Predictive Analytics of Stroke

Milan Mitrovic | s4663796

Contents

PROJECT SUMMARY/ABSTRACT

Stroke is a medical emergency that requires immediate attention as it can lead to severe disability or even death if not treated promptly. Identifying individuals at risk of stroke is crucial for timely intervention and prevention. This project is all about stroke prediction and determining the best suited algorithms for doing such a task. We utilized multiple machine learning algorithms to predict the likelihood of stroke based on given and unseen data. The goal was to determine the best algorithm for accurate stroke prediction, which could have significant implications for patient care and outcomes. Machine learning algorithms play a crucial role in stroke prediction by leveraging datasets to identify patterns and risk factors associated with stroke occurrence. By analyzing various patient characteristics and health indicators, machine learning algorithms can provide personalized risk assessments, enabling healthcare professionals to intervene early and implement preventive measures. This proactive approach can significantly reduce the likelihood of stroke and its associated complications.

INTRODUCTION

Supervised machine learning algorithms can analyse patient data to detect risk factors and forecast the probability of stroke by detecting patterns. This enables earlier and more precise diagnosis of strokes, leading to enhanced patient outcomes. Such insights aid in treatment planning and stroke prevention efforts. Machine learning holds promise in substantially enhancing the diagnosis, management, and overall prognosis for stroke patients. Supervised learning is a type of machine learning where the training data is labelled – the system learns with a teacher.

We'll evaluate five distinct supervised machine learning algorithms to determine the most effective one for diagnosing our patients:

1. **Logistic Regression** - Logistic regression, a supervised machine learning algorithm, predicts the likelihood of an instance belonging to a specific class – it does, or it doesn't: e.g., True/False, Success/Failure, Living/Deceased, etc. It analyzes independent variables to determine a binary outcome, where the results are categorized into one of two groups. These independent variables can be either categorical or numeric, while the dependent variable is always categorical. Logistic regression computes the probability of the dependent variable Y given the independent variable X. In this context, it predicts the probability of having a stroke or not (stroke – 1: stroke, 0: no stroke) where 1 indicates stroke and 0 indicates no stroke.

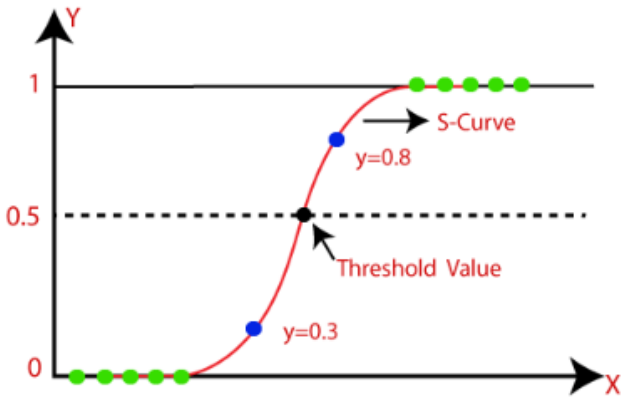


Figure 1. From A Guide to any Classification Problem Durgance Guar, 2022

2. **Support Vector Machine (SVM)** – A Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. In this context, we are using SVM for classification: The primary objective of SVMs in classification is to identify and learn the optimal decision boundary (hyperplane), that best separates different classes within the dataset – in our case the results of stroke. SVM achieves this by maximizing the margin between the decision boundary and the closest data points from each class, known as support vectors. It is also effective in handling both linearly separable and non-linearly separable datasets through the use of kernel functions.

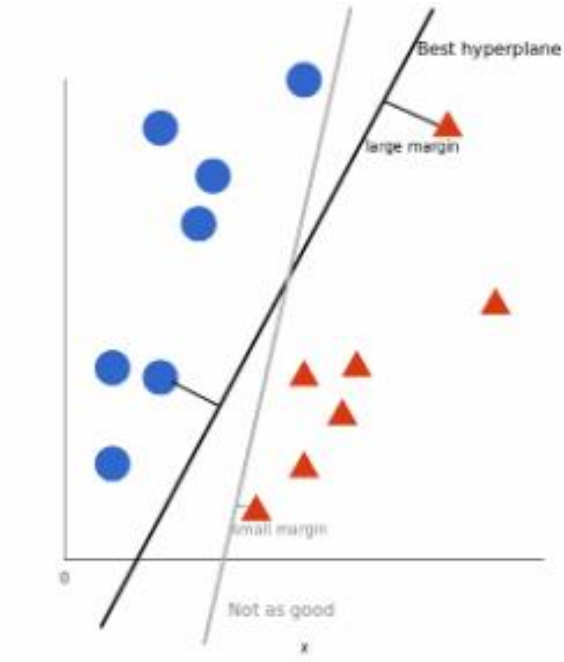


Figure 2. From A Guide to any Classification Problem Durgance Guar, 2022

3. **K-Nearest Neighbor (KNN)** – The K-Nearest Neighbor (KNN) is a supervised machine learning algorithm that relies on proximity to make classifications or predictions about the grouping of a particular data point. It achieves this by identifying the k-nearest neighbors among future examples. In classification tasks, KNN determines the category of a data point based on the category of its closest neighbor. For instance, if K equals 1, the data point would be assigned to the class of its closest neighbor. The value of K is determined by a majority vote among its neighbors. Typically, the optimal value of K is usually found to be the square root of the total number of samples (N). In essence, KNN operates on the assumption that similar entities are located close to each other -- calculating the distance between data points with distance metrics such as Euclidean, Manhattan, and Minkowski distances.

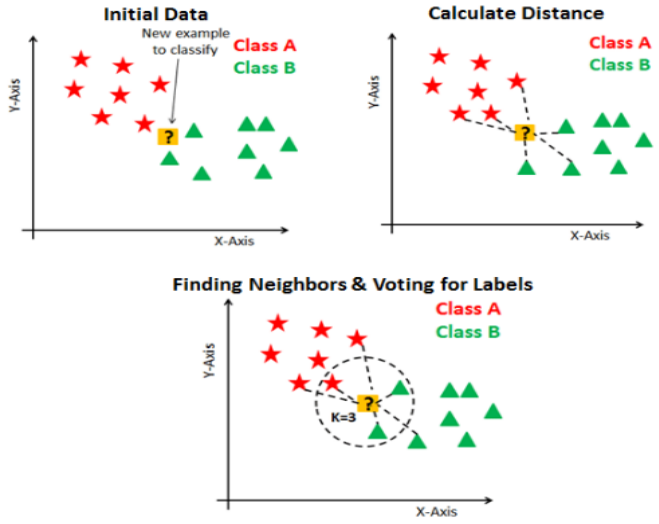


Figure 3. From A Guide to any Classification Problem Durgance Guar, 2022

4. **Decision Tree** - A decision tree is a supervised machine learning algorithm, which is utilized for both classification and regression tasks. It constructs a tree-like structure to partition the feature space into regions, each corresponding to a specific class label. At each node of the tree, the algorithm selects the feature that best separates the data into purest possible subsets based on some criterion, such as information gain. This process continues until a stopping criterion is met; reaching a maximum depth, minimum number of samples per leaf, or when further splits no longer improve the purity of the subsets. During classification, an instance's feature values are traversed down the tree from the root node to a leaf node, following the decision rules at each internal node. Once a leaf node is reached, the instance is assigned the class label associated with that leaf node.

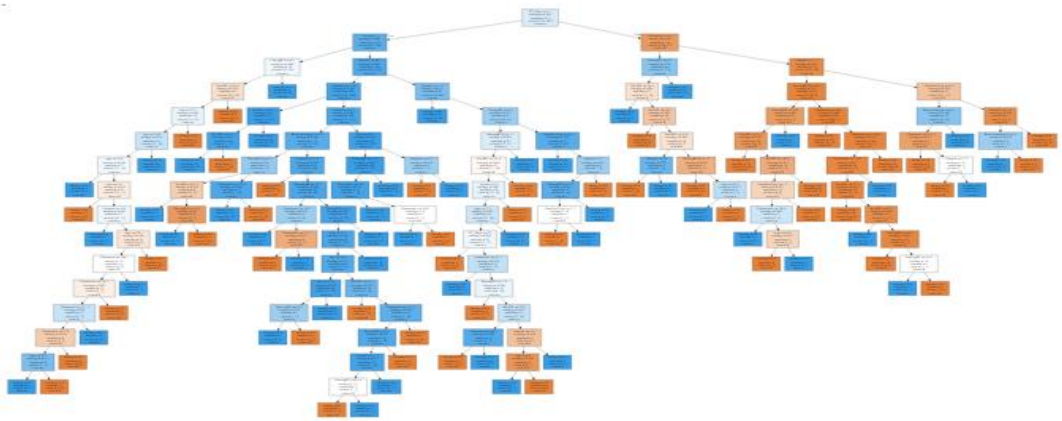


Figure 4. From A Guide to any Classification Problem Durgance Guar, 2022

5. **Random Forest Classifier** – A random forest classifier is a supervised machine learning algorithm used for classification tasks. Like decision trees it constructs a tree-like structure, which means it moves like a flow chart, separating each data point into more and more similar groups until it reaches a defined limit. The Random Forest Classifier takes this idea and multiplies it, running many individual trees at once and makes a prediction based on the most popular result.

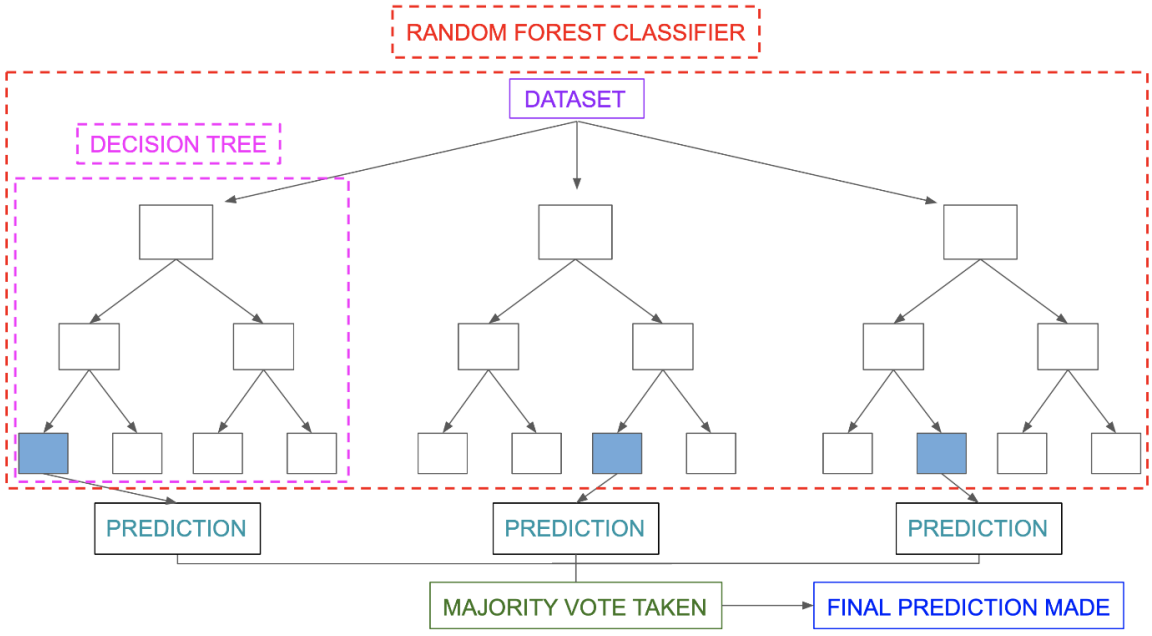


Figure 5. From A Guide to any Classification Problem Durgance Guar, 2022

We will measure the performance of our models by using these evaluation metrics:

- **Accuracy** – Accuracy is calculated as the proportion of accurate predictions made by the model relative to the total number of input samples, expressed as a percentage. This is calculated by the following formula:

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

  Basically, the total correct answers are divided by the total number of entries. It's an efficient way to measure the predictive success of the model.

- **Precision** – Precision evaluates a model's effectiveness by indicating the proportion of positive predictions made by the model that are correct. It is computed as the ratio of true positive predictions to the sum of true positive and false positive predictions.

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

- **Recall** – Recall represents the proportion of positive predictions made by the model relative to all relevant samples that should have been identified as positive. It measures the accuracy of identifying actual positive instances. This metric is computed by dividing the number of true positive predictions by the sum of true positive and false negative predictions.

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

- **F1 Score** – F1 Score is the harmonic mean between precision and recall. It reflects how precise the classifier is (how many instances it classifies correctly), as well as how robust it is (does not miss any significant number of instances). We use harmonic mean instead of regular mean because it punishes values that are further apart.

  $$F1 = \frac{2xPrecisionxRecall}{Precision+Recall}$$

- **Confusion Matrix** – A confusion matrix is an N X N matrix, where N represents the number of classes being predicted. It's a tool for summarizing the performance of a classification algorithm; it gives us a clear picture of the specific classification models performance and the types of errors produced by the model. The summary provides correct and incorrect predictions broken down by each category,

represented by a matrix. Almost all performance metrics are based on the confusion matrix and the numbers inside it.

Four types of outcomes are possible while evaluating a models performance:

1. **True Positives (TP)** - The number of observations that are actually positive (belonging to the stroke class) and are correctly classified as positive by the model.
2. **True Negatives (TN)** – The number of observations that are actually negative (not belonging to the stroke class) and are correctly classified as negative by the model.
3. **False Positives (FP)** - The number of observations that are actually negative but are incorrectly classified as positive by the model (incorrectly predicted as belonging to the stroke class). Also known as a Type I error.
4. **False Negatives (FN)** – The number of observations that are actually positive but are incorrectly classified as negative by the model (incorrectly predicted of not belonging to the stroke class) This error, known as Type II error, is particularly critical.

TP and FN are crucial as they directly affect whether a patient is correctly or incorrectly notified of having a stroke. TN and FP are of lesser importance since they relate to cases where the patient does not have a stroke – no urgent need to inform them of this absence.

DATASET DESCRIPTION

Stroke is the 2nd leading cause of death worldwide, accounting for approximately 11% of total global deaths – according to WHO (World Health Organization). This dataset contains 12 features which can be analyzed and used to predict strokes by utilizing machine learning algorithms. The features are: (id, gender, age, hypertension, heart disease, ever married, work type, residence type, average glucose level, BMI, smoking status, stroke). The feature stroke indicates whether a patient had a stroke or not; this is the label we are going to determine and predict – a form of classification. Supervised machine learning algorithms will be employed to fulfill this task: by training the algorithms with data, instructing them how to differentiate the positive and negative value for stroke. Once trained, the algorithms can be applied to new data that doesn't include a stroke attribute, allowing for the prediction of whether a patient has a stroke or not.

```
#Packages
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import os
from pandas import get_dummies
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split, KFold, cross_val_score, GridSearchCV, cross_validate, StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score, classification_report, confusion_matrix, Cor
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from imblearn.combine import SMOTEENN
from itertools import combinations
```

Importing the necessary packages.

```
#Importing dataset
df= pd.read_csv('C:/Users/Milan/healthcare-dataset-stroke-data.csv')
```

Importing the dataset.

