`).

  3. Prediction: Uses the trained model to make predictions (`y_rf`) on standardized testing data (`X_test_std`).

  4. Probability Estimation: Predicts class probabilities (`y_prob_rf`) for the standardized testing data.

  5. Performance Metrics: Computes various performance metrics:

    - Accuracy: Measures the accuracy of the model's predictions on the testing data. It achieved the value of 95.01%
    - AUC: Computes the Area Under the ROC Curve (AUC) to evaluate the model's discriminatory capability. It achieved the value of 0.99
    - F1 Score: Calculates the F1 score, which combines precision and recall into a single metric. It achieved the value of 0.95
    - Precision and Recall: Measures precision (accuracy of positive predictions) and recall (true positive rate) of the model are as 0.95 and 0.95

  6. Sensitivity and Specificity Calculation: Computes sensitivity (true positive rate) and specificity (true negative rate) for the Random Forest model.

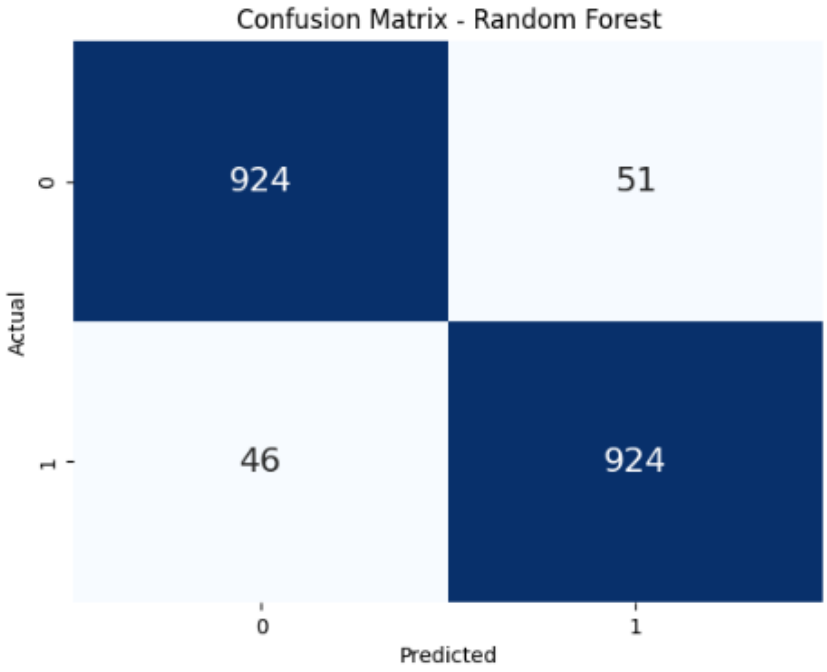
7. Classification Report: Prints a detailed classification report summarizing various classification metrics like precision, recall, and F1-score for both classes.

- The below code has been implemented.

```
# Create an instance of the random forest classifier a criterion of entropy and a random state of 42
model_rf = RandomForestClassifier(n_estimators=100,criterion='entropy', random_state=42)

# Fit the random forest model to the standardized training data
model_rf.fit(X_train_std, y_train)

# Use the trained model to make predictions on the standardized testing data
y_rf = model_rf.predict(X_test_std)

# Use the trained model to predict the class probabilities for the standardized testing data
y_prob_rf = model_rf.predict_proba(X_test_std)

# Calculate the accuracy of the random forest model on the testing data
accuracy_rf = accuracy_score(y_test, y_rf)

# Calculate the area under the ROC curve for the random forest model on the testing data
auc_rf = roc_auc_score(y_test, y_prob_rf[:, 1])

# Calculate the F1 score of the random forest model on the testing data
f1_rf = f1_score(y_test, y_rf)

# Calculate the precision of the random forest model on the testing data
precision_rf = precision_score(y_test, y_rf)

# Calculate the recall of the random forest model on the testing data
recall_rf = recall_score(y_test, y_rf)

# Print the accuracy, AUC, F1 score, precision, and recall for the random forest model
print("Random Forest Model")
print("Accuracy: {:.2f}%".format(accuracy_rf * 100))
print("AUC: {:.2f}".format(auc_rf))
print("F1 Score: {:.2f}".format(f1_rf))
print("Precision: {:.2f}".format(precision_rf))
print("Recall: {:.2f}".format(recall_rf))

# Calculate the false positive rate and true positive rate for the random forest model
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, y_prob_rf[:, 1])

# Calculate the sensitivity and specificity of the random forest model
sensitivity_rf, specificity_rf = sens_specs_calculator(y_test, y_rf)

# Print the sensitivity and specificity of the random forest model
print("Sensitivity: {:.2f}%".format(sensitivity_rf * 100))
print("Specificity: {:.2f}%".format(specificity_rf * 100))

# Print the classification report for the random forest model
print("\nClassification Report:\n", classification_report(y_test, y_rf))
```

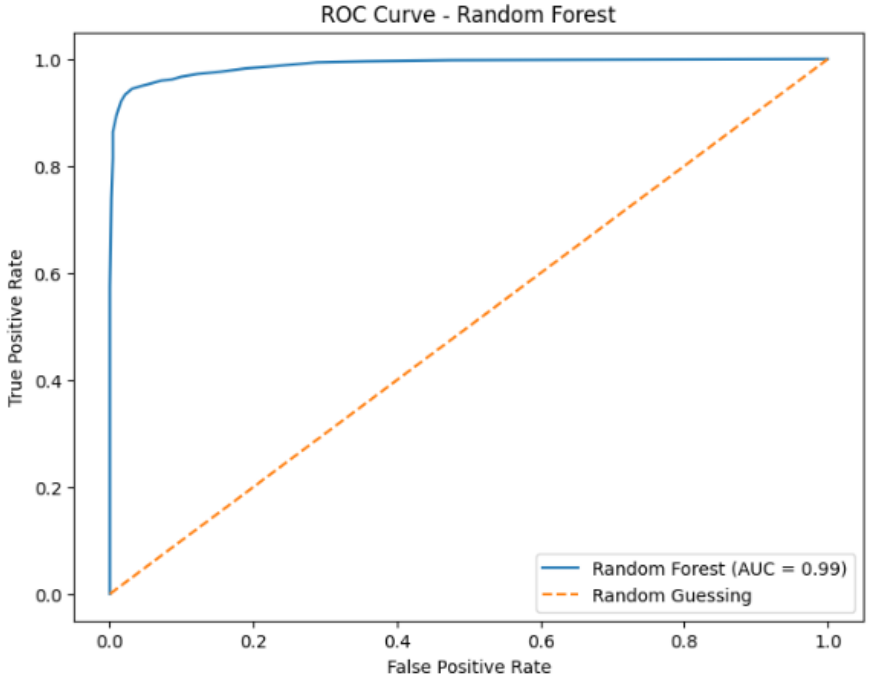
- Confusion Matrix Visualization: Generates a heatmap to visualize the confusion matrix for the Random Forest model.
    - Understanding Model Performance: The confusion matrix provides insights into the performance of the Random Forest model by displaying the count of true positive, true negative, false positive, and false negative predictions.

- Heatmap Plotting: Utilizes Seaborn's heatmap function (sns.heatmap) to create a visual representation of the confusion matrix. Notable parameters used:
    - conf_matrix_rf: Computed confusion matrix for the Random Forest model.
    - annot=True: Shows numeric values within the heatmap cells.
    - cmap='Blues': Color scheme used in the heatmap.
    - fmt='g': Format specification for the annotations (general numeric format).
    - cbar=False: Hides the color bar.
    - annot_kws={"fontsize":16}: Annotation style settings adjusting font size.
- Interpretation Assistance: The heatmap assists in understanding the model's performance by emphasizing correctly and incorrectly classified instances, aiding in the evaluation of predictive accuracy.
- The below snapshot shows our output from code.



- ROC Curve Visualization: The code generates a Receiver Operating Characteristic (ROC) curve for evaluating the performance of the Random Forest model
    - Understanding Model Performance: The ROC curve illustrates the trade-off between the true positive rate (sensitivity) and the false positive rate (1 - specificity) across various threshold values.
    - Plotting the ROC Curve: Utilizes Matplotlib to create the ROC curve. Important elements include:
    - plt.plot(fpr_rf, tpr_rf, label='Random Forest (AUC = {:.2f})'.format(auc_rf)): Plots the ROC curve, displaying the relationship between false positive rate (x-axis) and true positive rate (y-axis) with the AUC value in the legend.
    - plt.plot([0, 1], [0, 1], linestyle='--', label='Random Guessing'): Represents the baseline performance of a random classifier.

- Interpretation Assistance: The ROC curve demonstrates the model's ability to distinguish between classes by showcasing its trade-off between sensitivity and specificity, providing a comprehensive view of the model's classification accuracy.
- The below snapshot shows our output from code.



K –NEAREST NEIGHBORS (KNN)

- The K-Nearest Neighbors (KNN) algorithm is a non-parametric, instance-based learning method used for classification and regression tasks.

1. Classifier Initialization: Initializes the KNN classifier object with a parameter setting of 2 neighbors.

2. Training Phase: Trains the KNN classifier using the `fit` method on the provided training dataset (`X_train` and `y_train`).

3. Prediction: Utilizes the trained KNN model to predict the target labels for the testing dataset (`X_test`). The `predict` method generates predictions based on the trained model.

4. Probability Estimation Utilizes the trained KNN model to predict the class probabilities for the testing data using `predict_proba`.

5. Performance Metrics Calculation: Computes various performance metrics to evaluate the KNN model's performance on the testing dataset:

- Accuracy: Measures the ratio of correctly predicted instances. It achieved the value of 92.29%

- Area Under the ROC Curve (AUC): Quantifies the model's ability to distinguish between classes. It achieved the value of 0.94
- F1 Score: Harmonic mean of precision and recall. It achieved the value of 0.92

- Precision: Measures the ratio of correctly predicted positive observations to the total predicted positives. It achieved the value of 0.99
- Recall: Calculates the proportion of actual positives that were correctly predicted. It achieved the value of 0.95
- Sensitivity and Specificity: Additional metrics for binary classification evaluation.

6. Printed Summary: Displays a summary of key metrics (Accuracy, AUC, F1 Score, Precision, Recall, Sensitivity, Specificity) for the KNN model's performance on the testing data.

7. ROC Curve Calculation: Computes the false positive rate (fpr), true positive rate (tpr), and thresholds to plot the Receiver Operating Characteristic (ROC) curve.

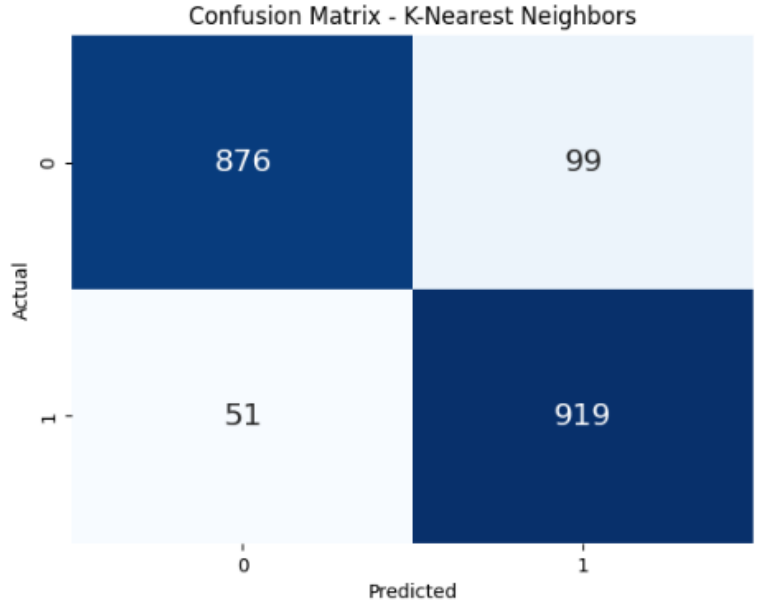
8. Classification Report: Generates a detailed classification report providing precision, recall, F1-score, and support for each class.

- The below code has been implemented.

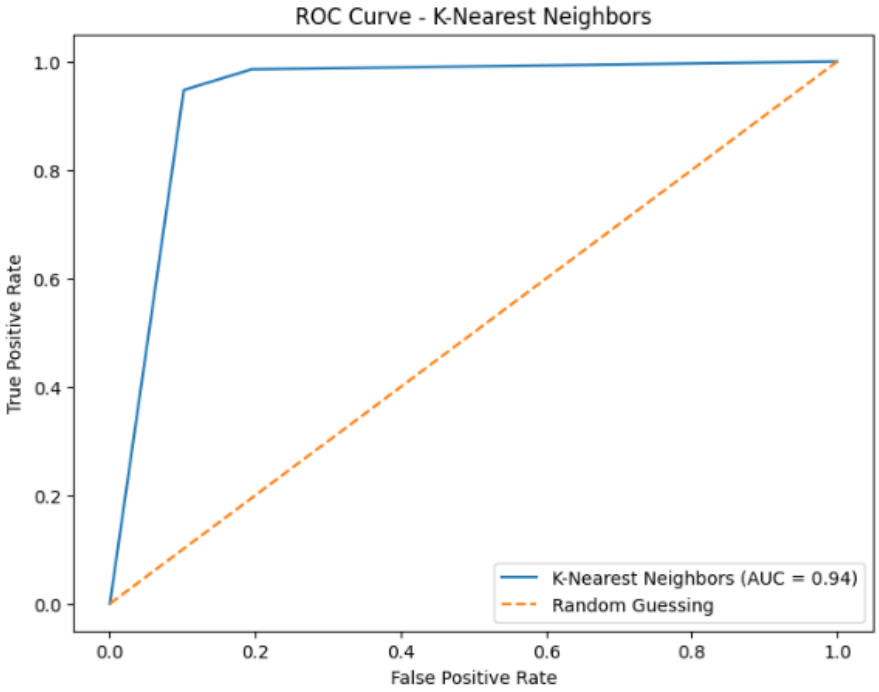
```
# Create the KNN classifier object with 2 neighbors
model_knn = KNeighborsClassifier(n_neighbors=2)

# Train the classifier on the training data
model_knn.fit(X_train, y_train)

# Use the trained model to make predictions on the testing data
y_knn = model_knn.predict(X_test)

# Use the trained model to predict the class probabilities for the testing data
y_prob_knn = model_knn.predict_proba(X_test)[:, 1]

# Calculate the accuracy of the KNN model on the testing data
accuracy_knn = accuracy_score(y_test, y_knn)

# Calculate the area under the ROC curve for the KNN model on the testing data
auc_knn = roc_auc_score(y_test, y_prob_knn)

# Calculate the F1 score of the KNN model on the testing data
f1_knn = f1_score(y_test, y_knn)

# Calculate the precision of the KNN model on the testing data
precision_knn = precision_score(y_test, y_knn)

# Calculate the recall of the KNN model on the testing data
recall_knn = recall_score(y_test, y_knn)

# Print the accuracy, AUC, F1 score, precision, and recall for the KNN model
print("K-Nearest Neighbors Model")
print("Accuracy: {:.2f}%".format(accuracy_knn * 100))
print("AUC: {:.2f}".format(auc_knn))
print("F1 Score: {:.2f}".format(f1_knn))
print("Precision: {:.2f}".format(precision_knn))
print("Recall: {:.2f}".format(recall_knn))

# Calculate the false positive rate and true positive rate for the KNN model
fpr_knn, tpr_knn, thresholds_knn = roc_curve(y_test, y_prob_knn)

# Calculate the sensitivity and specificity of the KNN model
sensitivity_knn, specificity_knn = sens_specs_calculator(y_test, y_knn)

# Print the sensitivity and specificity of the KNN model
print("Sensitivity: {:.2f}%".format(sensitivity_knn * 100))
print("Specificity: {:.2f}%".format(specificity_knn * 100))

# Print the classification report for the KNN model
print("\nClassification Report:\n", classification_report(y_test, y_knn))
```

- Visualizing Confusion Matrix: The code generates a heatmap to visually represent the confusion matrix for the K-Nearest Neighbors (KNN) model.
    - Insightful Visualization: Utilizes the Seaborn library to create a heatmap that encapsulates the confusion matrix, making it easier to comprehend model performance.
    - Matrix Annotations: Annotations are included within the heatmap cells (**annot=True**) to display the numeric values, aiding in a clear understanding of predictions.

- Customization**:** Custom color scheme (**cmap='Blues'**) enhances readability, using varying shades of blue to denote different values.
- Informative Labels: Axis labels ('Predicted' and 'Actual') and a descriptive title ('Confusion Matrix - K-Nearest Neighbors') provide context and clarity to the generated visualization.
- The below snapshot shows our output from code.



- ROC Curve Visualization: This code generates an ROC (Receiver Operating Characteristic) curve, a graphical representation illustrating the trade-off between true positive rate (sensitivity) and false positive rate.
  - Model Performance: The curve showcases the model's ability to discriminate between classes, plotting the true positive rate against the false positive rate across different threshold values.
  - Area Under the Curve (AUC): Incorporates the AUC value into the plot (`AUC = {:.2f}`), quantifying the model's performance with a summarized metric.
  - Random Guessing Baseline: Demonstrates the baseline for a random model with a diagonal dashed line (`linestyle='--'`), aiding in assessing the model's effectiveness compared to random chance.
  - Clear Labels and Title: Axes labels ('False Positive Rate' and 'True Positive Rate') and a descriptive title ('ROC Curve - K-Nearest Neighbors') enhance understanding and interpretation of the curve.
  - The below snapshot shows our output from code.



Light Gradient Boosting Machine (LGBM)

- Light Gradient Boosting Machine (LGBM) is a machine learning algorithm based on gradient boosting framework that utilizes tree-based learning algorithms. It's known for its efficiency, speed, and high performance in handling large datasets.

1. Model Training: An LGBMClassifier instance is created with 100 trees and a random state of 42, using the `n_estimators` parameter to define the number of boosting iterations (trees) in the model.

2. Model Performance Metrics: Various performance metrics are calculated to evaluate the LGBM model on the standardized testing data. These metrics include:

   - Accuracy: Measures the overall correctness of the model's predictions. It achieved the value of 95.17%
   - AUC (Area Under the Curve): Represents the model's ability to distinguish between classes across different threshold values. It achieved the value of 0.99
   - F1 Score: Balances precision and recall, providing a harmonic mean to gauge the model's accuracy. It achieved the value of 0.95
   - Precision: Indicates the ratio of correctly predicted positive observations to the total predicted positives. It achieved the value of 0.95
   - Recall (Sensitivity): Measures the proportion of actual positives correctly identified by the model. It achieved the value of 0.96

3. Evaluation of Specific Metrics: Specificity and sensitivity are computed for further understanding of the model's performance in correctly classifying negative and positive instances, respectively.

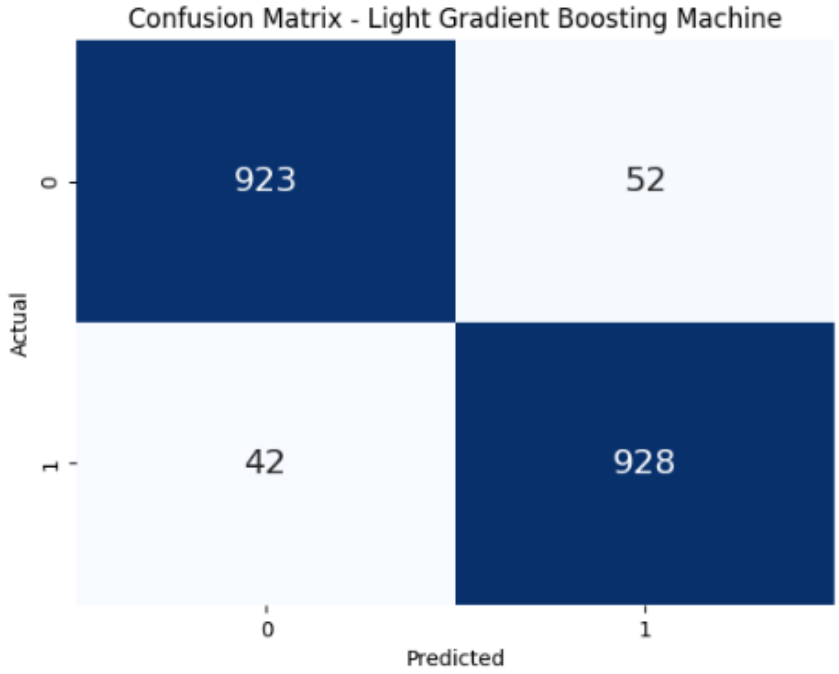
4. Classification Report: The detailed classification report is printed, displaying precision, recall, F1-score, and support for both classes (positive and negative).

5. Model Assessment: These metrics collectively assess the predictive power and robustness of the LGBM model in handling the given testing data, providing comprehensive insights into its performance across various evaluation criteria.