METHODS AND ALGORITHMS

Data Cleaning

Before starting Exploratory Data Analysis (EDA) we need to prepare and clean the data, as dirty data can lead to misleading results and inaccurate conclusions.

```
#Dimensions of the dataset
df.shape
```

```
(5110, 12)
```

```
#Removes duplicates
df.drop_duplicates()
```

	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
...	...	...	...	...	...	...	...	...	...	...	...	...
5105	18234	Female	80.0	1	0	Yes	Private	Urban	83.75	NaN	never smoked	0
5106	44873	Female	81.0	0	0	Yes	Self-employed	Urban	125.20	40.0	never smoked	0
5107	19723	Female	35.0	0	0	Yes	Self-employed	Rural	82.99	30.6	never smoked	0
5108	37544	Male	51.0	0	0	Yes	Private	Rural	166.29	25.6	formerly smoked	0
5109	44679	Female	44.0	0	0	Yes	Govt_job	Urban	85.28	26.2	Unknown	0

5110 rows × 12 columns

Checking the dimensions of the dataset (5110 rows, 12 columns), as well as removing duplicates in case they were present (there weren't any).

```
#Returns first 5 rows of the dataframe
df.head()
```

	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

```
#Concise summary of the dataframe
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5110 entries, 0 to 5109
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   id                 5110 non-null   int64  
 1   gender             5110 non-null   object 
 2   age                5110 non-null   float64
 3   hypertension       5110 non-null   int64  
 4   heart_disease      5110 non-null   int64  
 5   ever_married       5110 non-null   object 
 6   work_type          5110 non-null   object 
 7   Residence_type     5110 non-null   object 
 8   avg_glucose_level  5110 non-null   float64
 9   bmi                4909 non-null   float64
 10  smoking_status     5110 non-null   object 
 11  stroke             5110 non-null   int64  
dtypes: float64(3), int64(4), object(5)
memory usage: 479.2+ KB
```

```
#Summary of numerical features
df.describe()
```

	id	age	hypertension	heart_disease	avg_glucose_level	bmi	stroke
count	5110.000000	5110.000000	5110.000000	5110.000000	5110.000000	4909.000000	5110.000000
mean	36517.829354	43.226614	0.097456	0.054012	106.147677	28.893237	0.048728
std	21161.721625	22.612647	0.296607	0.226063	45.283560	7.854067	0.215320
min	67.000000	0.080000	0.000000	0.000000	55.120000	10.300000	0.000000
25%	17741.250000	25.000000	0.000000	0.000000	77.245000	23.500000	0.000000
50%	36932.000000	45.000000	0.000000	0.000000	91.885000	28.100000	0.000000
75%	54682.000000	61.000000	0.000000	0.000000	114.090000	33.100000	0.000000
max	72940.000000	82.000000	1.000000	1.000000	271.740000	97.600000	1.000000

Running a few commands to summarize and view the data. Besides the good amount of data entries, consistent data types and mostly non-nulls — there are issues. Firstly, the column ‘id’ is not needed: the feature is irrelevant as it does not provide any valuable information for analysis nor is it good for the modelling process; in fact, removing it will reduce noise and improve model performance. Another issue is the ‘bmi’ column, it appears to contain null values and outliers which can hold biases and skew model results. The last issue is in the ‘gender’ column having the value ‘Other’: irrelevant value which is not necessary and won’t be beneficial to the analysis/modelling process. We will run some code to determine how many null values ‘bmi’ contains and the number of ‘Other’ values ‘gender’ contains.

```
#Displays all rows with 'Other' value in gender column
df[df["gender"] == 'Other']
```

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
3116	56156	Other	26.0	0	0	No	Private	Rural	143.33	22.4	formerly smoked	0

```
#Returning count of values in gender column
df.gender.value_counts()
```

```
Female    2994
Male      2115
Other        1
```

```
#Identifies missing values
df.isnull().sum()
```

```
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
```

201 null values in 'bmi' and 1 'Other' value.

I deal with all the issues described above accordingly:

- We deal with the 'id' column by removing it from the data frame using the drop() method: it works by removing the specified column/row – in our case the 'id' column. We do this because of its irrelevancy and removing it will improve overall analyzation and modelling performance.

```
#Dropping id column
df.drop('id',axis=1,inplace=True)
df.head()
```

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

- We deal with the 1 'Other' row by removing it from the data frame by using the drop() method again; We decided to remove this specific row as it was irrelevant in aid of the analyzation/modelling process and it is only a singular row which equates to less than 0.1% of the data frame, which is insignificant – reducing noise.

```
#Drop 'Other' row
df.drop(index=df[df['gender']=='Other'].index,inplace=True)
df.gender.value_counts()
```

```
Female    2994
Male      2115
```

- We deal with the 201 null values in 'bmi' by replacing the null values with the mean values from the 'bmi' column, this is done by the fillna() method which works by replacing null values with specified values – in our case the mean values

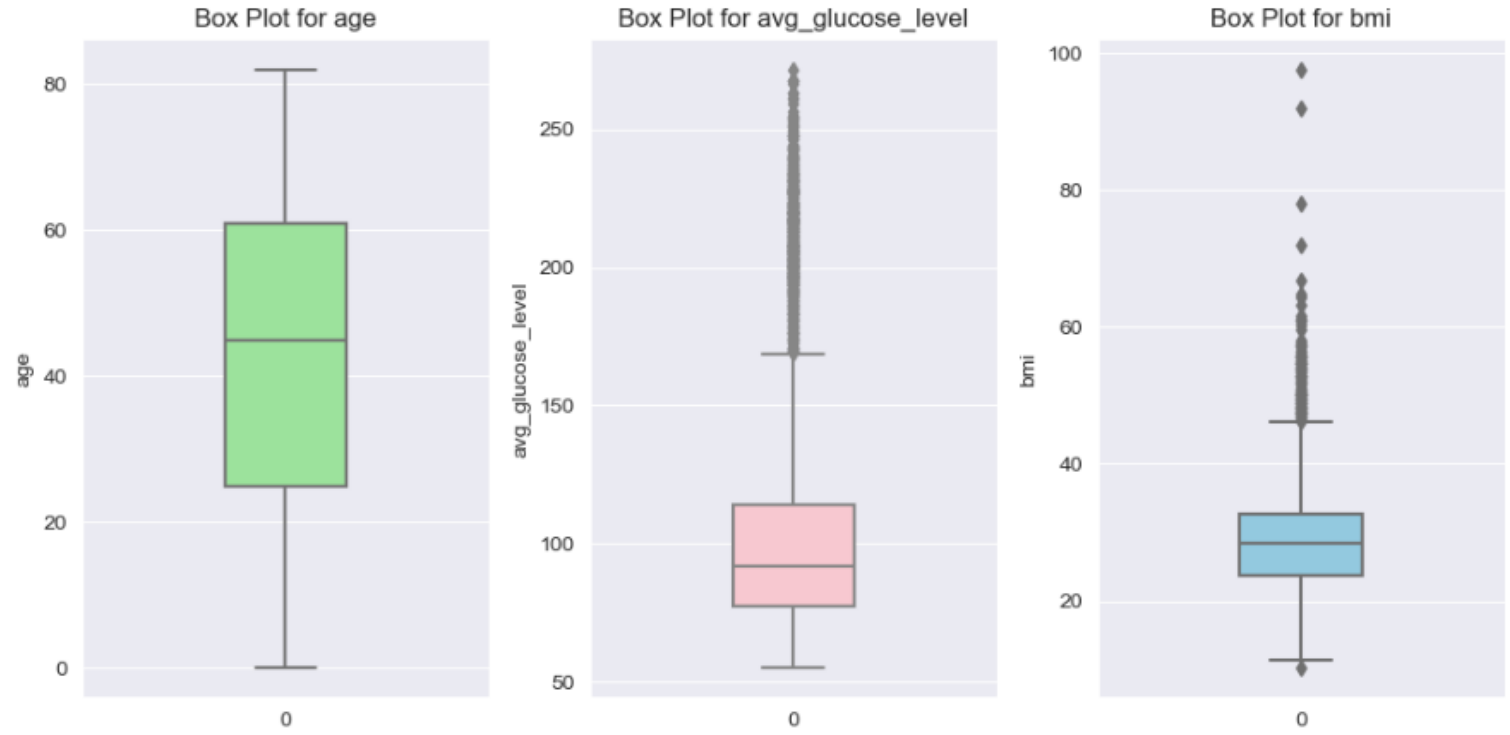
of ‘bmi’. We opt to substitute the null values with the mean of ‘bmi’ to preserve 4% of the data frame, which holds significance. This approach eliminates the null values, which could otherwise distort the data — negatively affecting the models performance.

```
#Replace bmi null values to bmi mean values
df['bmi'].fillna(((df['bmi'].mean())),inplace=True)
df.isnull().sum()
```

```
gender              0
age                 0
hypertension        0
heart_disease       0
ever_married        0
work_type           0
Residence_type      0
avg_glucose_level   0
bmi                 0
smoking_status      0
stroke              0
dtype: int64
```

Now that we've addressed the null values in the 'bmi' column, it's time to investigate the outliers in all numerical columns, including 'bmi'. This will be achieved by data visualization techniques — specifically boxplots. Boxplots displays several key descriptive statistics, including the median, quartiles, and potential outliers, in a concise and easy-to-understand manner. However, we will only be focusing on outliers here as this is a part of the data cleaning process.

```python
#Box Plots for numerical features to visualize if there are outliers

#Select numerical features
numerical_columns = df.select_dtypes(exclude=['object', 'int']).columns

#Create a figure with 3 subplots
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(10,5))
axes = axes.flatten()

#Define Colors
colors = ['lightgreen', 'pink', 'skyblue']

#Iterate over each numerical column and plot a Box Plot
for i, column in enumerate(numerical_columns):
    sns.boxplot(df[column], ax=axes[i], color=colors[i], width=0.3)
    axes[i].set_title(f"Box Plot for {column}")
    axes[i].set_ylabel(f"{column}")

plt.tight_layout()
plt.show()
```



Clearly, apart from age there are outliers present. The outliers in glucose appear to be valid extreme values, as even the highest outlier value of 271 mg/dL is humanly possible without being fatal. It can be concluded that the outliers in glucose are valid values that aren't errors. BMI's outliers on the other hand are biologically impossible: the highest outlier value being 97.6 — not physiologically possible for a living human. In contrast, we can conclude that the outliers in BMI are not valid extreme values and are actually errors that need to be rectified as they are junk values which would skew and distort the data, adversely affecting the quality of the dataset and the performance of the models.

There are a few methods to deal with outliers, however I will be using the percentile method to deal with BMI's outliers. The percentile method is an outlier treatment approach that involves identifying and limiting extreme values by setting a predefined percentage threshold. It works by computing threshold values using percentiles and removing any data points surpassing these thresholds. In our case we will be setting the threshold at 0.001 and 0.999 percentiles. Essentially, we will be trimming a certain threshold of extreme values (anomalies) within BMI.

Now, we will run some code to determine the length of these percentile thresholds and then remove the outliers.

```
#The percentile method with 0.001 and 0.999 thresholds
feature = 'bmi'

min_thre = df[feature].quantile(0.001)
max_thre = df[feature].quantile(0.999)
min_ex = len(df[df['bmi']<min_thre])
max_ex = len(df[df['bmi']>max_thre])

print(f'len of min threshold excluded {min_ex}\nlen of max threshold excluded {max_ex}')
```

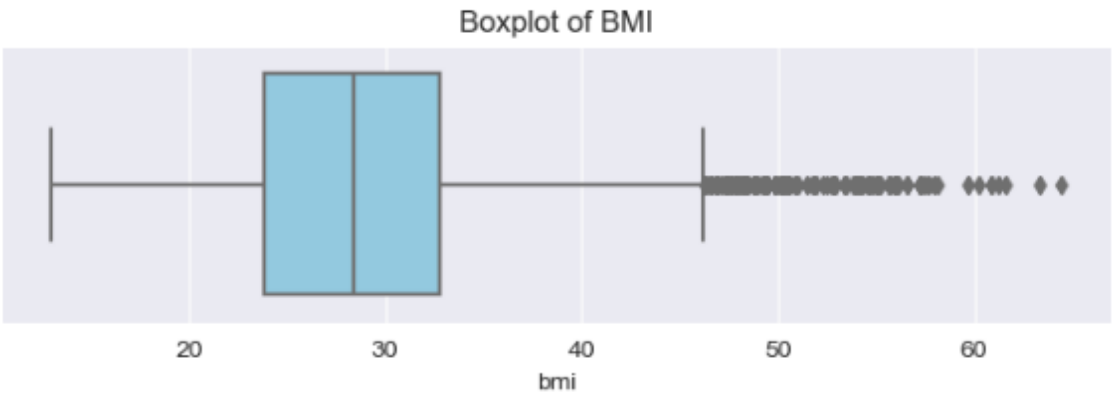
```
len of min threshold excluded 6
len of max threshold excluded 6
```

```
#Removing the extreme outliers of bmi
print("Old Shape: ", df.shape)

df.drop(index=df[df['bmi']<min_thre].index, inplace=True)
df.drop(index=df[df['bmi']>max_thre].index, inplace=True)

print("New Shape: ", df.shape)
```

```
Old Shape:  (5109, 11)
New Shape:  (5097, 11)
```

We utilized percentiles of <0.001 and >0.999 to primarily eliminate physiologically impossible outliers in BMI (errors). The overall rows we dropped was 12 — less than 0.3% of the dataset. We removed the 12 rows using the drop() method once more; we decided to do this to overall enhance the integrity and quality of the data frame as well as improving the model quality by removing noise and preventing biases.

```
#Boxplot which reflects the new BMI distribution
sns.set_style("darkgrid")
plt.figure(figsize=(8,2))
sns.boxplot(x='bmi', data=df, color='skyblue')
plt.title('Boxplot of BMI')
plt.show()
```

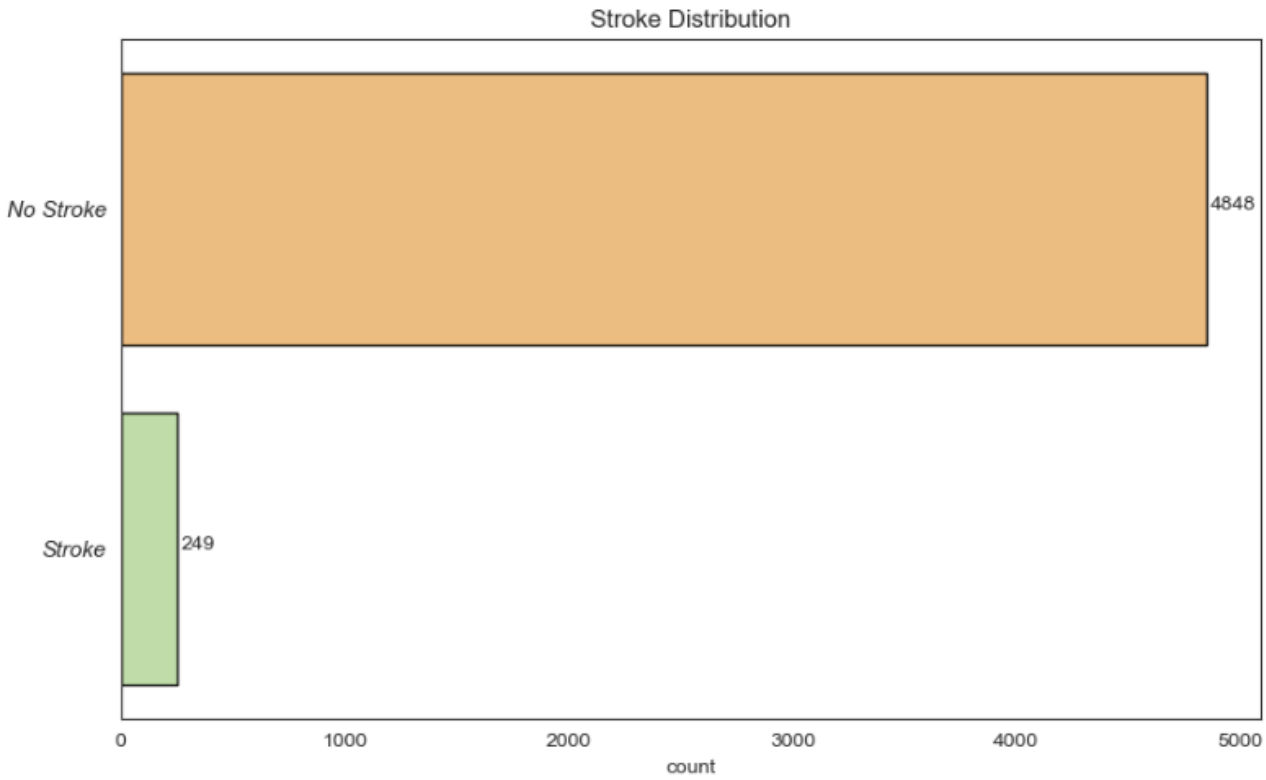


Updated boxplot for BMI, we can clearly observe the extreme outliers which were errors are removed (97.4, etc.,). Now that the data cleaning process is over, we can now move on to Exploratory Data Analysis (EDA).