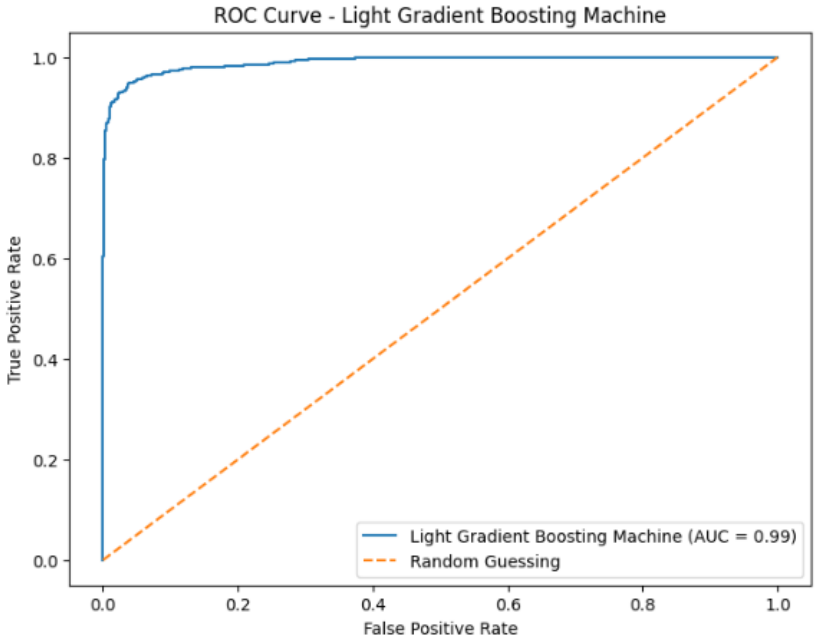
- The below code has been implemented.

```
# Create the KNN classifier object with 2 neighbors
model_knn = KNeighborsClassifier(n_neighbors=2)

# Train the classifier on the training data
model_knn.fit(X_train, y_train)

# Use the trained model to make predictions on the testing data
y_knn = model_knn.predict(X_test)

# Use the trained model to predict the class probabilities for the testing data
y_prob_knn = model_knn.predict_proba(X_test)[:, 1]

# Calculate the accuracy of the KNN model on the testing data
accuracy_knn = accuracy_score(y_test, y_knn)

# Calculate the area under the ROC curve for the KNN model on the testing data
auc_knn = roc_auc_score(y_test, y_prob_knn)

# Calculate the F1 score of the KNN model on the testing data
f1_knn = f1_score(y_test, y_knn)

# Calculate the precision of the KNN model on the testing data
precision_knn = precision_score(y_test, y_knn)

# Calculate the recall of the KNN model on the testing data
recall_knn = recall_score(y_test, y_knn)

# Print the accuracy, AUC, F1 score, precision, and recall for the KNN model
print("K-Nearest Neighbors Model")
print("Accuracy: {:.2f}%".format(accuracy_knn * 100))
print("AUC: {:.2f}".format(auc_knn))
print("F1 Score: {:.2f}".format(f1_knn))
print("Precision: {:.2f}".format(precision_knn))
print("Recall: {:.2f}".format(recall_knn))

# Calculate the false positive rate and true positive rate for the KNN model
fpr_knn, tpr_knn, thresholds_knn = roc_curve(y_test, y_prob_knn)

# Calculate the sensitivity and specificity of the KNN model
sensitivity_knn, specificity_knn = sens_specs_calculator(y_test, y_knn)

# Print the sensitivity and specificity of the KNN model
print("Sensitivity: {:.2f}%".format(sensitivity_knn * 100))
print("Specificity: {:.2f}%".format(specificity_knn * 100))

# Print the classification report for the KNN model
print("\nClassification Report:\n", classification_report(y_test, y_knn))
```

- Visualization of Predictive Performance of Confusion Matrix: The heatmap representation of the confusion matrix vividly demonstrates the predictive performance of the LGBM model on the testing data.
    - Comprehensive Assessment: Through color-coded cells and annotations, the matrix provides a comprehensive view of true positives, true negatives, false positives, and false negatives, aiding in understanding the model's accuracy and error types.
    - The below snapshot shows our output from code.



- Model Performance Evaluation for ROC Curve: The ROC curve visually represents the trade-off between true positive rate (sensitivity) and false positive rate. It illustrates the model's ability to discriminate between classes and its overall performance.
    - AUC Metric Assessment: The Area under the Curve (AUC) value quantifies the model's performance. A higher AUC closer to 1 indicates better predictive ability, and the AUC value presented on the plot helps in comparing and evaluating the model against random guessing.
    - The below snapshot shows our output from code.



XGBoost (XGB)

- XGBoost (Extreme Gradient Boosting) is a popular machine learning algorithm used for supervised learning tasks, especially in regression and classification problems. It's a type of gradient boosting algorithm that has gained significant popularity due to its speed and performance in various data science competitions.

    1. Model Initialization: An instance of the XGBoost classifier (XGBClassifier) is created with specific settings. The objective is set to "binary:logistic" indicating a binary classification problem. The random_state parameter is set to 42 to ensure reproducibility.
    2. Model Training: The XGBoost model is fitted or trained using the standardized training data (X_train_std, y_train).
    3. Prediction and Evaluation: After training, the model is used to make predictions on the standardized testing data (X_test_std). Predictions are generated both in terms of class labels (y_xgb) and class probabilities (y_prob_xgb).
    4. Performance Metrics Calculation: Several performance metrics are computed to assess the model's effectiveness:
        - Accuracy (accuracy_xgb): It achieved the value of 95.42%
        - Area Under the ROC Curve (AUC) (auc_xgb): It achieved the value of 0.99
        - F1 Score (f1_xgb): It achieved the value of 0.95
        - Precision (precision_xgb): It achieved the value of 0.95
        - Recall (recall_xgb): It achieved the value of 0.96
        - Sensitivity and Specificity are also calculated separately using a custom function (sens_specs_calculator).
    5. Result Presentation: The script outputs the computed metrics such as accuracy, AUC, F1 score, precision, recall, sensitivity, specificity, and a classification report showing precision, recall, F1 score, and support for

each class in the test set. The printed information summarizes the XGBoost model's performance in classification tasks on the test dataset.

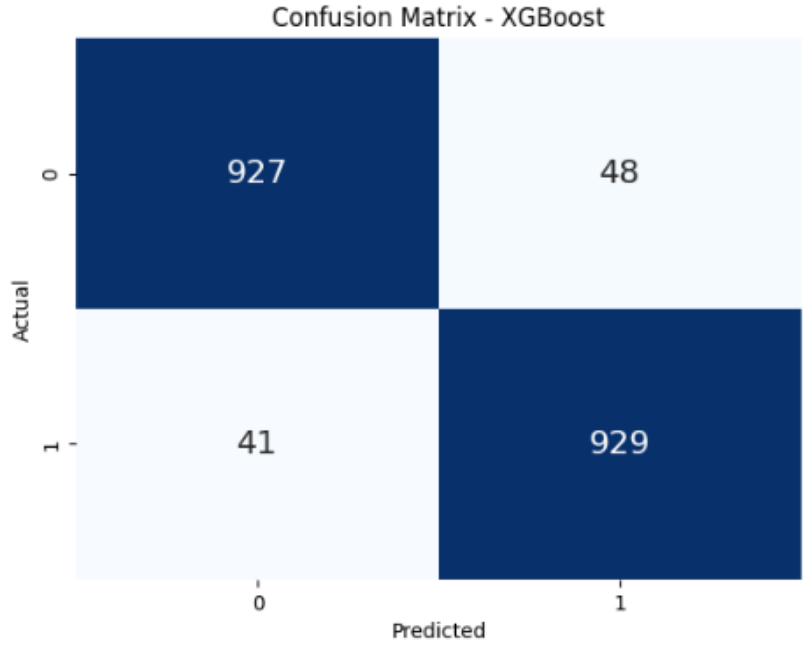
- The below code has been implemented.

```
# Create an instance of the XGBClassifier with the objective of "binary:logistic" and a random state of 42
model_xgb = XGBClassifier(objective="binary:logistic", random_state=42)

# Fit the XGB model to the standardized training data
model_xgb.fit(X_train_std, y_train)

# Use the trained model to make predictions on the standardized testing data
y_xgb = model_xgb.predict(X_test_std)

# Use the trained model to predict the class probabilities for the standardized testing data
y_prob_xgb = model_xgb.predict_proba(X_test_std)

# Calculate the accuracy of the XGB model on the testing data
accuracy_xgb = accuracy_score(y_test, y_xgb)

# Calculate the area under the ROC curve for the XGB model on the testing data
auc_xgb = roc_auc_score(y_test, y_prob_xgb[:, 1])

# Calculate the F1 score of the XGB model on the testing data
f1_xgb = f1_score(y_test, y_xgb)

# Calculate the precision of the XGB model on the testing data
precision_xgb = precision_score(y_test, y_xgb)

# Calculate the recall of the XGB model on the testing data
recall_xgb = recall_score(y_test, y_xgb)

# Print the accuracy, AUC, F1 score, precision, and recall for the XGB model
print("XGBoost Model")
print("Accuracy: {:.2f}%".format(accuracy_xgb * 100))
print("AUC: {:.2f}".format(auc_xgb))
print("F1 Score: {:.2f}".format(f1_xgb))
print("Precision: {:.2f}".format(precision_xgb))
print("Recall: {:.2f}".format(recall_xgb))

# Calculate the false positive rate and true positive rate for the XGB model
fpr_xgb, tpr_xgb, thresholds_xgb = roc_curve(y_test, y_prob_xgb[:, 1])

# Calculate the sensitivity and specificity of the XGB model
sensitivity_xgb, specificity_xgb = sens_specs_calculator(y_test, y_xgb)

# Print the sensitivity and specificity of the XGB model
print("Sensitivity: {:.2f}%".format(sensitivity_xgb * 100))
print("Specificity: {:.2f}%".format(specificity_xgb * 100))

# Print the classification report for the XGB model
print("\nClassification Report:\n", classification_report(y_test, y_xgb))
```

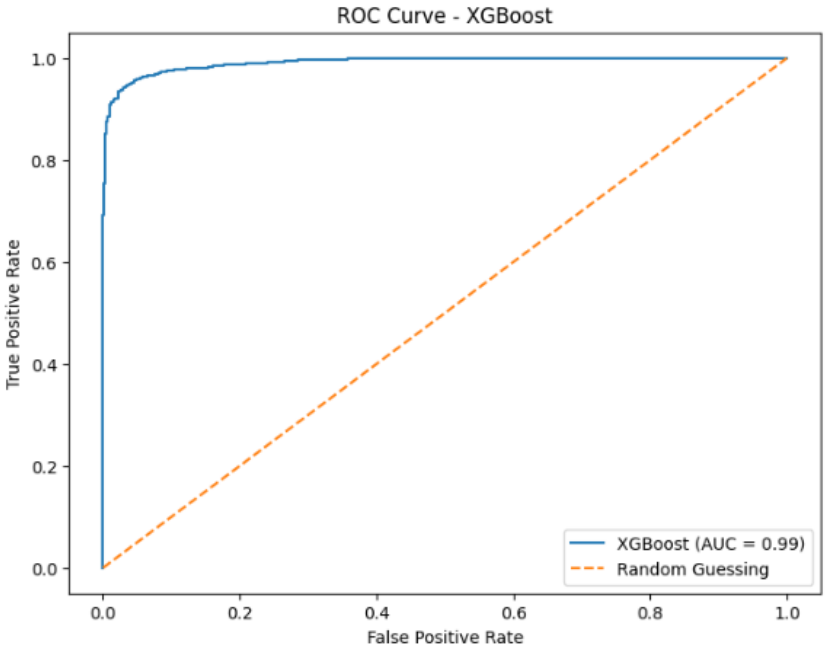
- Confusion Matrix Visualization: This code generates a visual representation of the confusion matrix for the XGBoost (XGB) model. It employs a heatmap using the Seaborn library to display the confusion matrix. Each cell in the heatmap corresponds to the count of predictions, allowing for a clear visualization of true positives, true negatives, false positives, and false negatives made by the XGBoost model on the test dataset.
    - Insightful Representation: The plotted heatmap with annotations offers a concise and intuitive way to understand the model's performance in terms

of classification accuracy. The heatmap's color intensity and annotated values provide a quick insight into the model's predictive strengths and weaknesses, aiding in the assessment of its performance on the test data.

- The below snapshot shown our output from code.



- ROC Curve for XGBoost Model: This code generates the Receiver Operating Characteristic (ROC) curve for the XGBoost (XGB) model. It plots the true positive rate against the false positive rate, showcasing the model's performance across various classification thresholds. The curve illustrates how well the model distinguishes between classes, with the AUC (Area Under the Curve) providing a quantifiable measure of its predictive accuracy.
    - Performance Visualization: By visualizing the ROC curve, this plot allows a quick assessment of the XGBoost model's discrimination ability and effectiveness in differentiating between positive and negative classes. The AUC value, depicted in the legend, quantifies the overall performance of the model – a higher AUC indicates better predictive performance of the XGBoost model on the test dataset.
    - The below snapshot shows our output from code.



Performance Benchmark Across Models

- We generated comparative bar plots showcasing performance metrics across various machine learning models, specifically focusing on metrics like AUC, accuracy, F1 score, precision, sensitivity, and specificity. Additionally, it generates a summary table of these metrics for each model. We have attached the snapshot to show performs from output of code.

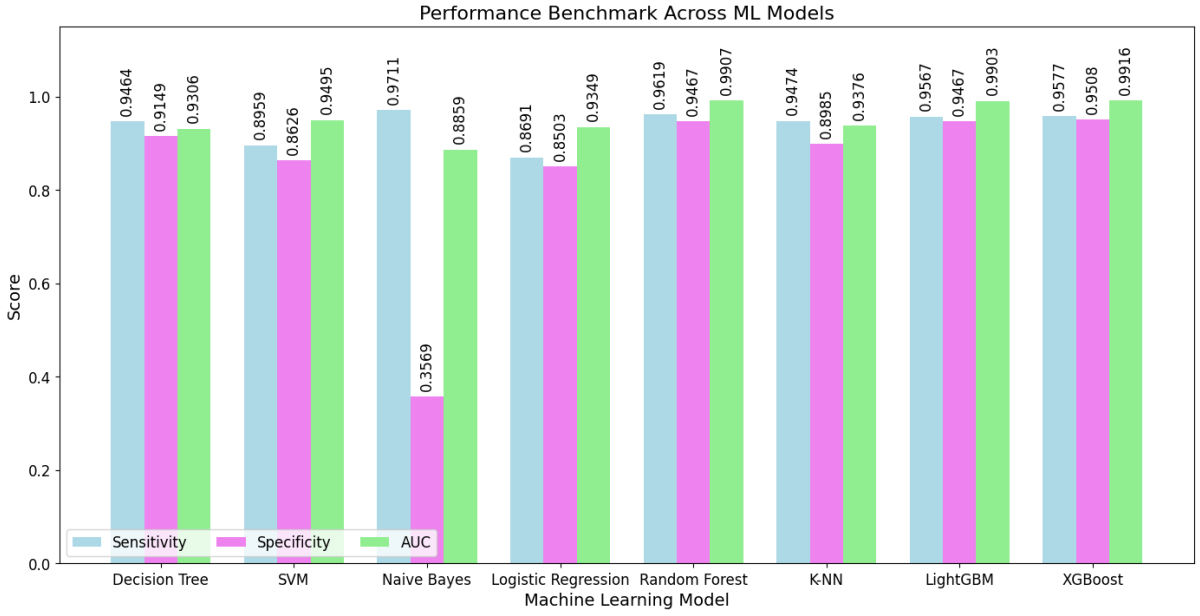


Figure 1: Sensitivity, Specificity, and AUC for each machine learning model.

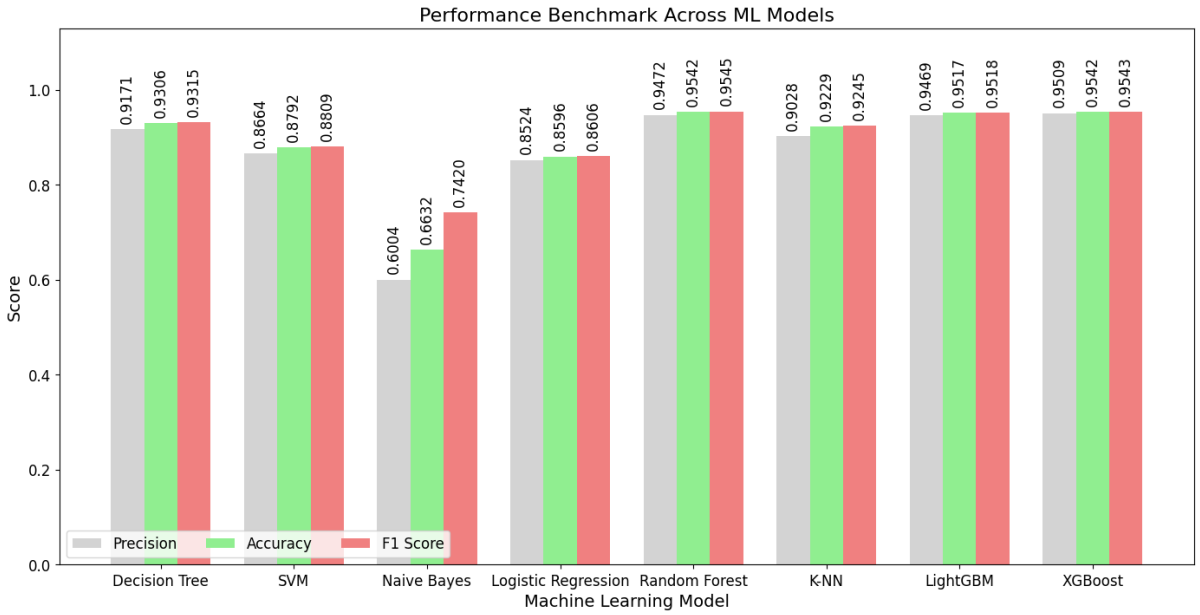


Figure 2: Precision, Accuracy, and F1 score for each machine learning model.

PERFORMANCE TABLE FOR THE MODELS

Model	AUC	Accuracy	F1 Score	Precision	Sensitivity	Specificity
Decision Tree	0.9306	0.9306	0.9415	0.9171	0.9464	0.9149
SVM	0.9495	0.8792	0.8809	0.8664	0.8959	0.8626
Naive Bayes	0.8859	0.6632	0.7420	0.6004	0.9711	0.3569
Logistic Regression	0.9349	0.8596	0.8606	0.8524	0.8691	0.8503
Random Forest	0.9907	0.9542	0.9534	0.9472	0.9619	0.9467
K-NN	0.9376	0.9229	0.9245	0.9028	0.9474	0.8985
LightGBM	0.9903	0.9903	0.9518	0.9469	0.9567	0.9467
XGBoost	0.9916	0.9542	0.9543	0.9509	0.9577	0.9508

PERFORMANCE BENCHMARK ACROSS MODELS WITH SKEWNESS CORRECTION

Transformation	Decision Tree	SVM	Naive Bayes	Logistic Regression	Random Forest	K-NN	LightGBM	XGBoost
No Preprocessing	0.9306	0.8792	0.6632	0.8596	0.9542	0.9229	0.9517	0.9542
Glucose Log Transformation	0.9244	0.8771	0.6668	0.8488	0.9506	0.9378	0.9434	0.9491
Glucose Log Transformation and BMI Log Transformation	0.9033	0.8581	0.6632	0.8144	0.9404	0.9254	0.9270	0.9301
BMI Log Transformation	0.9219	0.8735	0.6658	0.8432	0.9512	0.9224	0.9450	0.9496
Glucose Box-Cox Transformation and BMI Log Transformation	0.9141	0.8514	0.6663	0.8165	0.9332	0.9177	0.9270	0.9414
BMI Box-Cox Transformation and Glucose Log Transformation	0.9033	0.8576	0.6638	0.8144	0.9388	0.9260	0.9280	0.9342
Glucose Box-Cox Transformation	0.9270	0.8761	0.6653	0.8509	0.9537	0.9326	0.9486	0.9527
BMI Box-Cox Transformation	0.9213	0.8735	0.6658	0.8432	0.9506	0.9219	0.9393	0.9517
Glucose Box-Cox Transformation and BMI Box-Cox Transformation	0.9147	0.8514	0.6663	0.8165	0.9357	0.9183	0.9301	0.9414

Graphical Representation for all Variants:

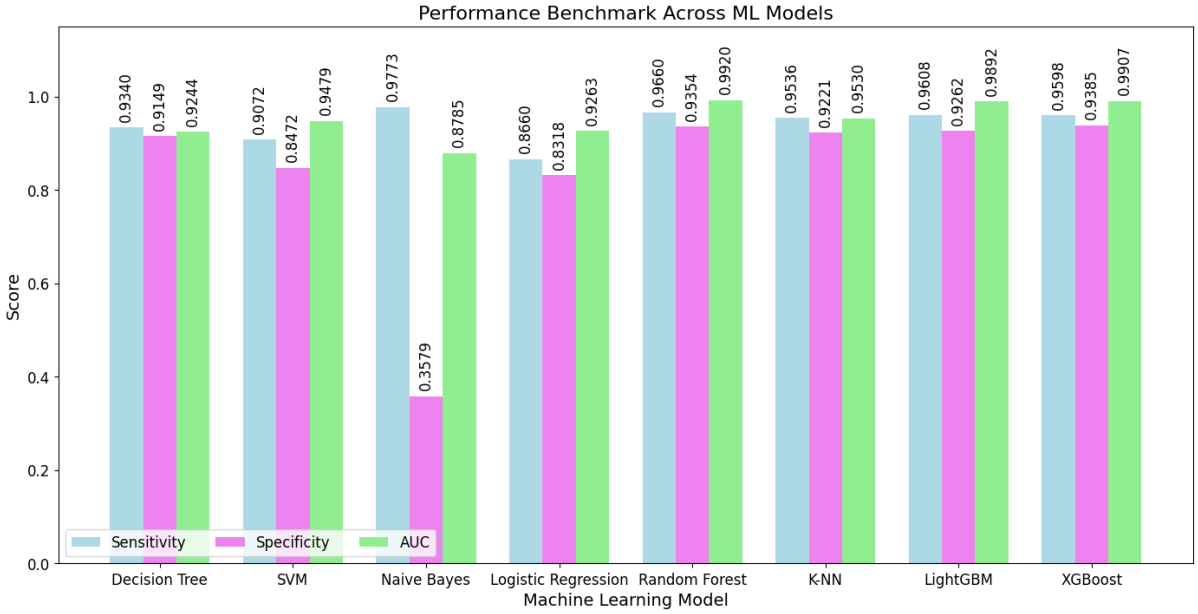


Figure 1: Sensitivity, Specificity, and AUC for each machine learning model when Glucose is log-transformed.

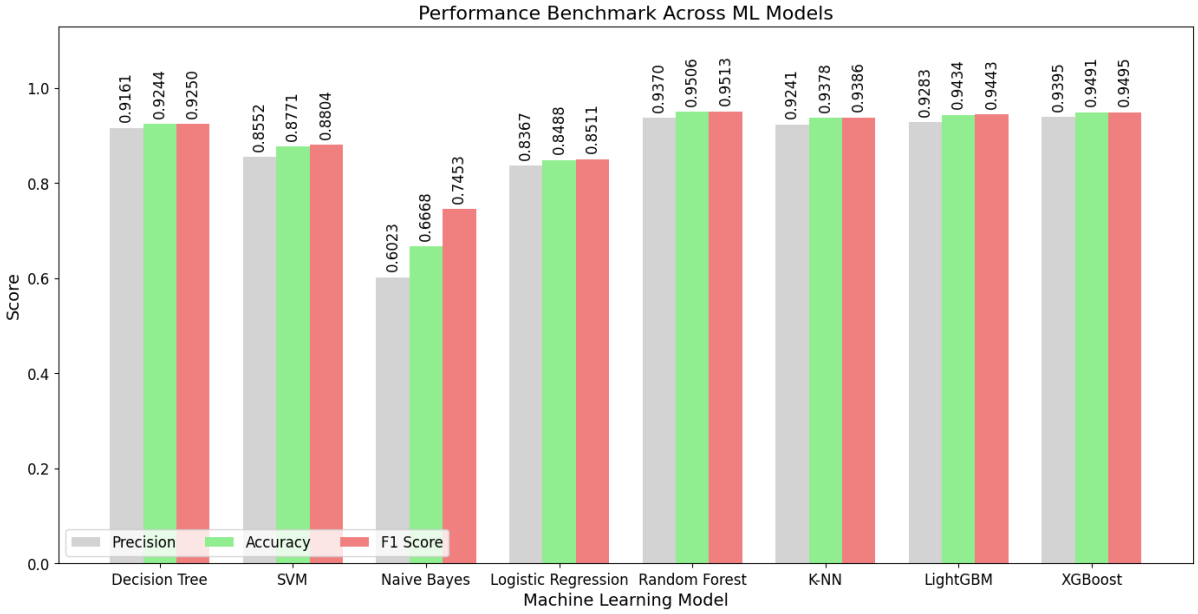


Figure 2: Precision, Accuracy, and F1 score for each machine learning model when Glucose is log-transformed.