Exploratory data analysis

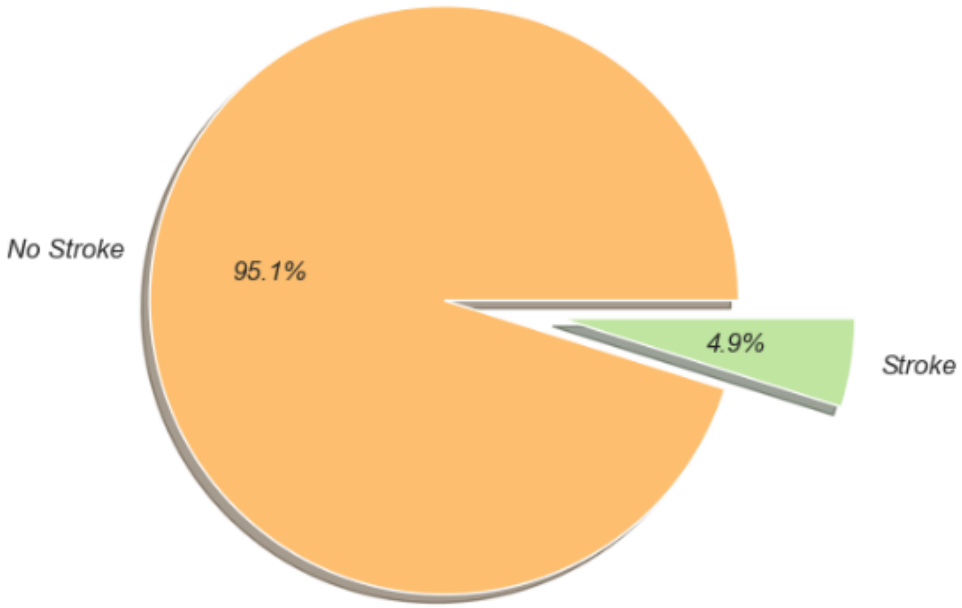
Exploratory Data Analysis (EDA): the process of analyzation and visualization to summarize the attributes and their corresponding relationships, to gain valuable insights about the dataset.

```
#Countplot depicting Stroke Distribution
plt.figure(figsize=(10, 6))
sns.set_style("white")
ax = sns.countplot(y='stroke', data=df, palette='Spectral', edgecolor='black', linewidth=1)
plt.title('Stroke Distribution')
plt.ylabel('')
plt.yticks([0, 1], ['No Stroke', 'Stroke'], fontsize = 11, style='italic')

#Add count as annotations
for p in ax.patches:
    ax.annotate(f'{int(p.get_width())}', (p.get_width() + 20, p.get_y() + 0.4), fontsize=10)

plt.show()
```



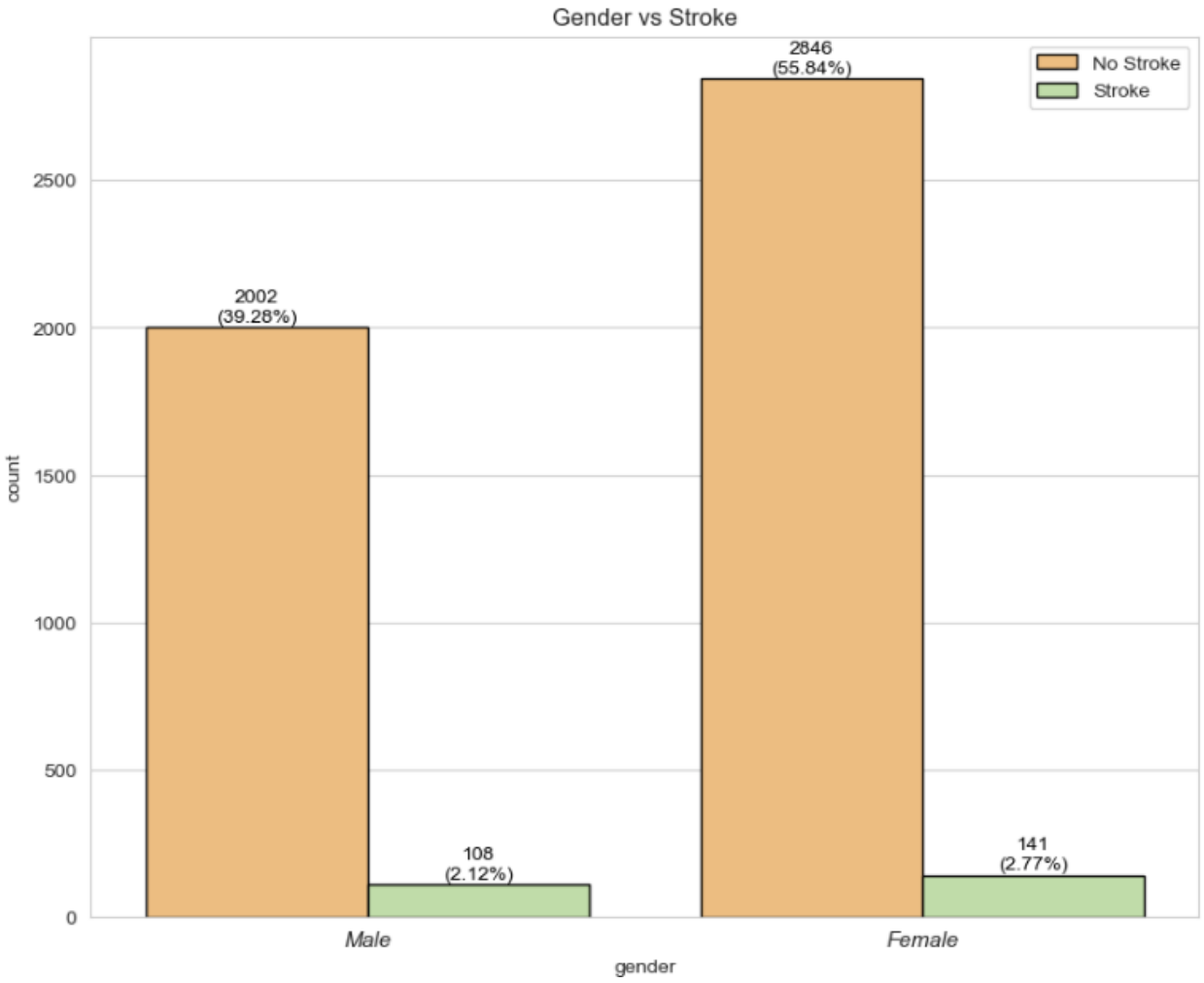
This count plot displays the count of the stroke feature. Clearly there is a vast difference between the values: 4848 people have not had a stroke, whereas only 249 patients had a stroke. The dataset is unbalanced and will need to be amended before the feature engineering/modelling phase, otherwise it will lead to terribly biased and skewed data — impacting the models evaluation on predicting whether patients have a stroke catastrophically.

```
#Pie chart displaying the proportion of stroke
plt.figure(figsize=(10,6))
plt.pie(df.stroke.value_counts(),labels=['No Stroke','Stroke'],autopct='%1.1f%%',
colors=sns.color_palette('Spectral', 2),explode=(0.3,0.1),shadow=True,textprops={'fontsize':12,'style':'italic'})
plt.show()
```



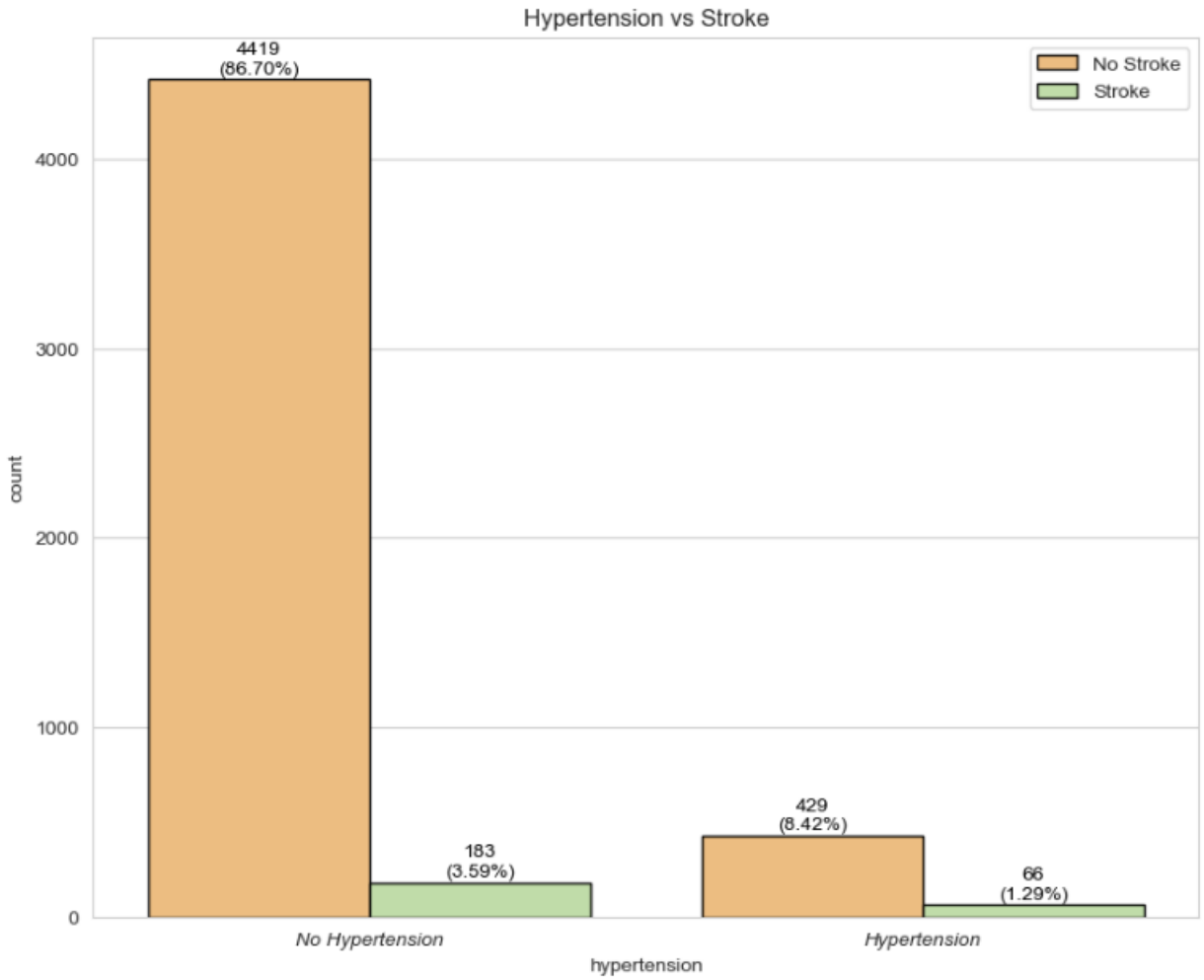
This pie chart represents the proportion of the stroke feature – the label we are trying to predict. It paints a clearer picture of just how heavily imbalanced the dataset actually is: 95.1% of people within the dataset haven't experienced a stroke and only 4.9% of the people have suffered from a stroke.

```python
#Countplot depicting gender and stroke correlation
plt.figure(figsize=(10,8))
sns.set_style("whitegrid")
ax = sns.countplot(x='gender', hue="stroke", data=df, palette='Spectral', edgecolor='black', linewidth=1)
plt.title('Gender vs Stroke')
plt.xticks(style='italic', fontsize=11)
plt.legend(labels=['No Stroke', 'Stroke'])

#Add count and percentage on top of each bar
for p in ax.patches:
    height = p.get_height()
    count = int(height) if height.is_integer() else height
    ax.annotate(f'{count}\n({height/len(df)*100:.2f}%)', (p.get_x() + p.get_width() / 2., height),
                ha='center', va='center', fontsize=10, color='black', xytext=(0, 10), textcoords='offset points')
plt.show()
```



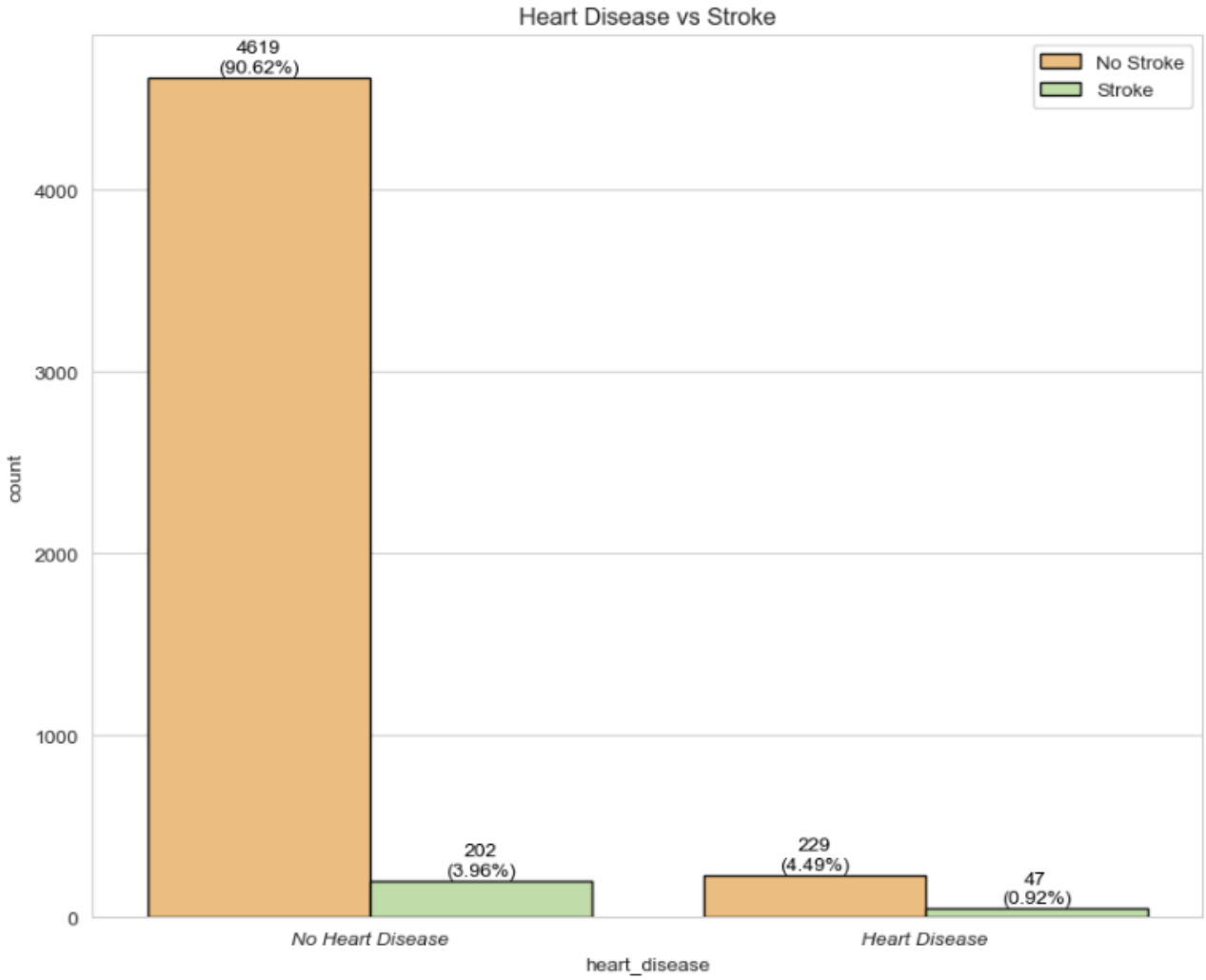
This count plot illustrates the gender feature and its correlation to stroke. While females are more numerous than males in the dataset (2987 vs 2110), and they also experience more strokes overall (141 vs 108), proportionally, males show a slightly higher likelihood of experiencing a stroke compared to females (stroke rates of 5.4% vs 4.7%).

```
#Countplot depicting hypertension and stroke correlation
plt.figure(figsize=(10,8))
sns.set_style("whitegrid")
ax = sns.countplot(x='hypertension',hue="stroke",data=df, palette='Spectral', edgecolor = 'black', linewidth = 1)
plt.title('Hypertension vs Stroke')
plt.xticks([0,1], ['No Hypertension','Hypertension'], style = 'italic')
plt.legend(labels=['No Stroke','Stroke'])

#Add count and percentage on top of each bar
for p in ax.patches:
    height = p.get_height()
    count = int(height) if height.is_integer() else height
    ax.annotate(f'{count}\n({height/len(df)*100:.2f}%)', (p.get_x() + p.get_width() / 2., height),
                ha='center', va='center', fontsize=10, color='black', xytext=(0, 10), textcoords='offset points')
plt.show()
```