Glucose Log Transformation

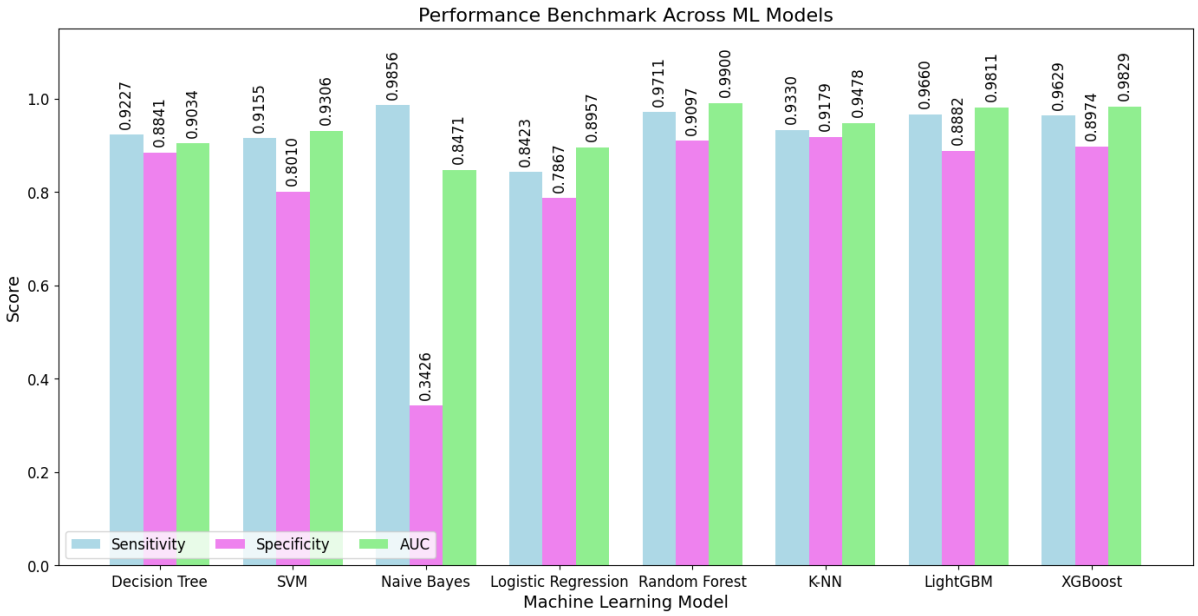


Figure 1: Sensitivity, Specificity, and AUC for each machine learning model when Glucose and BMI are log-transformed.

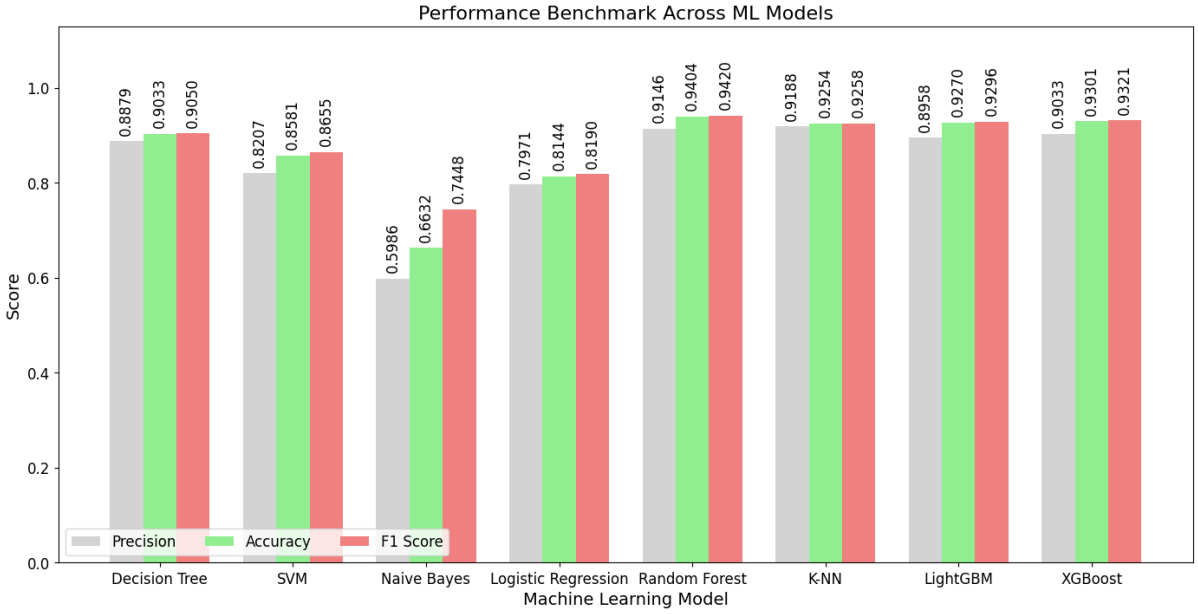


Figure 2: Precision, Accuracy, and F1 score for each machine learning model when Glucose and BMI are log-transformed.

Glucose Log Transformation and BMI Log Transformation

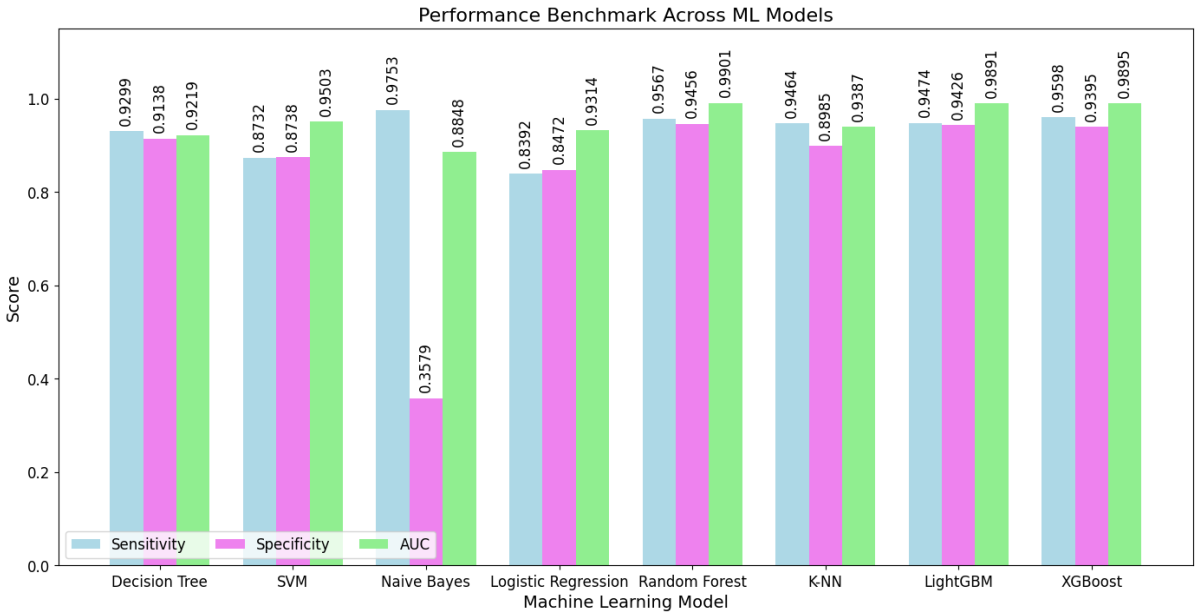


Figure 1: Sensitivity, Specificity, and AUC for each machine learning model when BMI is log-transformed

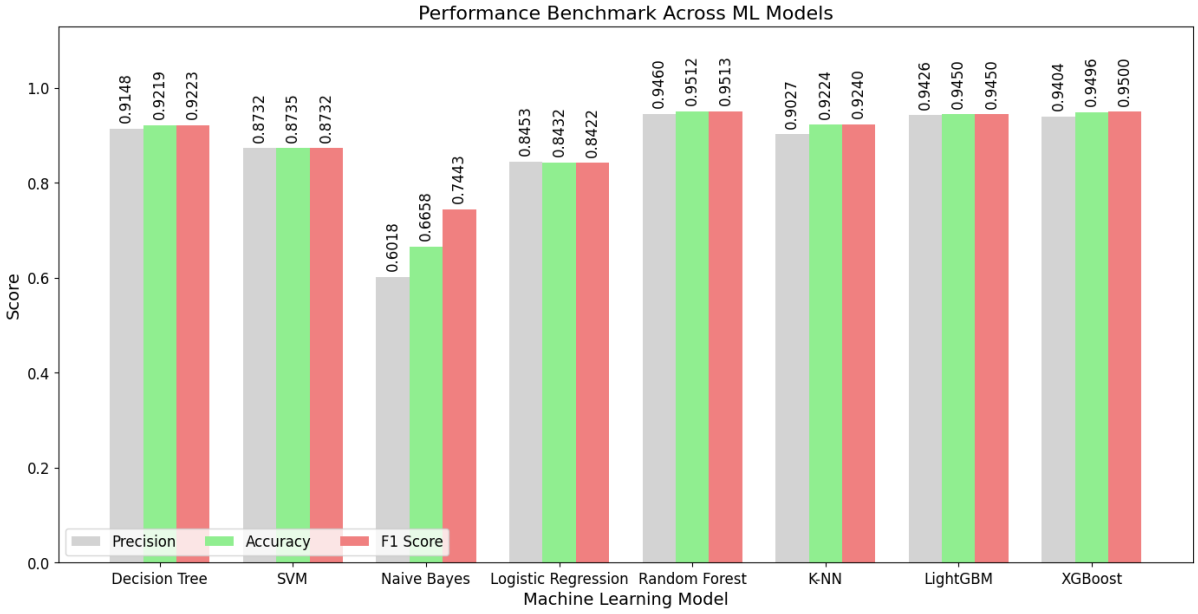


Figure 2: Precision, Accuracy, and F1 score for each machine learning model when BMI is log-transformed

BMI Log Transformation

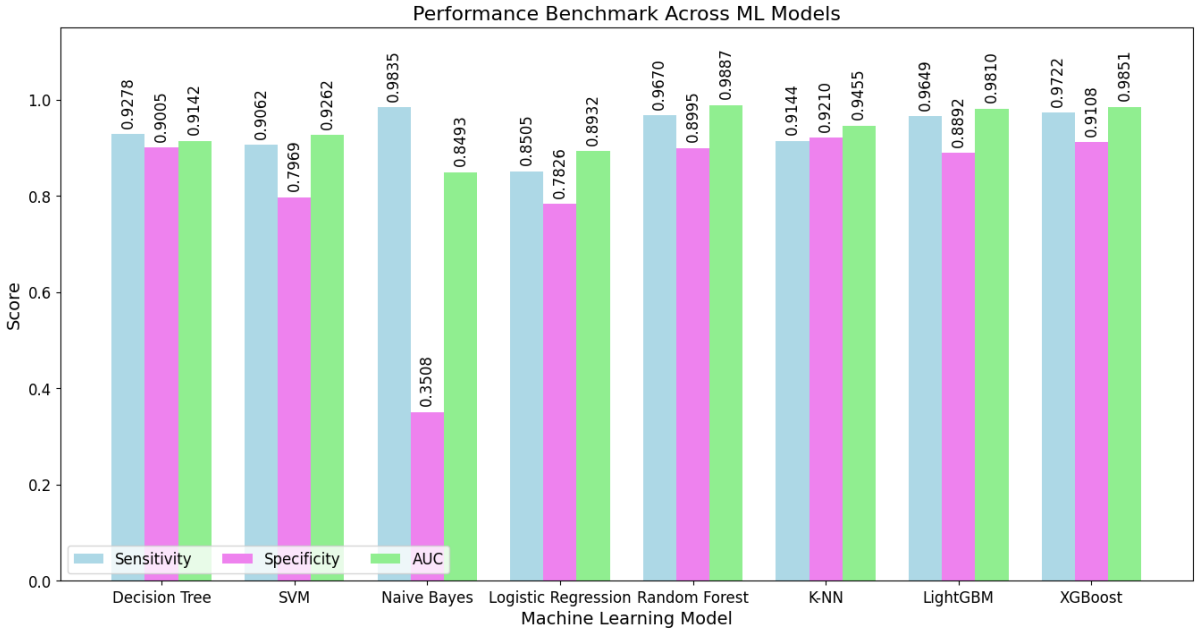


Figure 1: Sensitivity, Specificity, and AUC for each machine learning model when BMI is log-transformed and Glucose is boxcox-transformed.

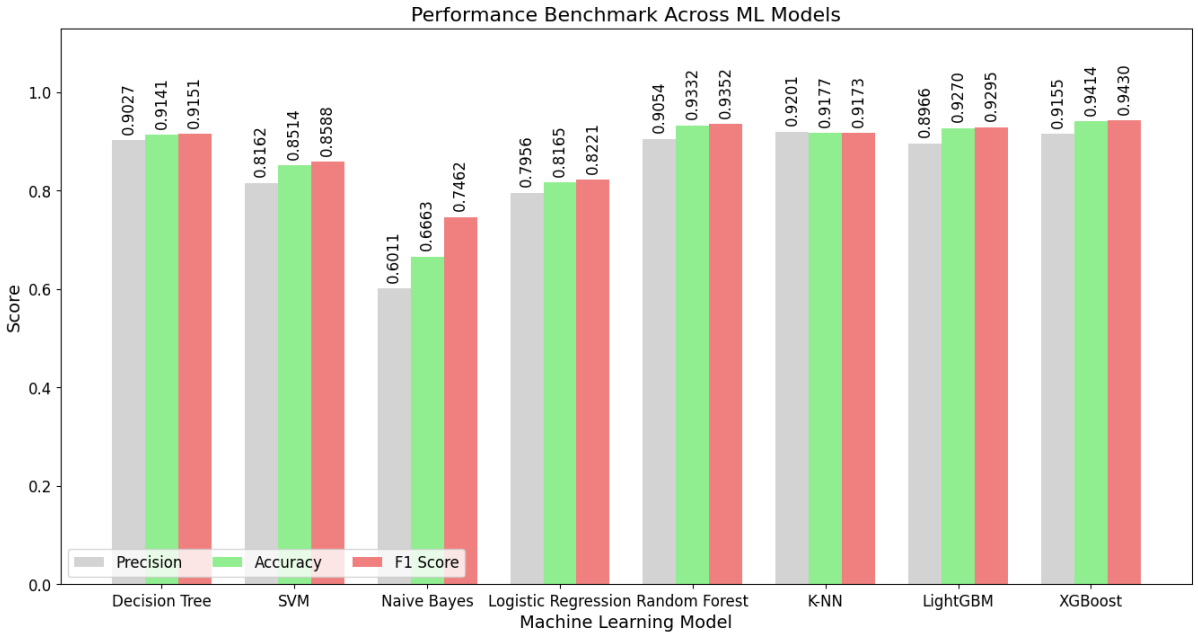


Figure 2: Precision, Accuracy, and F1 score for each machine learning model when BMI is log-transformed and Glucose is boxcox-transformed.

Glucose Box-Cox Transformation and BMI Log Transformation

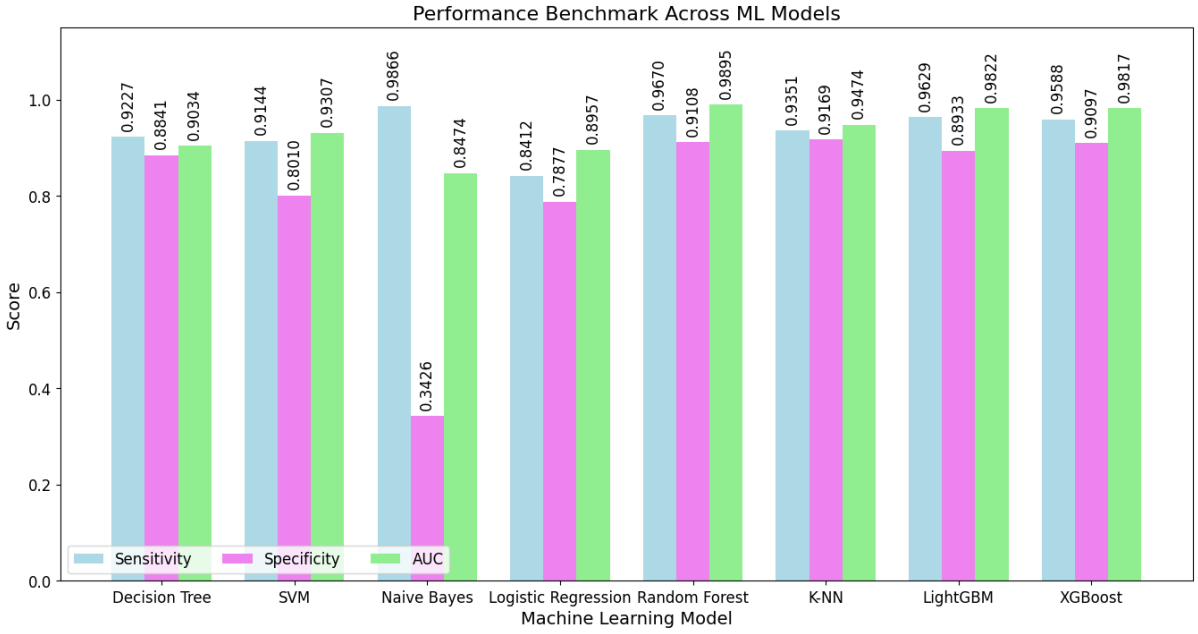


Figure 1: Sensitivity, Specificity, and AUC for each machine learning model when BMI is boxcox-transformed and Glucose is log-transformed.

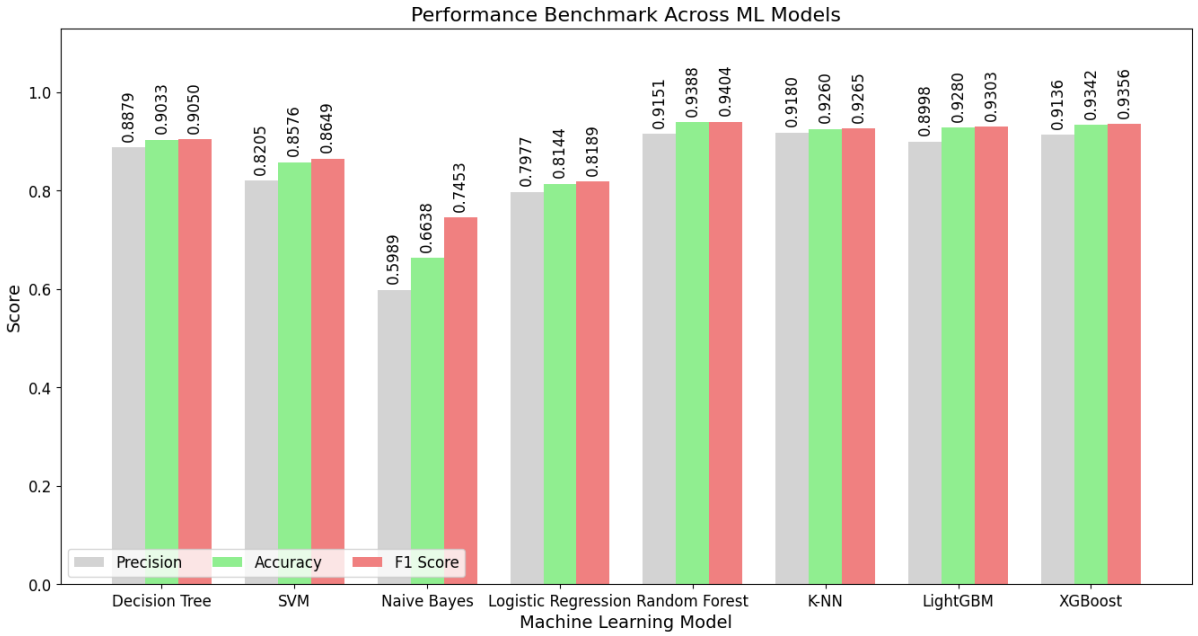


Figure 2: Precision, Accuracy, and F1 score for each machine learning model when BMI is boxcox-transformed and Glucose is log-transformed.

BMI Box-Cox Transformation and Glucose Log Transformation

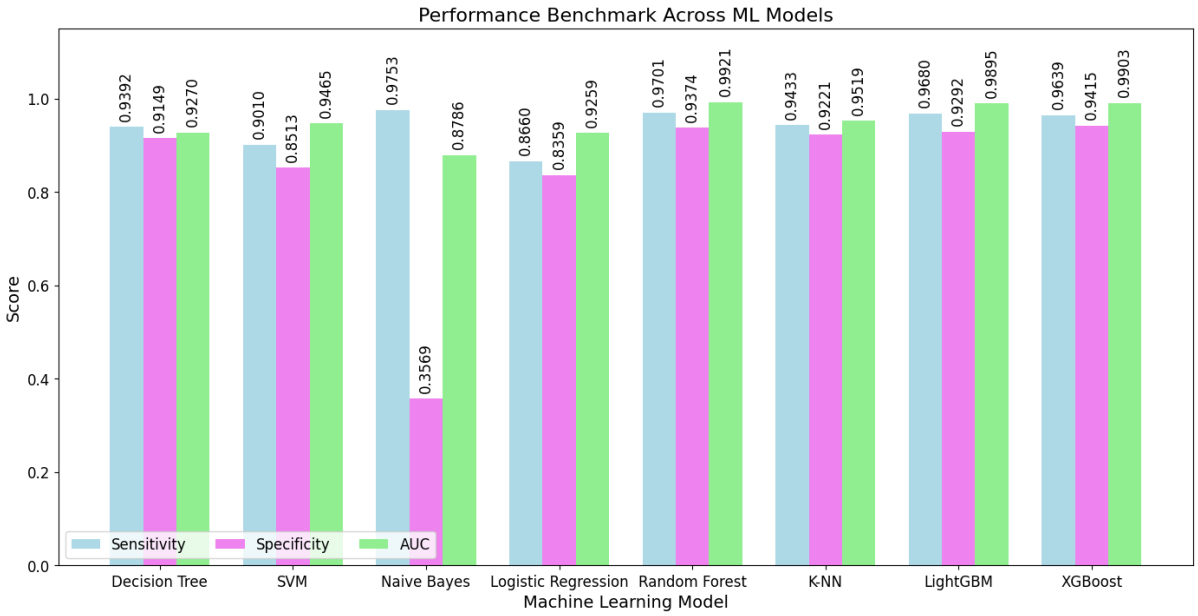


Figure 1: Sensitivity, Specificity, and AUC for each machine learning model when Glucose is boxcox-transformed.