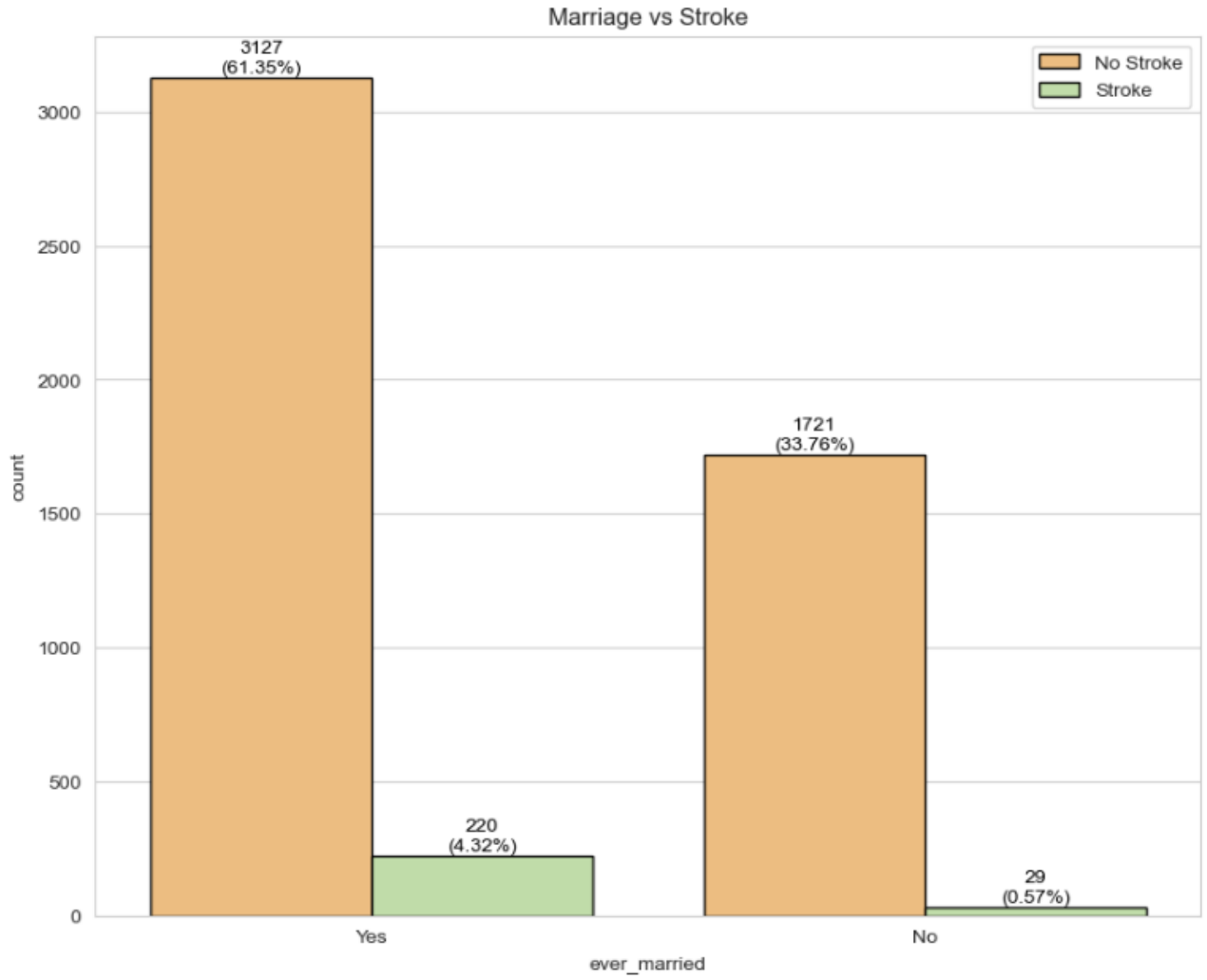
This count plot depicts hypertension and its relationship to stroke. Comprehensively there is a significant disparity: 4602 individuals do not have hypertension, with 183 of them experiencing a stroke. In contrast, around 495 individuals have hypertension, with approximately 66 of them suffering from a stroke. Proportionally patients with hypertension are 9% more likely to experience a stroke compared to those without hypertension (stroke rates of 13% vs 4%).

```
#Countplot depicting heart_disease and stroke correlation
plt.figure(figsize=(10,8))
sns.set_style("whitegrid")
ax = sns.countplot(x='heart_disease',hue="stroke",data=df, palette='Spectral', edgecolor = 'black', linewidth = 1)
plt.title('Heart Disease vs Stroke')
plt.xticks([0,1], ['No Heart Disease','Heart Disease'], style = 'italic')
plt.legend(labels=['No Stroke','Stroke'])

#Add count and percentage on top of each bar
for p in ax.patches:
    height = p.get_height()
    count = int(height) if height.is_integer() else height
    ax.annotate(f'{count}\n({height/len(df)*100:.2f}%)', (p.get_x() + p.get_width() / 2., height),
                ha='center', va='center', fontsize=10, color='black', xytext=(0, 10), textcoords='offset points')
plt.show()
```



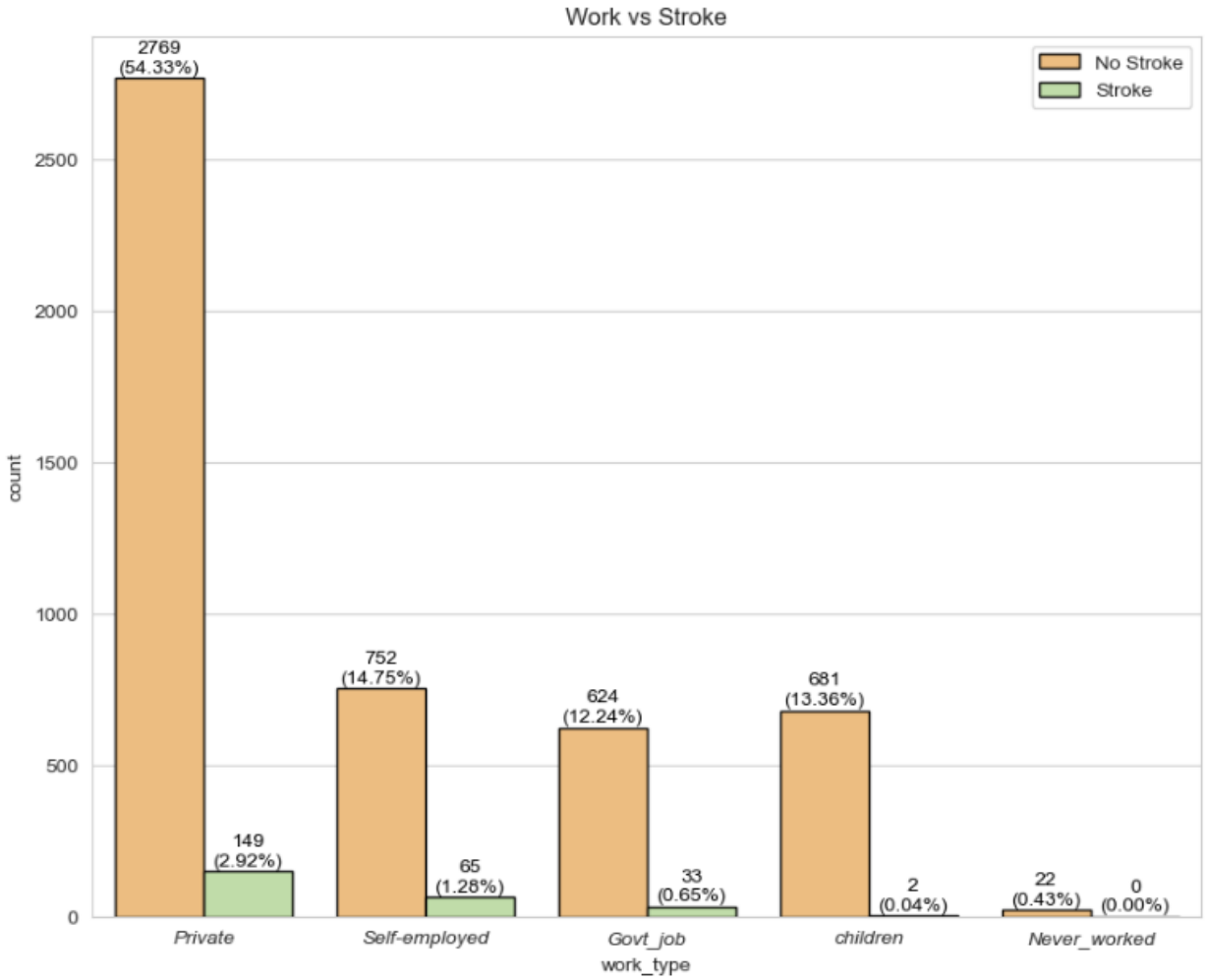
This count plot represents the heart disease feature and its correlation to stroke. Substantial difference — approximately 4821 people have no heart disease, with 202 of them experiencing a stroke. Comparatively, 276 people with heart disease, and 47 of them experienced a stroke. Proportionally, Individuals who have heart disease are 13% more likely to have a stroke, in contrast to those who haven't had heart disease (stroke rates of 17% vs 4%).

```
#Countplot depicting ever_married and stroke correlation
plt.figure(figsize=(10,8))
sns.set_style("whitegrid")
ax = sns.countplot(x='ever_married',hue="stroke",data=df, palette='Spectral', edgecolor = 'black', linewidth = 1)
plt.title('Marriage vs Stroke')
plt.legend(labels=['No Stroke','Stroke'])

#Add count and percentage on top of each bar
for p in ax.patches:
    height = p.get_height()
    count = int(height) if height.is_integer() else height
    ax.annotate(f'{count}\n({height/len(df)*100:.2f}%)', (p.get_x() + p.get_width() / 2., height),
                ha='center', va='center', fontsize=10, color='black', xytext=(0, 10), textcoords='offset points')
plt.show()
```



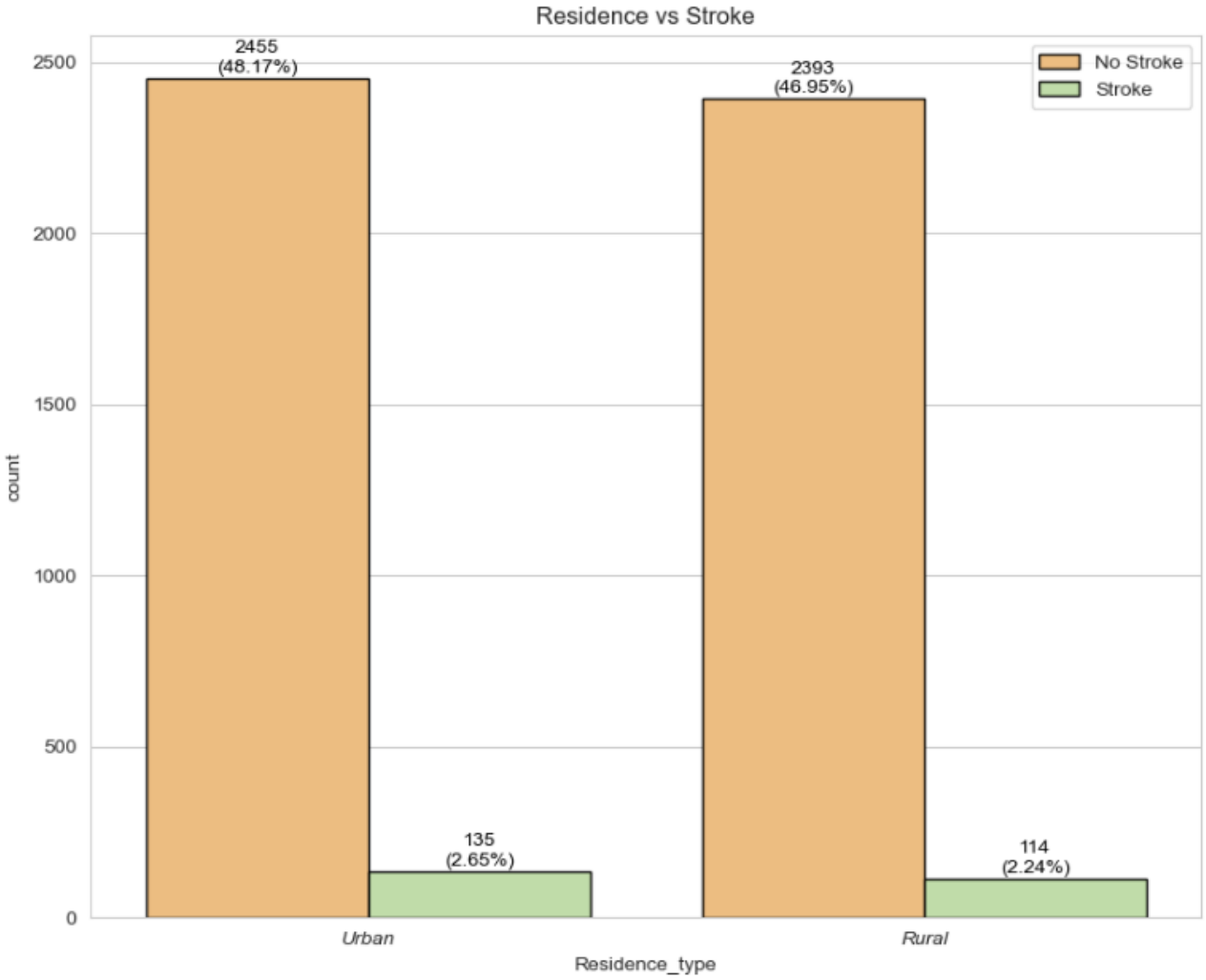
This count plot highlights the ever-married feature and its relationship with stroke. Among the 3347 married individuals, approximately 220 have experienced a stroke. Conversely, out of the 1721 unmarried individuals, 29 have had a stroke. Proportionally, married individuals are 5% more prone to strokes compared to unmarried individuals (with stroke rates of 6% versus 1%).

```
#Countplot depicting work_type and stroke correlation
plt.figure(figsize=(10,8))
sns.set_style("whitegrid")
ax = sns.countplot(x='work_type',hue="stroke",data=df, palette='Spectral', edgecolor = 'black', linewidth = 1)
plt.title('Work vs Stroke')
plt.xticks(style = 'italic')
plt.legend(labels=['No Stroke','Stroke'])

#Add count and percentage on top of each bar
for p in ax.patches:
    height = p.get_height()
    count = int(height) if height.is_integer() else height
    ax.annotate(f'{count}\n({height/len(df)*100:.2f}%)', (p.get_x() + p.get_width() / 2., height),
                ha='center', va='center', fontsize=10, color='black', xytext=(0, 10), textcoords='offset points')
plt.show()
```



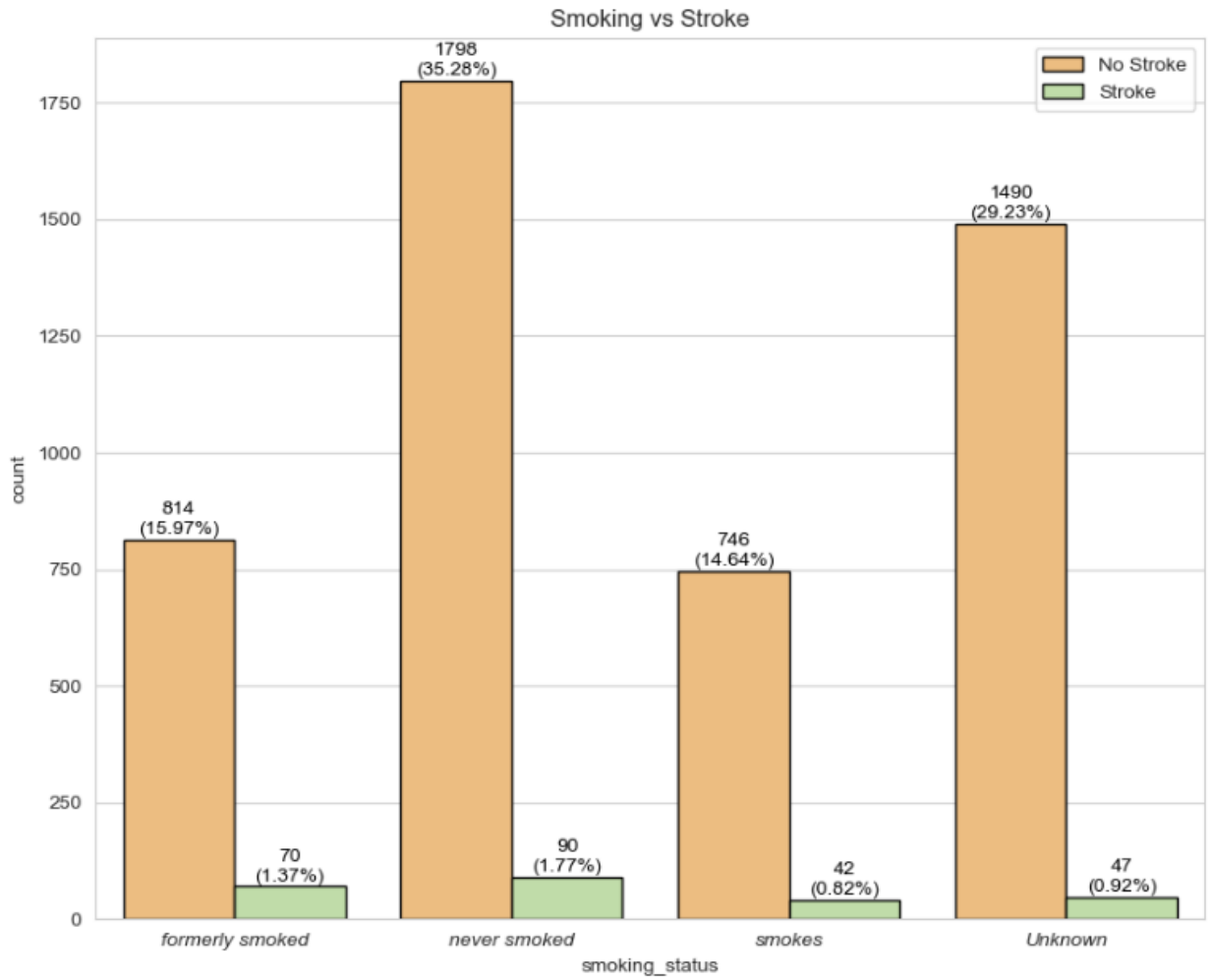
This count plot displays the work type feature and its correlation with stroke. Among the various work types, private work stands out with the highest count at 2918 individuals and the highest number of strokes at 149. In contrast, the "never worked" category has the smallest count at 22, with no reported strokes. However, in terms of proportions, self-employment displays the highest stroke rate at 8%, encompassing 817 individuals, among whom 65 have experienced a stroke. This is followed up by private and government employment, both at 5% stroke occurrence rates, respectively.

```
#Countplot depicting Residence_type and stroke correlation
plt.figure(figsize=(10,8))
sns.set_style("whitegrid")
ax = sns.countplot(x='Residence_type',hue="stroke",data=df, palette='Spectral', edgecolor = 'black', linewidth = 1)
plt.title('Residence vs Stroke')
plt.xticks(style = 'italic')
plt.legend(labels=['No Stroke','Stroke'])

#Add count and percentage on top of each bar
for p in ax.patches:
    height = p.get_height()
    count = int(height) if height.is_integer() else height
    ax.annotate(f'{count}\n({height/len(df)*100:.2f}%)', (p.get_x() + p.get_width() / 2., height),
                ha='center', va='center', fontsize=10, color='black', xytext=(0, 10), textcoords='offset points')
plt.show()
```



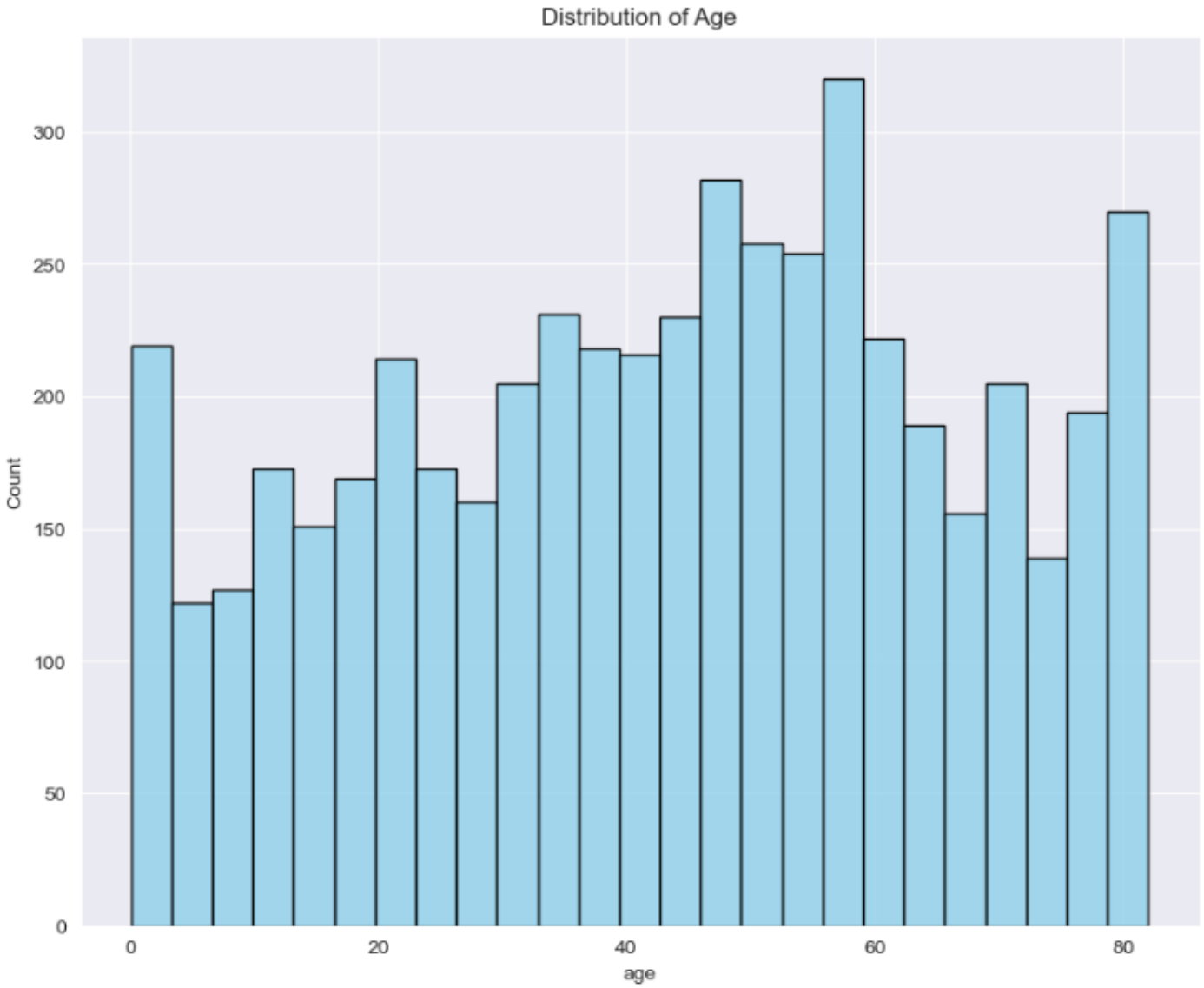
This count plot depicts the feature residence type and its relation to stroke. Among the two categories, urban residents have a slightly higher count at 2590 individuals, along with a greater number of strokes at 135. In comparison, rural residents number 2507, with 114 experiencing strokes. Proportionally speaking, urban inhabitants exhibit a slightly elevated likelihood of experiencing a stroke compared to rural inhabitants (stroke rates of 5% vs 4%)

```
#Countplot depicting smoking_status and stroke correlation
plt.figure(figsize=(10,8))
sns.set_style("whitegrid")
ax = sns.countplot(x='smoking_status',hue="stroke",data=df, palette='Spectral', edgecolor = 'black', linewidth = 1)
plt.title('Smoking vs Stroke')
plt.xticks(style = 'italic')
plt.legend(labels=['No Stroke','Stroke'])

#Add count and percentage on top of each bar
for p in ax.patches:
    height = p.get_height()
    count = int(height) if height.is_integer() else height
    ax.annotate(f'{count}\n({height/len(df)*100:.2f}%)', (p.get_x() + p.get_width() / 2., height),
                ha='center', va='center', fontsize=10, color='black', xytext=(0, 10), textcoords='offset points')
plt.show()
```



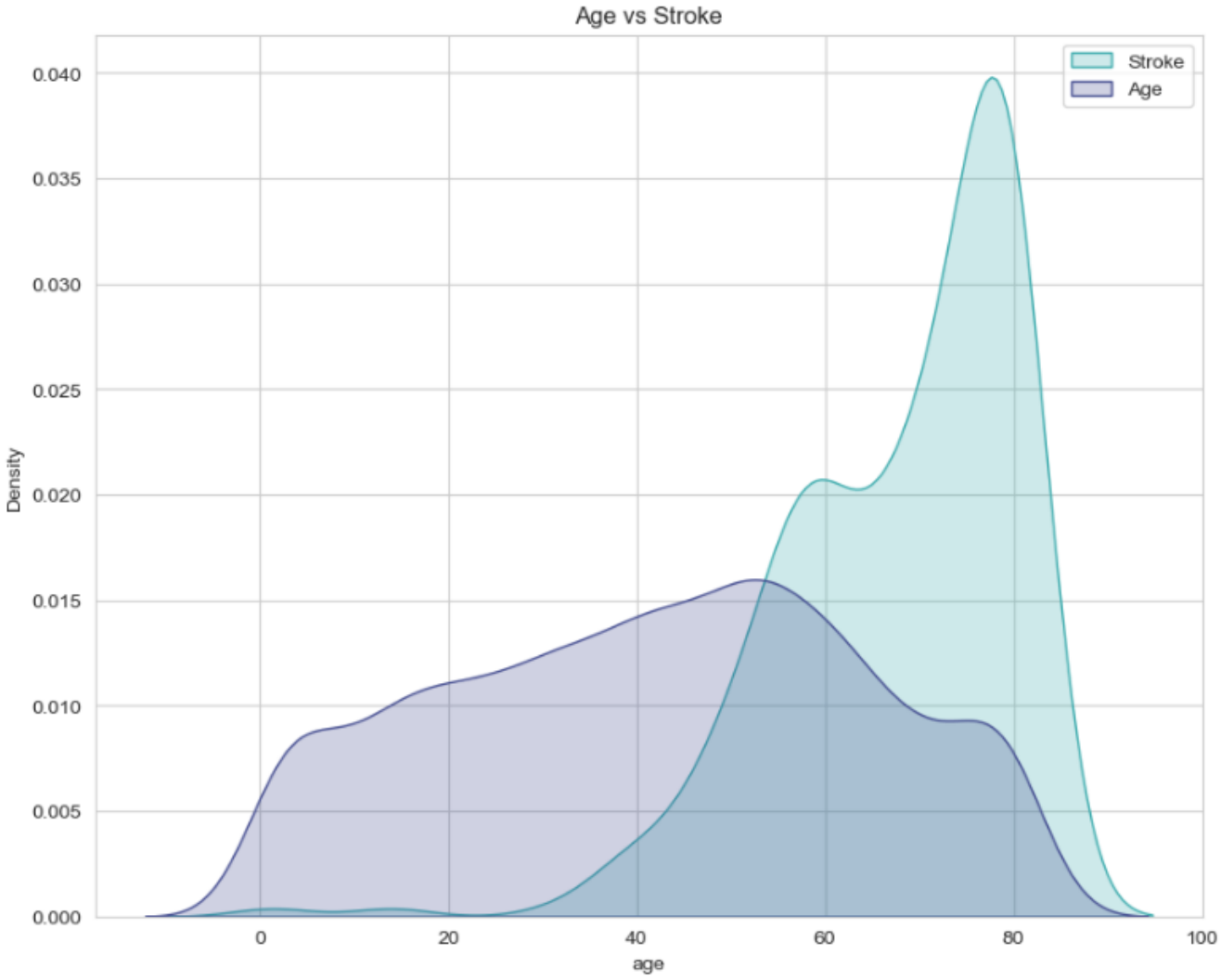
This count plot illustrates the smoking status feature and its correlation to stroke. Among the different categories, the highest count is observed in the never smoked category, totaling 1888 individuals, out of which 90 have experienced a stroke. Conversely, the smokes category has the lowest count, along with the fewest strokes. Proportionally, the formerly smoked category exhibits the highest stroke rate: comprising 884 individuals, with 70 experiencing a stroke — people who have formerly smoked are 8% more likely to have a stroke.

```
#Histogram which reflects the Age distribution
plt.figure(figsize=(10,8))
sns.set_style("darkgrid")
sns.histplot(data=df, x='age', bins = 25, color='skyblue', edgecolor = 'black', linewidth = 1)
plt.title('Distribution of Age')
plt.show()
```



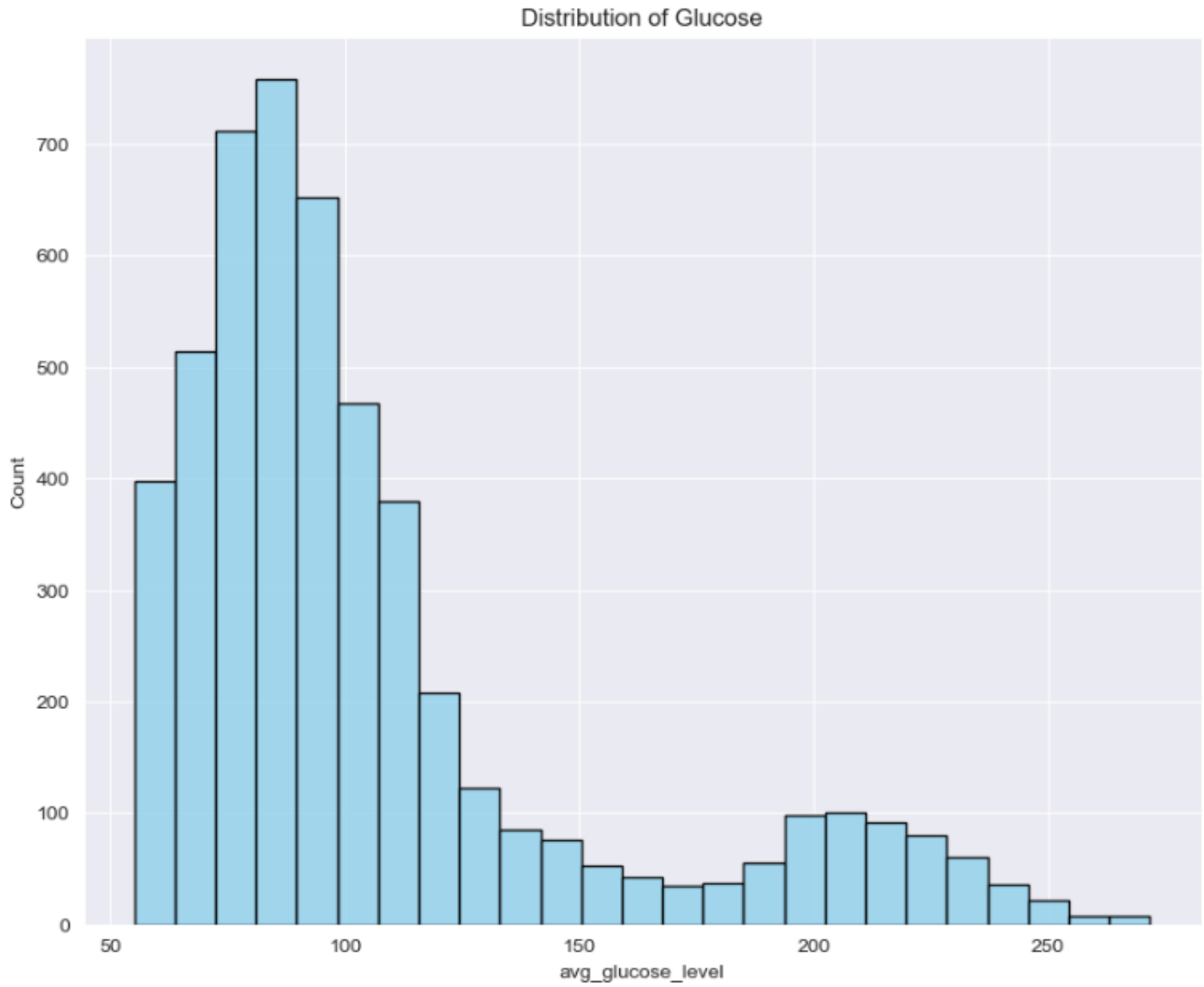
This histogram illustrates the distribution of the age feature. The ages of individuals within the data are aged between 0-80. The majority of individuals fall within the vicinity of 58 years old, totaling approximately 320 counts. In contrast, the minority contingent is represented by individuals aged 3 years old, comprising roughly 120 counts.

```
#KDE plot which reflects the age vs stroke correlation
plt.figure(figsize=(10,8))
sns.set_style("whitegrid")
sns.kdeplot(data=df, x='age', hue='stroke', fill=True, common_norm=False, palette='mako')
plt.legend(labels=['Stroke','Age'])
plt.title('Age vs Stroke')
plt.show()
```