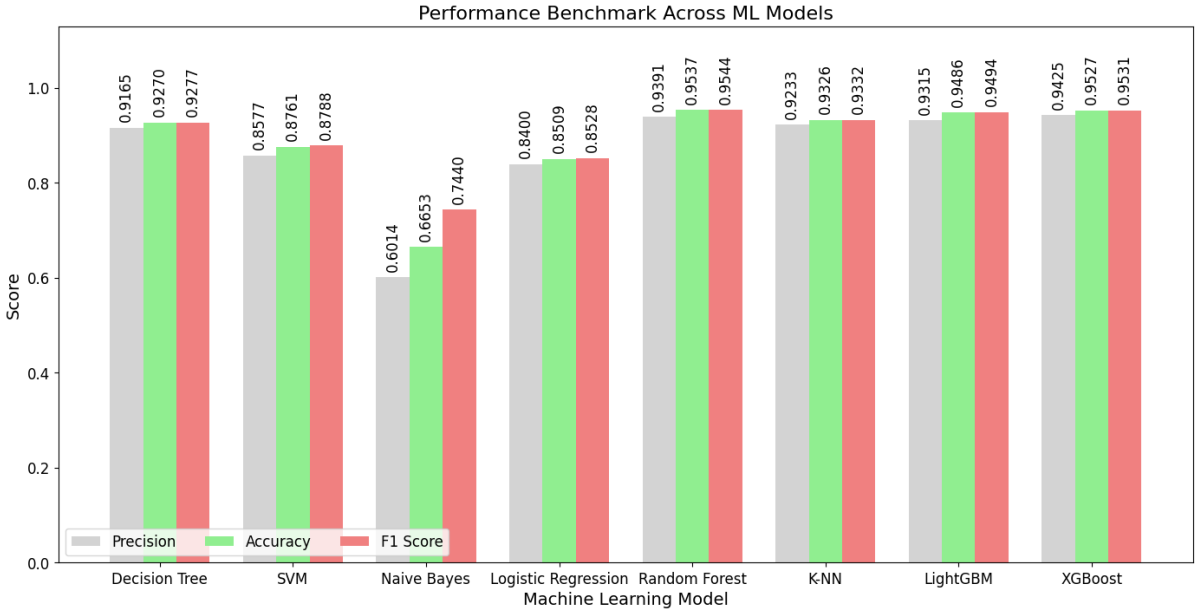


Figure 2: Precision, Accuracy, and F1 score for each machine learning model when Glucose is boxcox-transformed.

Glucose Box-Cox Transformation

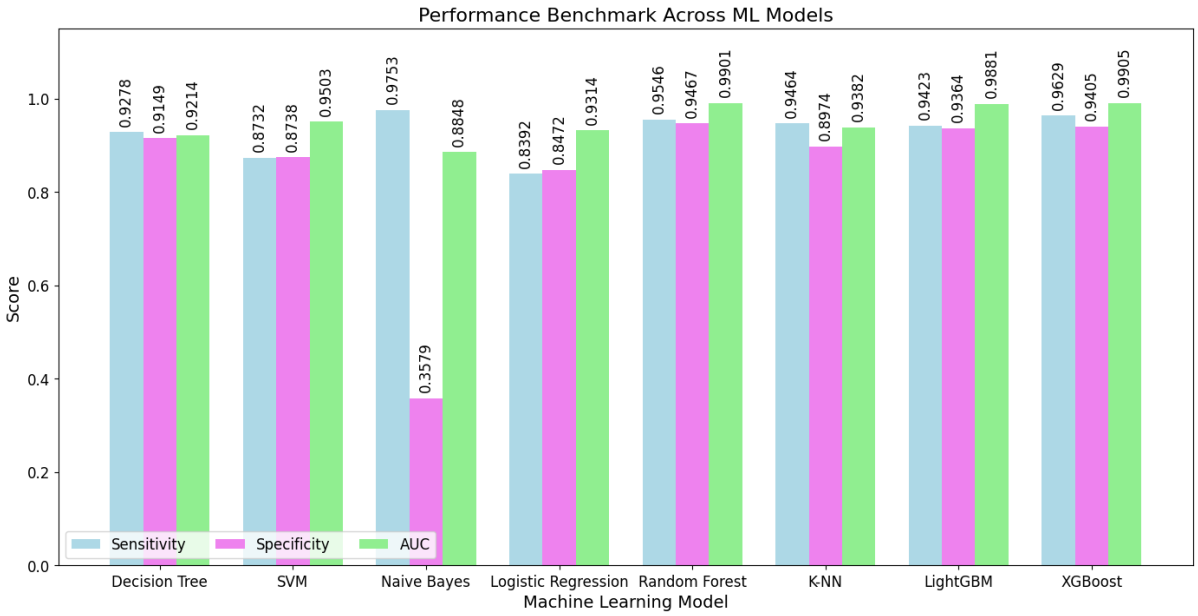


Figure 1: Sensitivity, Specificity, and AUC for each machine learning model when BMI is boxcox-transformed.

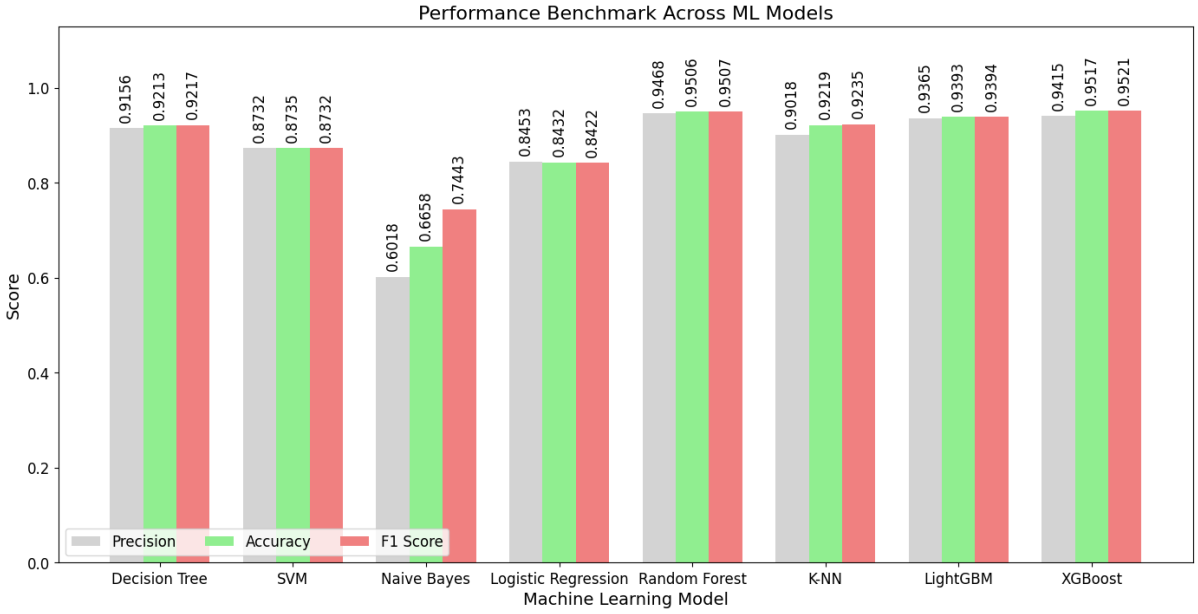


Figure 2: Precision, Accuracy, and F1 score for each machine learning model when BMI is boxcox-transformed.

BMI Box-Cox Transformation

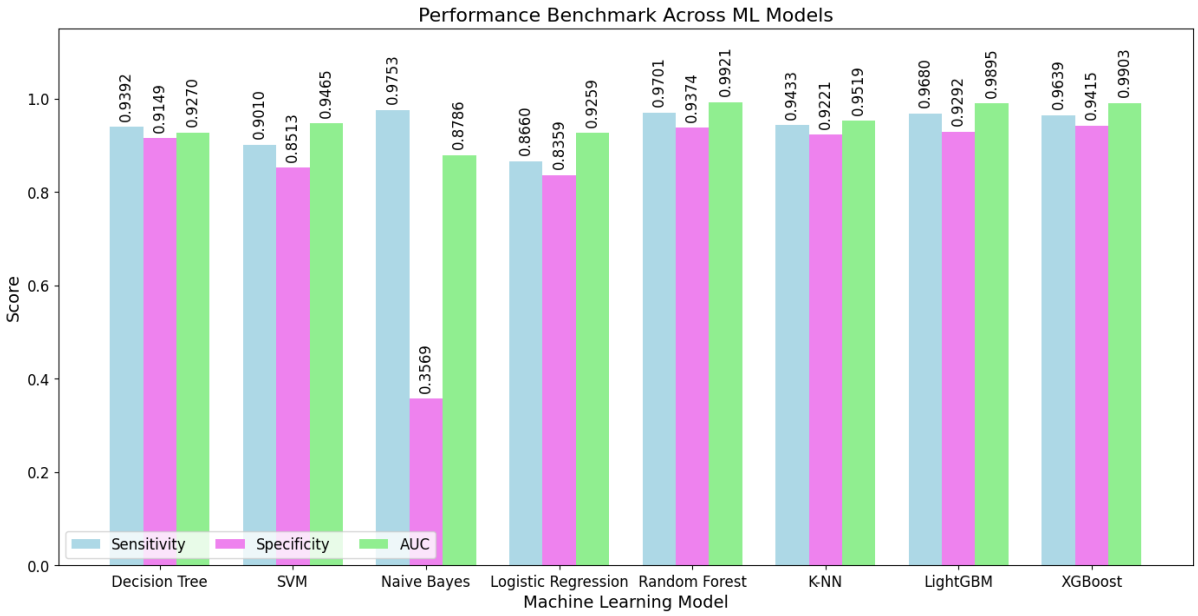


Figure 1: Sensitivity, Specificity, and AUC for each machine learning model when Glucose is boxcox-transformed.