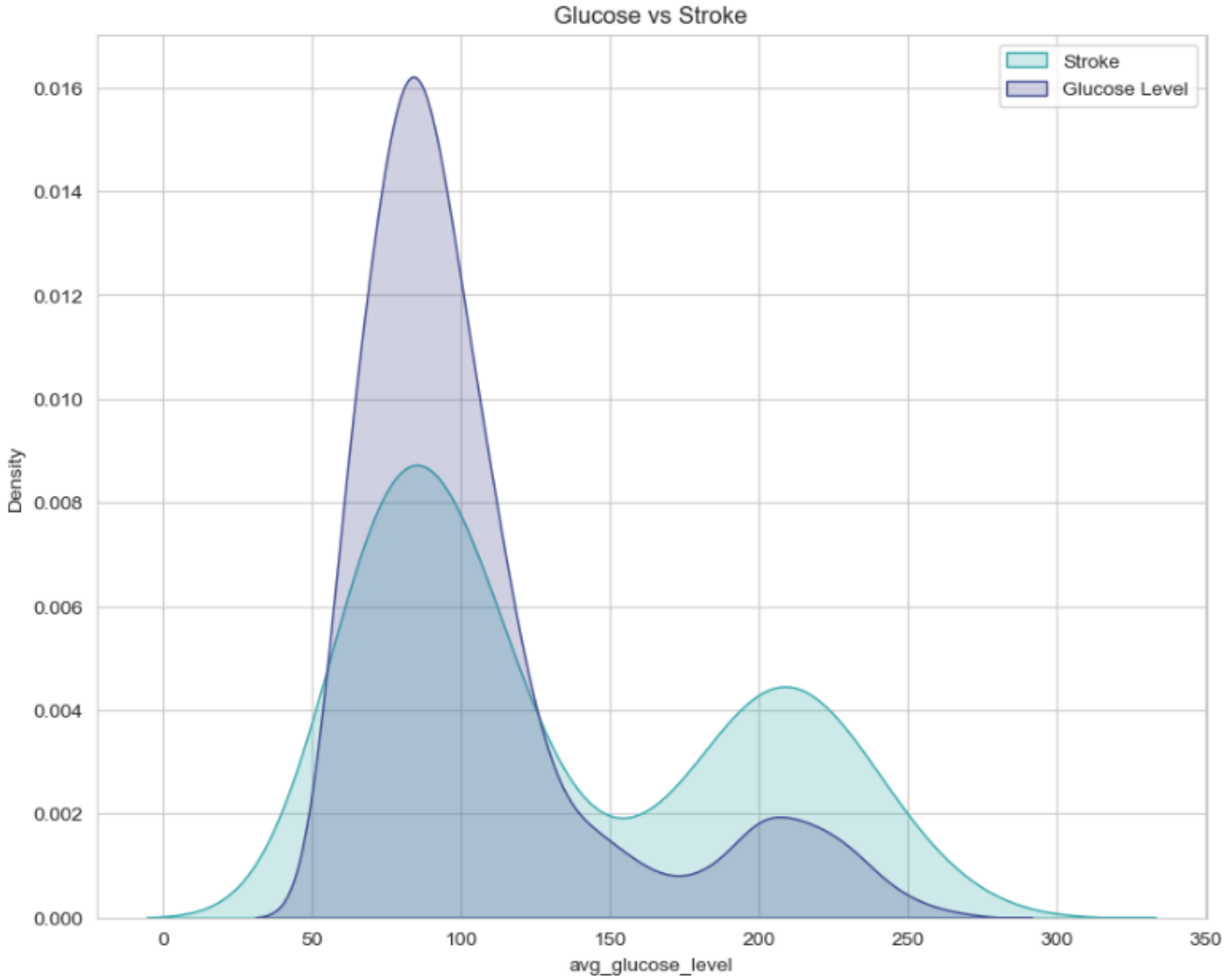
This Kernel Density Estimation (KDE) plot illustrates how the occurrence of strokes correlates with age. Notably, the stroke curve begins to sharply ascend around the age of 40 and reaches its peak between 76 and 78 years old. This suggests that strokes predominantly affect elderly individuals within this age range. The observed trends in the curve strongly indicate a correlation between age and stroke occurrence, highlighting the significance of age as a contributing factor to strokes.

```
#Histogram which reflects the gluclose distribution
plt.figure(figsize=(10,8))
sns.set_style("darkgrid")
sns.histplot(data=df, x='avg_glucose_level', bins = 25, color='skyblue', edgecolor = 'black', linewidth = 1)
plt.title('Distribution of Glucose')
plt.show()
```



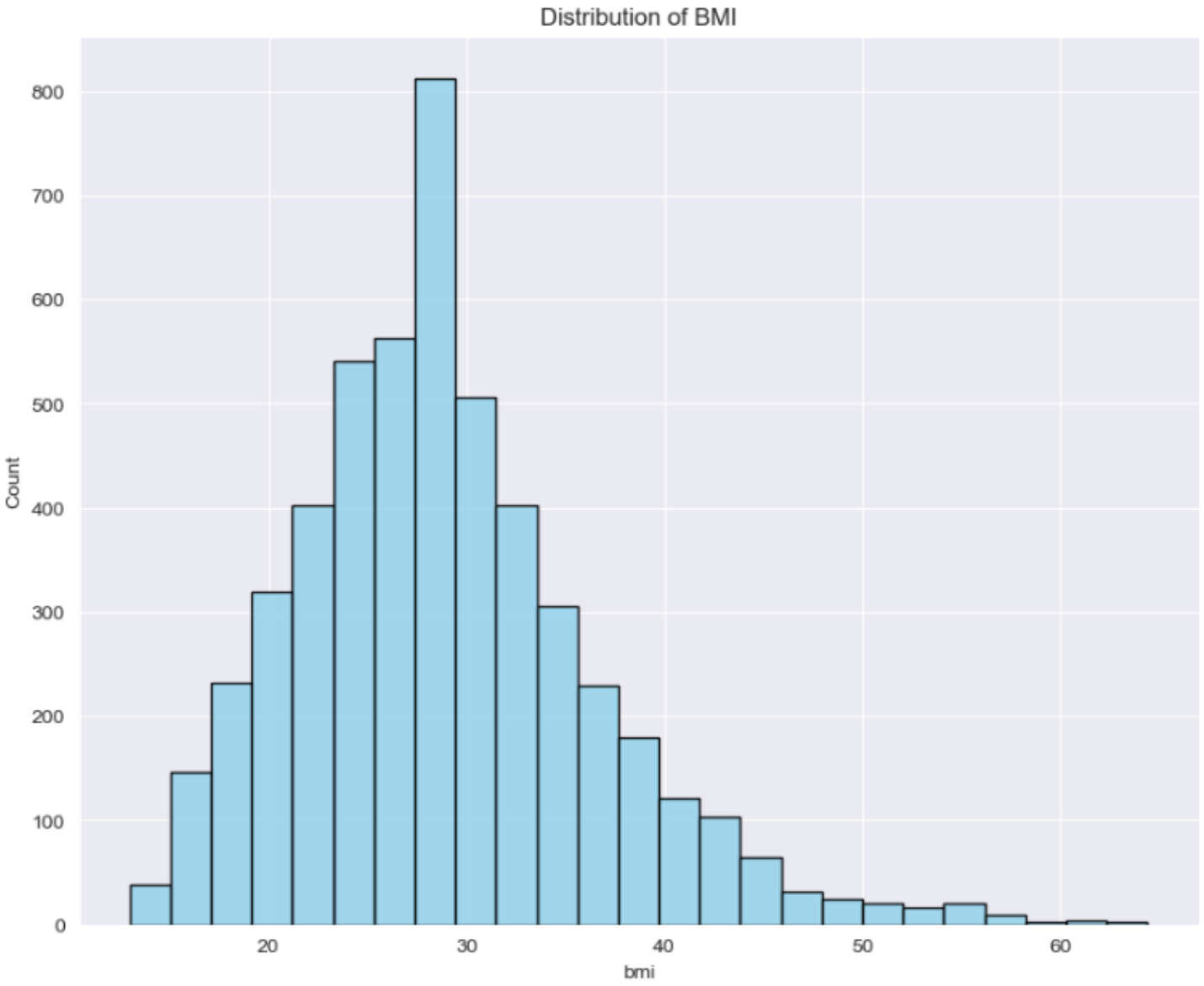
This histogram depicts the distribution of the glucose feature. Glucose levels fluctuate between 55-271 mg/dL within the data frame. The most common glucose levels are clustered around 90 mg/dL, accounting for roughly 750 observations. Conversely, the smallest subset of values are valid outliers, with values exceeding 250 mg/dL and peaking at 271 mg/dL.

```
#KDE plot which reflects the glucose level vs stroke correlation
plt.figure(figsize=(10,8))
sns.set_style("whitegrid")
sns.kdeplot(data=df, x='avg_glucose_level', hue='stroke', fill=True, common_norm=False, palette='mako')
plt.legend(labels=['Stroke','Glucose Level'])
plt.title('Glucose vs Stroke')
plt.show()
```



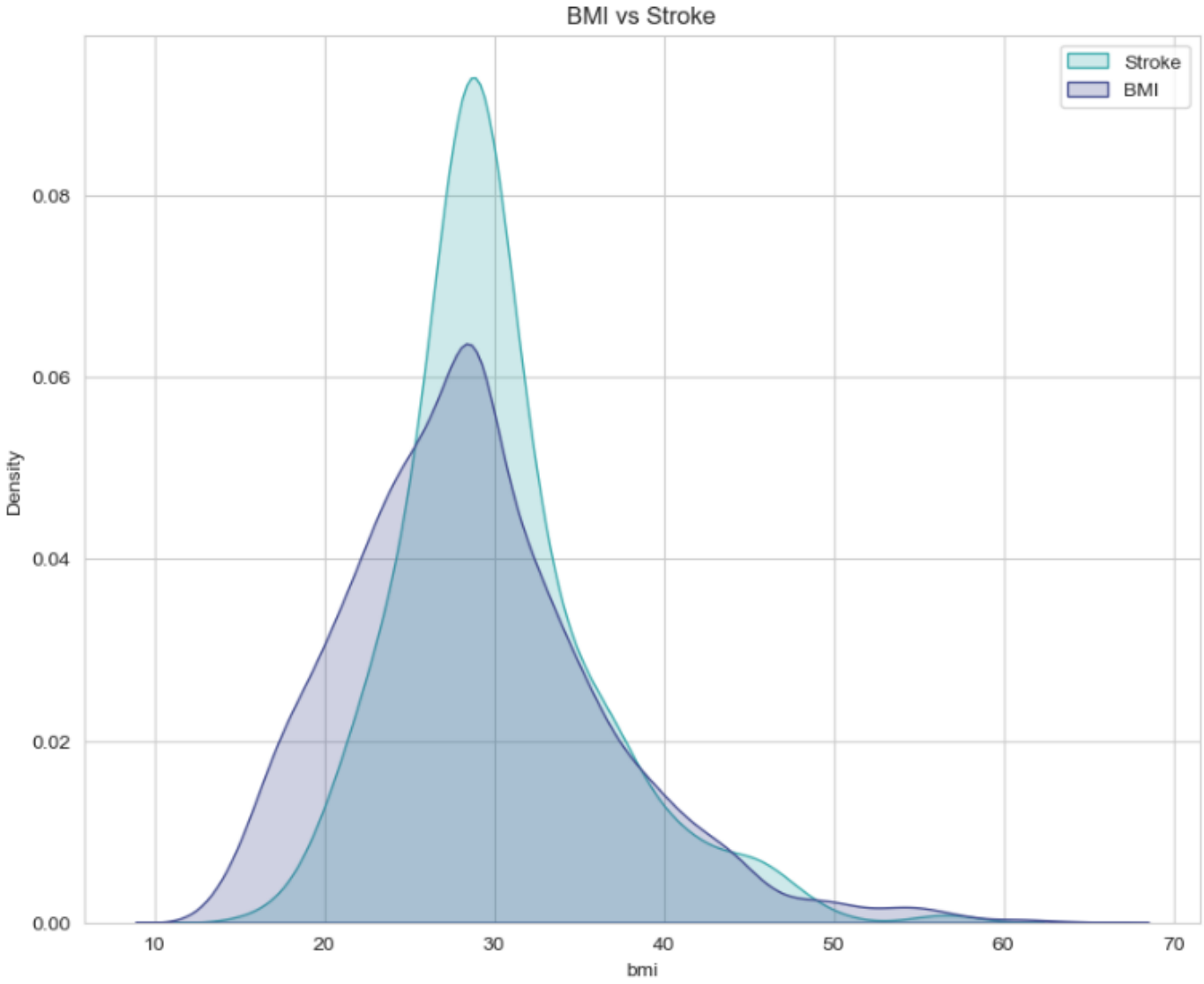
The Kernel Density Estimation (KDE) plot visualizes the relationship between glucose levels and strokes. The curve representing strokes starts to rise at approximately 50 mg/dL of glucose and peaks around 90 mg/dL. This implies that strokes are most prevalent among individuals with glucose levels within this range. The observed patterns in the curve suggest a connection between glucose levels and the likelihood of experiencing a stroke.

```
#Histogram which reflects the bmi distribution
plt.figure(figsize=(10,8))
sns.set_style("darkgrid")
sns.histplot(data=df, x='bmi', bins = 25, color='skyblue', edgecolor = 'black', linewidth = 1)
plt.title('Distribution of BMI')
plt.show()
```



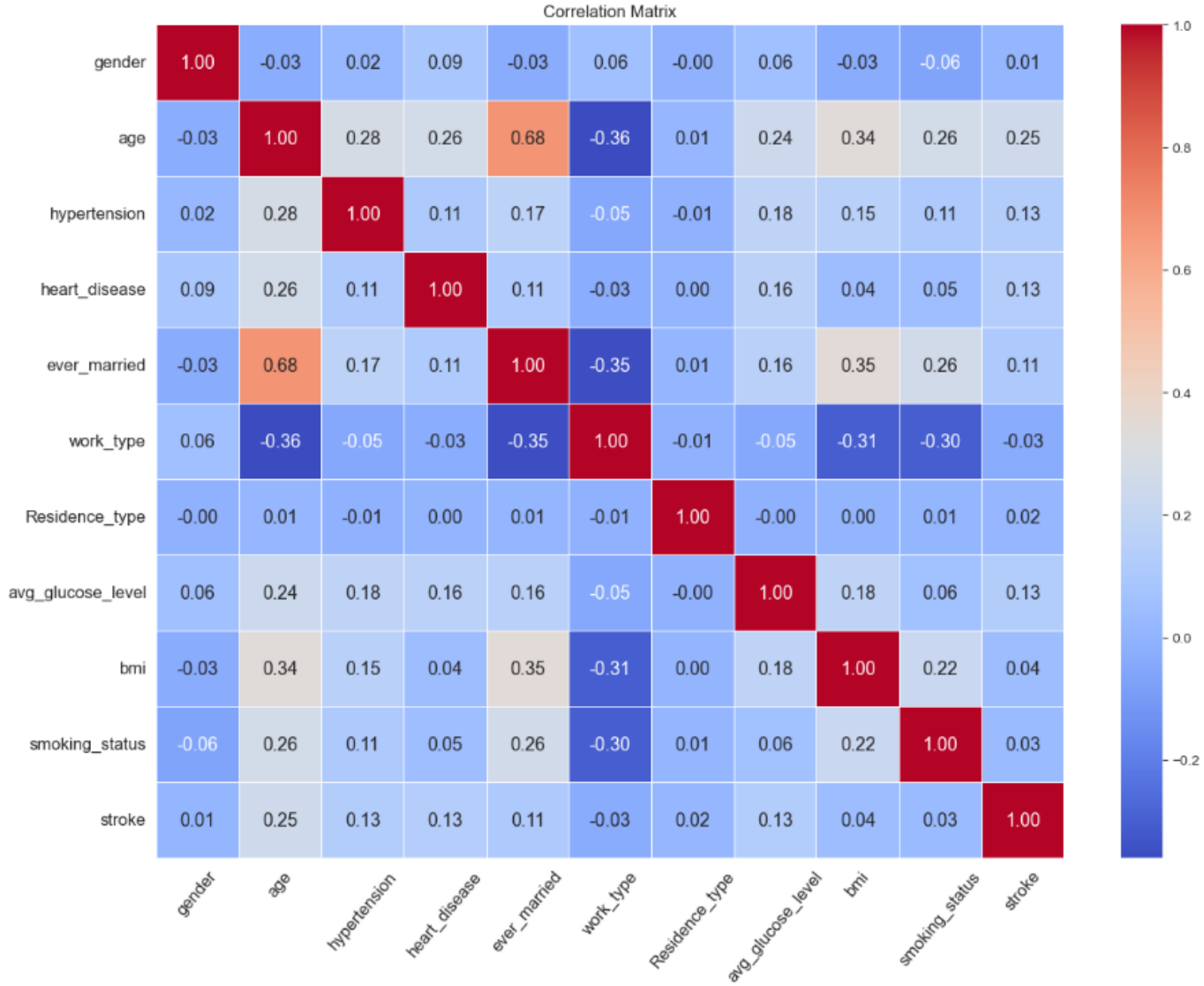
This histogram displays the distribution of the BMI feature. BMI values range from 13-64 within the data. The most frequently occurring BMI value is centered around 28, accounting for approximately 810 observations. In contrast, the smallest subset of BMI values are valid outliers, with values exceeding 60 and peaking at 64 BMI.

```
#KDE plot which reflects the bmi vs stroke correlation
plt.figure(figsize=(10,8))
sns.set_style("whitegrid")
sns.kdeplot(data=df, x='bmi', hue='stroke', fill=True, common_norm=False, palette='mako')
plt.legend(labels=['Stroke','BMI'])
plt.title('BMI vs Stroke')
plt.show()
```



The Kernel Density Estimation (KDE) plot depicts the correlation between BMI and stroke occurrence. Notably, the stroke curve begins to steeply increase at approximately 20 BMI and reaches its highest point around 28-29 BMI. This indicates that strokes are most common among individuals with BMI values in the range of 28-29. The trends observed in the curve suggest a correlation between BMI levels and the risk of experiencing a stroke.

```
#Data correlations between all features
plt.figure(figsize=(13,10))
sns.heatmap(df.corr(numeric_only = True),annot=True, fmt = ".2f", cmap = "coolwarm", linewidths=0.5, annot_kws={"size":13})
plt.title('Correlation Matrix')
plt.yticks(rotation= 0, fontsize = 12)
plt.xticks(rotation=50,fontsize = 12)
plt.tight_layout()
plt.show()
```



This is a correlation matrix between all features, The bottom row is strokes correlation against every other feature. The closer the number is to 1 (or -1), the stronger the positive or negative correlation is.

- Age, hypertension, heart_disease, avg_glucose_level, ever married have a positive correlation with stroke — with age having the highest rate of correlation.
- Gender, residence type, BMI, and smoking status has low correlation with stroke.
- Work type has a slightly negative correlation with stroke.

Feature Engineering

For feature engineering, we are going to run a code that performs feature selection by iteratively dropping each column individually and evaluating the model's performance (random forest classifier). It then prints the best scoring combination of columns based on metrics such as accuracy, precision, recall, and F1-score. Essentially, it runs every possible combination from the low/negative correlated features to our target stroke (gender, work type, residence type, BMI, smoking status) and prints out the best scoring combination.

```
best_score = {'accuracy': 0, 'precision': 0, 'recall': 0, 'f1': 0}
best_columns = []

#Function to train and evaluate the model
def train_evaluate_model(X_train, X_test, y_train, y_test):
    global best_score, best_columns

    #Train the model
    model = RandomForestClassifier()
    model.fit(X_train, y_train)

    #Predict on the test set
    y_pred = model.predict(X_test)

    #Evaluate the model
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    #Print evaluation metrics
    print(f"Accuracy: {accuracy}")
    print(f"Precision: {precision}")
    print(f"Recall: {recall}")
    print(f"F1 Score: {f1}")

    #Update best score and columns if necessary
    if accuracy > best_score['accuracy'] and precision > best_score['precision'] \
        and recall > best_score['recall'] and f1 > best_score['f1']:
        best_score['accuracy'] = accuracy
        best_score['precision'] = precision
        best_score['recall'] = recall
        best_score['f1'] = f1
        best_columns = X_train.columns.tolist()

#Test dropping each column individually
columns_to_test = ['gender', 'work_type', 'Residence_type', 'bmi', 'smoking_status']
for column in columns_to_test:
    print(f"Testing without dropping column '{column}':")
    X_train_no_column = X_train.drop(column, axis=1)
    X_test_no_column = X_test.drop(column, axis=1)
    train_evaluate_model(X_train_no_column, X_test_no_column, y_train, y_test)
    print("----------------------------------------------")

#Test combinations of columns
for r in range(1, len(columns_to_test) + 1):
    print(f"Testing combinations of {r} columns:")
    for combination in combinations(columns_to_test, r):
        print(f"Testing combination: {combination}")
        columns_to_drop = list(combination)
        X_train_no_columns = X_train.drop(columns_to_drop, axis=1)
        X_test_no_columns = X_test.drop(columns_to_drop, axis=1)
        train_evaluate_model(X_train_no_columns, X_test_no_columns, y_train, y_test)
        print("----------------------------------------------")

#Print the best scoring combination
print("Best scoring combination of columns:")
print(best_columns)
```

```
Best scoring combination of columns:
['gender', 'age', 'hypertension', 'heart_disease', 'ever_married', 'work_type', 'Residence_type', 'avg_glucose_level', 'bmi']
```

Based on the output, we can deduce that the most optimal combination of columns, in terms of metrics, is one that includes all features except for 'smoking_status'. In light of this, we are now going to remove the smoking status feature.

```
#Dropping smoking_status
df.drop('smoking_status',axis=1,inplace=True)
df.head(1)
```

Removing smoking status from the data frame using the drop() method. In essence, this was done to increase model performance, as removing smoking status was necessary to maximize the performance metrics of our models as it was not a part of the most optimal combinations of features.

```
#Converting categorical to numerical data
df = pd.get_dummies(df, columns =  ['gender', 'ever_married', 'work_type', 'Residence_type'])
```

Now we have to convert categorical data to numerical data. This is necessary as the features gender, ever married, work type, and residence type have words in them, and machine learning algorithms cannot process words, only numbers. I'll be utilizing the get_dummies() method from pandas to handle the conversion process. This method transforms categorical data into dummy variables, creating binary (0/1) variables for each unique value in the categorical feature. We've opted for get_dummies over the widely used label encoder provided by sklearn because it typically delivers superior performance during the modeling stage and is more compatible with the various algorithms we'll be employing.

```
#Feature-splitting
X = df.drop('stroke', axis = 1)
y = df['stroke']
```

Before we start modelling, we need to split the features into X and y, with y being just the target (stroke feature) and X being every other feature which will be used to predict the target.

```
#Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)
```

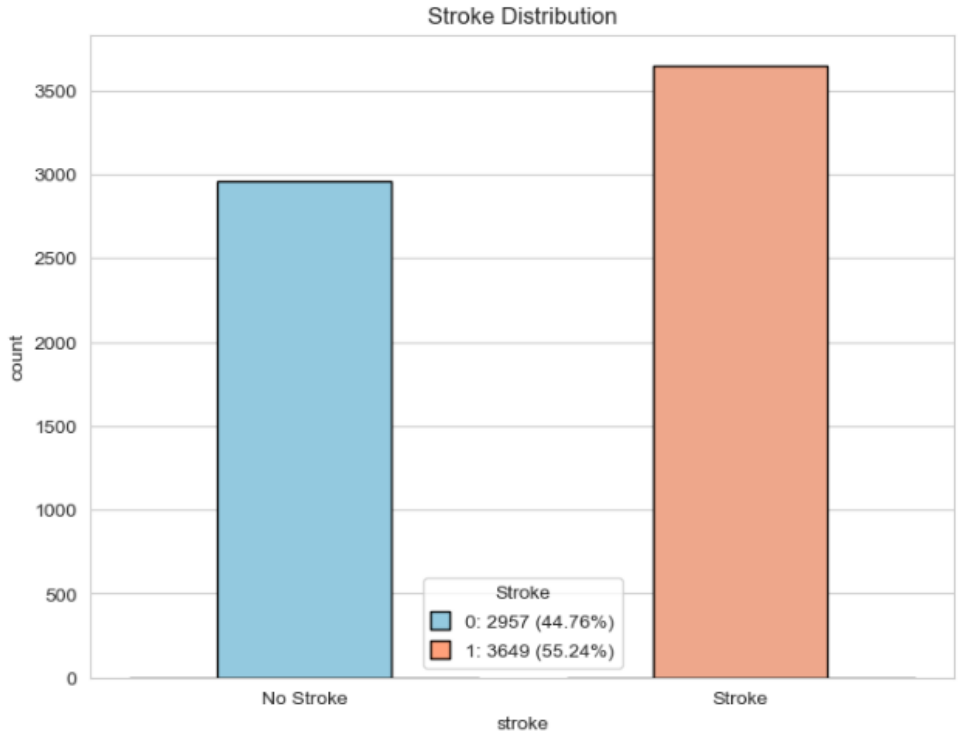
Now, we split our data into train and test sets by using the train_test_split() method from sklearn. The method works by splitting our data into (X_train/y_train and X_test/y_test) sets and in our configuration the split is 80% for training, 20% reserved for the test set. X represents the features of the dataset and y represents the stroke class we want to predict. The training set is used to train the machine learning model, during training the model acquires insights into the patterns and relationships in the data, facilitating its ability to classify. Following training, the model needs to be assessed against unseen data to determine how well it performs on making classifications. The testing set provides the unseen data — allowing us to evaluate the models.

```
#Apply SMOTEENN to balance the stroke class
smoteenn = SMOTEENN(random_state=42)
X_train, y_train = smoteenn.fit_resample(X_train, y_train)
print('After Balancing:')
print('X balanced shape:', X_train.shape)
print('y balanced shape:', y_train.shape)
```

```
After Balancing:
X balanced shape: (6606, 16)
y balanced shape: (6606,)
```

We needed to balance the dataset, as it was heavily imbalanced — the minority class of stroke only made up for 4.9% of the data frame. We want to preserve our data, but also re-balance our data at the same time, the solution: over samplers. Vigorous testing went into finding the best over sampler, testing multiple popular over samplers such as SMOTE, ADASYN, Borderline SMOTE, and SMOTEENN. Ultimately, we found SMOTEENN to be the best performing over sampler for this dataset. SMOTEENN stands for Synthetic Minority Over-sampling Technique (SMOTE) combined with Edited Nearest Neighbours (ENN). It's a method designed to address imbalanced datasets in machine learning. It combines two techniques: SMOTE, which creates synthetic samples for the minority class to balance class distribution, and ENN, which removes instances misclassified by a k-nearest neighbour classifier to clean the dataset from noise. In essence, SMOTEENN aims to enhance classification performance of the models, particularly when dealing with imbalanced data where the minority class is inadequately represented. It achieves this by providing more balanced training data, which helps the model learn more effectively and make better predictions on both training and testing data.

```
#Countplot depicting Stroke Distribution

#Balanced df
df_balanced = pd.DataFrame()
df_balanced[X.columns] = X_train
df_balanced['stroke'] = y_train

#Calculate counts for each option
counts = df_balanced['stroke'].value_counts()

#Define colors for each option
colors = {0: 'skyblue', 1: 'lightsalmon'}

#Configure countplot
plt.figure(figsize=(8,6))
sns.set_style("whitegrid")
ax = sns.countplot(x='stroke', data=df_balanced, palette=colors, edgecolor='black', linewidth=1, width=0.4)
plt.title('Stroke Distribution')
plt.xticks([0,1], ['No Stroke', 'Stroke'])

#Add legend with counts and percentages
legend_labels = []
for i, count in enumerate(counts):
    label_text = f'{1-i}: {count} ({(count/df_balanced.shape[0]*100):.2f}%)'
    legend_labels.append(label_text)
    plt.bar(i, 0, color=colors[1-i], label=label_text)

#Customize legend handles
handles, _ = ax.get_legend_handles_labels()
for handle in handles:
    for patch in handle.patches:
        patch.set_edgecolor('black')

plt.legend(title='Stroke', loc='lower center', labels=legend_labels[::-1], handlelength=1, handleheight=1)
plt.show()
```



The count plot displays the updated stroke distribution, we can clearly observe it has been rebalanced drastically: 2957 people have not suffered from a stroke and 3649 have suffered

from a stroke. Proportionally speaking, originally the dataset had 95.1% of individuals having no stroke, with 4.9% of individuals having stroke. Now the classes are reversed with experiencing a stroke being the majority class at 55.24% and no stroke being the minority class at 44.76%. Stroke needed to be rebalanced as the models would be biased and perform poorly, especially towards the minority class (having stroke). By rebalancing the stroke class, we ensure that the model learns from a more representative set of data, therefore improving its ability to make accurate predictions for both classes (no stroke & stroke). Ultimately, balancing prevents the models from being overly influenced by the majority class — leading to better model performance.

```
#Feature Scaling
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

Feature scaling X_train/X_test by using the StandardScaler() method from sklearn. Standard scalers main objective is to normalize all the data, scaling all of the values between -1 and 1. We conduct feature scaling because distance algorithms like K-nearest neighbor and support vector machines are most affected by the range of features. This is because they are using distance between data points to determine their similarity and make predictions. When two features have different scales, there is a chance that higher priority is given to the feature with a higher magnitude. Negatively impacting the performance of our machine learning algorithms, and obviously we do not want our distance-based algorithms to be biased towards one feature. Thus, we scale our data before employing a distance-based algorithm so that all features contribute equally to the result, avoiding biases and improving model performance.

Before employing our models for training and testing, we have defined two functions to cross-validate, record and print out our test set performances, along with the confusion matrixes displayed concisely.

```python
def evaluation(model, X_train_std, y_train, X_test_std, y_test, train=True, cv=5):
    if train:
        #Perform stratified k-fold cross-validation on training data
        skf = StratifiedKFold(n_splits=cv, shuffle=True, random_state=42)
        cv_scores_train = cross_val_score(model, X_train_std, y_train, cv=skf, scoring='accuracy')
        scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
        cv_results_train = cross_validate(model, X_train_std, y_train, cv=skf, scoring=scoring)

        #Print cross-validation results for training data
        print("")
        print("Stratified K-Fold Cross-Validation Results for Training Data (k={}):".format(cv))
        print("=====================================================")
        print("Mean Accuracy: {:.2f}%".format(cv_results_train['test_accuracy'].mean() * 100))
        print("Mean Precision: {:.2f}".format(cv_results_train['test_precision_macro'].mean()))
        print("Mean Recall: {:.2f}".format(cv_results_train['test_recall_macro'].mean()))
        print("Mean F1-score: {:.2f}".format(cv_results_train['test_f1_macro'].mean()))
        print("_____________________________________________________")
        print()

        #Train the model on the full training data
        model.fit(X_train_std, y_train)

        #Evaluate the model on training data
        pred_train = model.predict(X_train_std)
        print("Train Result:\n=====================================================")
        print("Accuracy Score: {:.2f}%".format(accuracy_score(y_train, pred_train) * 100))
        print("Precision: {:.2f}".format(precision_score(y_train, pred_train, average='macro')))
        print("Recall: {:.2f}".format(recall_score(y_train, pred_train, average='macro')))
        print("F1 Score: {:.2f}".format(f1_score(y_train, pred_train, average='macro')))
        print("_____________________________________________________")
        print("CLASSIFICATION REPORT:\n{}".format(classification_report(y_train, pred_train)))
        print("_____________________________________________________")
        #Plot confusion matrix for training data
        cm_train = confusion_matrix(y_train, pred_train)
        plot_confusion_matrix(cm_train)
        print("\n\n\n")

    else:
        #Perform stratified k-fold cross-validation on testing data
        skf = StratifiedKFold(n_splits=cv, shuffle=True, random_state=42)
        cv_scores_test = cross_val_score(model, X_test_std, y_test, cv=skf, scoring='accuracy')
        scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
        cv_results_test = cross_validate(model, X_test_std, y_test, cv=skf, scoring=scoring)