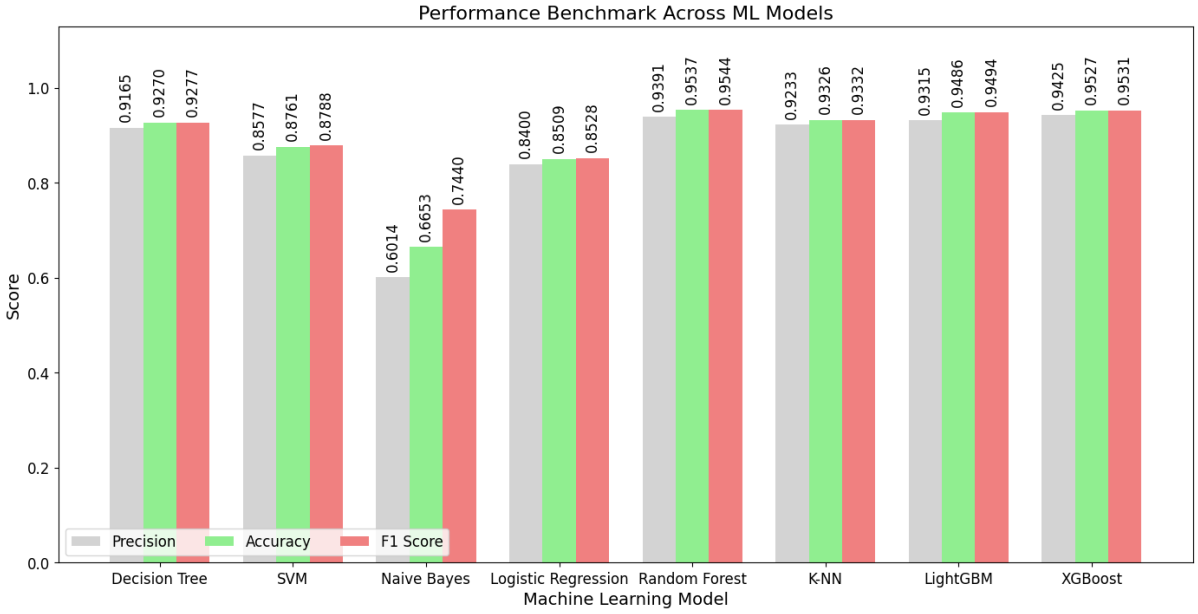


Figure 2: Precision, Accuracy, and F1 score for each machine learning model when Glucose is boxcox-transformed.

Glucose Box-Cox Transformation and BMI Box-Cox Transformation

CONCLUSION:

1. Best Performing Models:

- Random Forest, LightGBM, XGBoost:
    - These models consistently outperform others across different preprocessing scenarios.
    - The Random Forest model, with an accuracy of **95.42%,** showcases its ability to handle complex relationships in the data and mitigate overfitting.
    - LightGBM with accuracy of **95.17%** and XGBoost with accuracy of **95.42%**, gradient boosting techniques, also demonstrate robust performance, benefiting from their ensemble nature and efficient handling of both numerical and categorical features.
    - The AUC score for all the above mentioned models is above 99%

2. **Impact of Data Preprocessing:**

- Loss of Information:
    - The various preprocessing techniques, such as log and Box-Cox transformations, seem to introduce complexity without a significant boost in performance.
    - The absence of consistent improvement across all models and metrics suggests that the original data contains sufficient information for effective prediction.
- Overfitting:
    - The risk of overfitting is mitigated when using the original data, as evidenced by the high accuracy in the "No Preprocessing" scenario.
    - Models might be susceptible to overfitting when transformations are applied without clear necessity or theoretical basis.

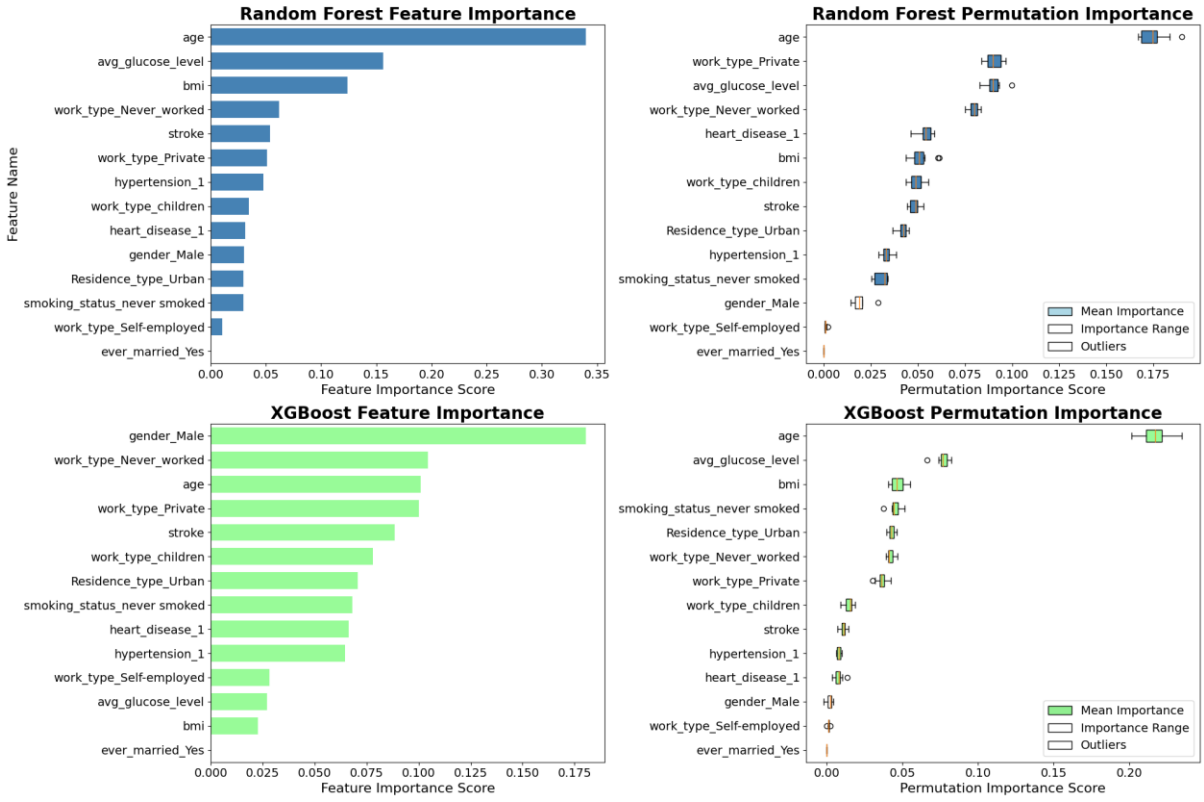
3**. Recommendation:**

- No Preprocessing:
    - The simplicity of using the original data is advantageous, reducing the risk of introducing unnecessary noise or biases.
    - Decision-makers should consider adopting the "No Preprocessing" scenario as the default setting for the Brain Stroke Prediction Model.
- Random Forest, LightGBM, XGBoost:
    - These models demonstrate the highest potential for real-world application due to their consistently strong performance.
    - The ensemble nature of Random Forest and gradient boosting techniques in LightGBM and XGBoost contributes to their stability and predictive power.

4. **Future Considerations:**

- Feature Importance Analysis:
    - Investigate feature importance to identify the key factors influencing stroke predictions. This can enhance interpretability and guide targeted interventions.
- Hyperparameter Tuning:
    - Explore model hyperparameter tuning for Random Forest, LightGBM, and XGBoost to further optimize their performance.
    - Fine-tuning can unlock additional potential in these models, ensuring they are finely tuned to the characteristics of the dataset.

Feature And Permutation Importance Comparison of 2 Best Resulting Models



REFERENCE AND CHALLENGES WE FACED:

Challenge: Choosing the appropriate random state in hyperparameters.

Solution: We opted for **random_state=42** in hyperparameters to ensure consistent train and test sets across different executions. This choice provides reproducibility in the shuffling process, enhancing the reliability of our model evaluation. By fixing the random state, we

maintain control over the randomness introduced during the train-test split, enabling us to compare model performance consistently.

Reference: Why Do We Set a Random State in Machine Learning Models?

Challenge: Handling Imbalanced Class in the dataset.

Solution: The challenge involved addressing the imbalanced class issue in the dataset, where the instances of the positive class (stroke) were significantly lower than the negative class. To mitigate this imbalance and prevent the model from being biased towards the majority class, we chose to oversample using the Synthetic Minority Oversampling Technique (SMOTE). By generating synthetic samples from the minority class, we ensured a more balanced representation, allowing the model to learn from both classes without losing essential information. This approach is preferred over undersampling, as it retains a more comprehensive set of data for training.

Resampling Details:

- Shape of the original dataset: (5110, 14)
- Number of instances with stroke in the original dataset: 249
- Shape of the resampled dataset: (9722, 14)
- Number of instances with stroke in the resampled dataset: 4861

Reference: Resampling to Properly Handle Imbalanced Datasets in Machine Learning

Challenge: Choosing between SMOTE and random oversampling for handling imbalanced data.

Solution: We opted for SMOTE (Synthetic Minority Over-sampling Technique) over random oversampling due to its ability to generate synthetic examples for the minority class. Unlike random oversampling, which duplicates existing instances, SMOTE focuses on creating artificial examples based on feature space similarities. This method addresses the challenge of imbalanced data more effectively by introducing diversity in the synthetic samples, contributing to a more robust and representative training dataset.

Reference: SMOTE: Synthetic Minority Over-sampling Technique

Challenge: Choosing between hot encoding and label/integer encoding for categorical variables.

Solution: We opted for one-hot encoding over label/integer encoding to avoid introducing bias into the data. Integer encoding may imply a natural ordering between categories, potentially leading to poor model performance or unexpected outcomes. One-hot encoding, on the other hand, eliminates this issue by representing each category as a binary vector, preventing the model from assuming any ordinal relationship between them. This enhances the model's ability to learn and generalize effectively.

One-hot encoding also facilitates easier learning for the model as it can distinctly recognize unique binary representations for each input value, contributing to improved performance. Additionally, it proves to be more memory and computationally efficient, especially with

large datasets or complex models, due to the shorter and sparser nature of binary vectors compared to integer encodings.

Reference: Machine Learning: One-Hot Encoding vs Integer Encoding

Challenge: Calculating sensitivity and specificity for model evaluation.

Solution: To calculate sensitivity and specificity, we used the formulas:

Sensitivity=True Positives / (False Negatives + True Positives)

Sensitivity=True Positives / (False Negatives + True Positives)

In Python, this was implemented as follows:

sensitivity = cm[0, 0] / (cm[0, 0] + cm[0, 1])

specificity = cm[1, 1] / (cm[1, 0] + cm[1, 1])

This approach provides insights into the model's ability to correctly identify positive and negative instances, respectively. Sensitivity measures the proportion of actual positive instances correctly identified, while specificity measures the proportion of actual negative instances correctly identified. These metrics are crucial for evaluating the performance of a classification model.

Reference: Calculating Sensitivity and Specificity in Python

Challenge: Correcting the skewness of data in the project.

Solution with Reasoning: We addressed the skewness of the data using different methods based on the nature and extent of skewness:

1. Square Root Transformation: Applied when the data is moderately skewed. The square root ($x^{1/2}$) moderately affects the distribution shape and is effective in reducing right-skewed data. This method can be applied to zero values and is commonly used for counted data.
2. Log Transformation: Employed for reducing right skewness, the logarithmic transformation ($\log(x)$) has a significant impact on the distribution shape. However, it cannot be applied to zero or negative values.
3. Box-Cox Transformation: This method is used to transform non-normal data into a normal shape. It involves identifying a suitable exponent (Lambda = λ) to transform skewed data.

Each transformation method was chosen based on its suitability for the specific skewness characteristics of the data, ensuring an effective correction while considering the nature of the dataset.

Reference: Transform Skewed Data using Square Root, Log, Box-Cox Methods in Python