        #Print cross-validation results for testing data
        print("")
        print("Stratified K-Fold Cross-Validation Results for Testing Data (k={}):".format(cv))
        print("=====================================================")
        print("Mean Accuracy: {:.2f}%".format(cv_results_test['test_accuracy'].mean() * 100))
        print("Mean Precision: {:.2f}".format(cv_results_test['test_precision_macro'].mean()))
        print("Mean Recall: {:.2f}".format(cv_results_test['test_recall_macro'].mean()))
        print("Mean F1-score: {:.2f}".format(cv_results_test['test_f1_macro'].mean()))
        print("_____________________________________________________")
        print()

        #Evaluate the model on test data
        pred_test = model.predict(X_test_std)
        print("Test Result:\n=====================================================")
        print("Accuracy Score: {:.2f}%".format(accuracy_score(y_test, pred_test) * 100))
        print("Precision: {:.2f}".format(precision_score(y_test, pred_test, average='macro')))
        print("Recall: {:.2f}".format(recall_score(y_test, pred_test, average='macro')))
        print("F1 Score: {:.2f}".format(f1_score(y_test, pred_test, average='macro')))
        print("_____________________________________________________")
        print("CLASSIFICATION REPORT:\n{}".format(classification_report(y_test, pred_test)))
        print("_____________________________________________________")
        #Plot confusion matrix for test data
        cm_test = confusion_matrix(y_test, pred_test)
        plot_confusion_matrix(cm_test)
```

```
def plot_confusion_matrix(cm, labels=['No Stroke', 'Stroke'], fontsize=12):
    plt.figure(figsize=(5, 5))
    ax = plt.gca()
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)
    disp.plot(ax=ax, cmap='coolwarm')

    #Remove default annotations
    for text in ax.texts:
        text.set_visible(False)

    #Annotate the cells with TP, TN, FP, FN counts
    tn, fp, fn, tp = cm.ravel()
    total = tn + fp + fn + tp
    plt.text(0, 0, f'TN\n{tn}\n({tn / total:.2%})', ha='center', va='center', color='white', fontsize=fontsize)
    plt.text(1, 0, f'FP\n{fp}\n({fp / total:.2%})', ha='center', va='center', color='white', fontsize=fontsize)
    plt.text(0, 1, f'FN\n{fn}\n({fn / total:.2%})', ha='center', va='center', color='white', fontsize=fontsize)
    plt.text(1, 1, f'TP\n{tp}\n({tp / total:.2%})', ha='center', va='center', color='white', fontsize=fontsize)

    plt.title('Confusion Matrix')
    plt.xlabel('Predicted label')
    plt.ylabel('True label')
    plt.grid(False)
    plt.show()
```

The first function evaluation cross-validates and evaluates the performance metrics (accuracy, precision, recall, f1, confusion matrix) of a machine learning model on both training and testing sets.

It works like this:

- If train is set to True, it performs stratified k-fold cross-validation on the training data, computes various evaluation metrics such as accuracy, precision, recall, and F1-score, and prints the results. It then trains the model on the full training data, evaluates its performance, prints the cross-validation means, evaluation metrics, and displays a classification report and a confusion matrix.

- If train is set to False, it performs stratified k-fold cross-validation on the testing data, computes evaluation metrics, and prints the results. It then evaluates the model on the testing data, prints the evaluation metrics, and displays a classification report and a confusion matrix. Essentially repeating the train set to true process, although applied to test this time.

- The second function plot_confusion_matrix is a helper function that plots the confusion matrix for train and test with annotations for true negatives (TN), false positives (FP), false negatives (FN), and true positives (TP), along with their respective counts and percentages.

Overall, these functions provides a comprehensive evaluation of a machine learning model, including cross-validation results, evaluation metrics, classification reports, and plotting confusion matrixes for both training and testing data.

For cross-validation we have chosen the stratified k-fold cross-validation method as it is particularly effective on imbalanced datasets. Stratified k-fold cross-validation involves dividing the dataset into k folds, typically 5-10 (we chose 5), while ensuring that each fold maintains the same class distribution. Then, the model is trained and evaluated iteratively k times, with each fold serving as the test set once and the remaining folds as the training set. This approach is crucial for classification tasks with imbalanced class distributions, where some classes may be significantly underrepresented. By ensuring that each fold contains a proportional representation of all classes, stratified k-fold cross-validation provides more accurate performance estimates, especially in scenarios like ours with skewed class distributions.

DETAILED PREDICTIVE ANALYSIS

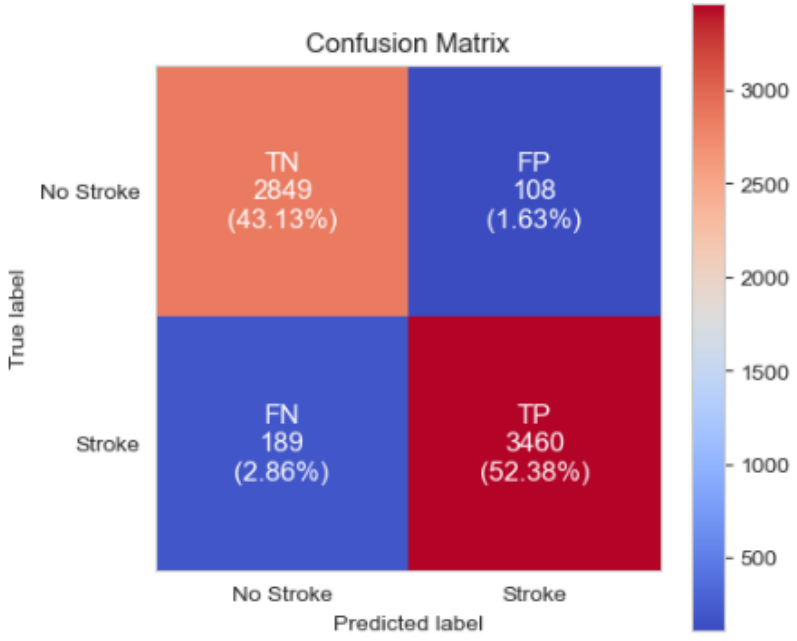
Modelling

Before we start the modelling phase, we want to emphasize the difference and importance of the training and testing set scores. To reiterate, the testing set metrics represent the model performance on predicting unseen data, simulating real-world scenarios on classifying stroke in patients — more importance placed on testing set. However, the training set metrics indicate the performance on how well the model fits the training data. Essentially, the scores indicate how accurately the model can predict the stroke class within the original dataset. Less importance on the training set since it does not represent real-world scenarios however, it serves as a good benchmark to see if the model adequately fits the training data; if the scores are subpar the model shouldn't even be tested on unseen data — because you are certain that the model is not trained well enough.

Now that we have emphasized the differences between the scoring of training and testing sets, we can go ahead and start with the predictive modelling phase.

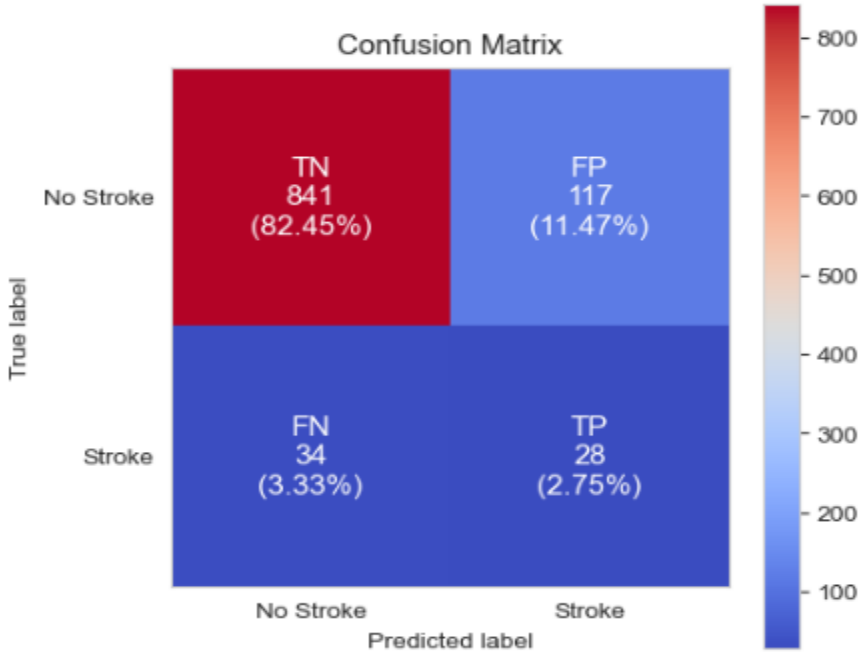
```
#Supervised Machine Learning Algorithm: Logistic Regression
lr = LogisticRegression()
#Fitting the model with training data
lr.fit(X_train_std, y_train)

#Evaluate the model on the training data
evaluation(lr, X_train_std, y_train, X_test_std, y_test, train=True)
#Evaluate the model on the testing data
evaluation(lr, X_train_std, y_train, X_test_std, y_test, train=False)
```

```
Stratified K-Fold Cross-Validation Results for Training Data (k=5):
======================================================
Mean Accuracy: 95.52%
Mean Precision: 0.95
Mean Recall: 0.96
Mean F1-score: 0.95
_____________________________________________________

Train Result:
======================================================
Accuracy Score: 95.50%
Precision: 0.95
Recall: 0.96
F1 Score: 0.95
_____________________________________________________
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.94      0.96      0.95      2957
           1       0.97      0.95      0.96      3649

    accuracy                           0.96      6606
   macro avg       0.95      0.96      0.95      6606
weighted avg       0.96      0.96      0.96      6606

_____________________________________________________
```



Logistic Regression model evaluation on training data: Clearly, the model is well-fit with these results. The stratified k-fold cross validation results line up with the train results, which indicates our model is working as intended and is consistent. The train results are excellent with 95.50% accuracy, 95% precision, 96% recall, and 95% f1 score. Subsuquently, the confusion matrix looks great, 2849 TN and 3640 TP. The model has performed outstandingly in classifying both no stroke and stroke correctly; with minimal classification errors at 108 FP and 189 FN, proportionally accounting for only 4.49% of the confusion matrix.

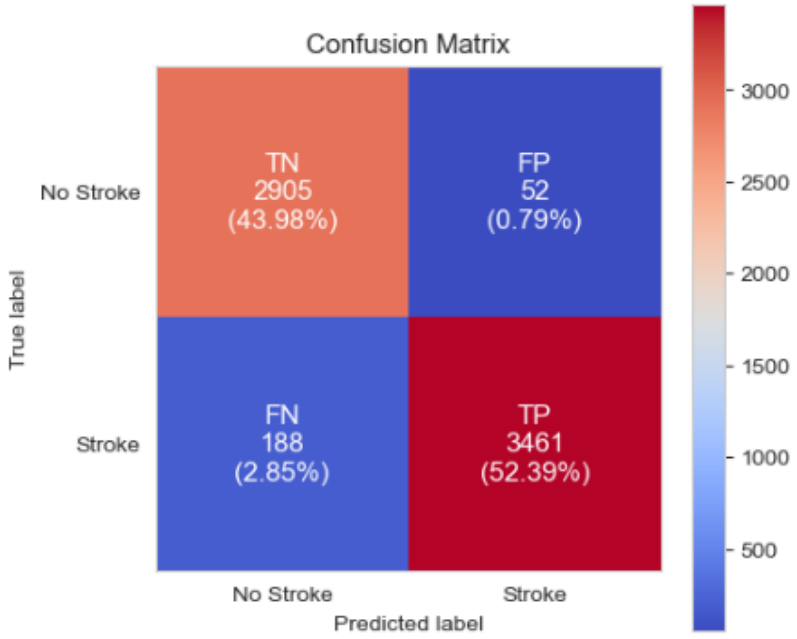
```
Stratified K-Fold Cross-Validation Results for Testing Data (k=5):
======================================================
Mean Accuracy: 93.82%
Mean Precision: 0.50
Mean Recall: 0.51
Mean F1-score: 0.50
------------------------------------------------------

Test Result:
======================================================
Accuracy Score: 85.20%
Precision: 0.58
Recall: 0.66
F1 Score: 0.59
------------------------------------------------------
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.96      0.88      0.92       958
           1       0.19      0.45      0.27        62

    accuracy                           0.85      1020
   macro avg       0.58      0.66      0.59      1020
weighted avg       0.91      0.85      0.88      1020

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



Logistic regression model evaluation on testing data: At first glance you could assume everything is fine by looking at the test accuracy, however there are issues present. Firstly, the stratified k-fold cross-validation results don't align with the test results, this is because of the poor results observed in the classification report and the data distribution in test. If we observe the 0 and 1 rows we can find the problem. 0 is no stroke and the model is excellent in predicting no stroke correctly: 96% precision, 88% recall, 92% F1-score, and 958 support (the number of samples in each class). In comparison, predicting stroke is in the ground: 19% precision, 45% recall, 27% F1-score, and 62 support; massive discrepancy in the support values. The actual test results do get balanced out somewhat because of the great no stroke classifications. 85.20% accuracy, 58% precision, 66% recall, and 59% f1-score. The reason precision, recall, F1 isn't as good/close to the result of accuracy is because these scores are measured by the macro average and not weighted average. Basically, macro-average provides an unbiased assessment across all classes, while weighted average considers class imbalance but may skew results towards the majority classes. This is why the results vary vastly between macro and weighted average in the classification report. The confusion matrix highlights the drastic differences in correctly classifying no stroke vs stroke — 841 TN & only 28 TP. Also, the model has a rate of 14.8% false classifications (11.47% FP & 3.33% FN).

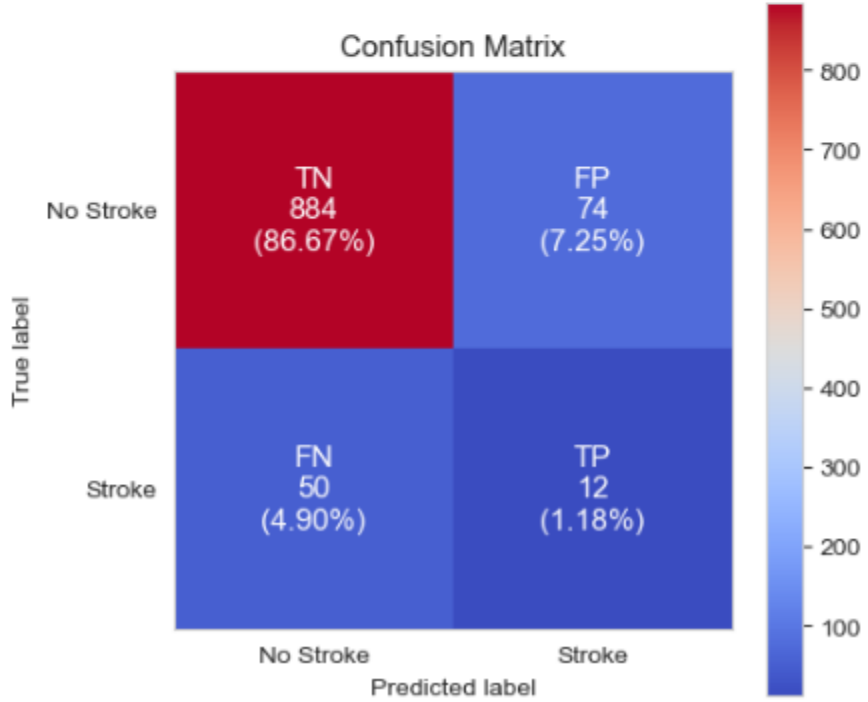
```python
#Supervised Machine Learning Algorithm: Support Vector Machine
svm = SVC()
#Fitting the model with training data
svm.fit(X_train_std, y_train)

#Evaluate the model on the training data
evaluation(svm, X_train_std, y_train, X_test_std, y_test, train=True)
#Evaluate the model on the testing data
evaluation(svm, X_train_std, y_train, X_test_std, y_test, train=False)
```

```
Stratified K-Fold Cross-Validation Results for Training Data (k=5):
=====================================================
Mean Accuracy: 95.85%
Mean Precision: 0.96
Mean Recall: 0.96
Mean F1-score: 0.96
_____________________________________________________

Train Result:
=====================================================
Accuracy Score: 96.37%
Precision: 0.96
Recall: 0.97
F1 Score: 0.96
_____________________________________________________
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.94      0.98      0.96      2957
           1       0.99      0.95      0.97      3649

    accuracy                           0.96      6606
   macro avg       0.96      0.97      0.96      6606
weighted avg       0.96      0.96      0.96      6606

_____________________________________________________
```



Support Vector Machine (SVM) model evaluation on training data: The stratified k-fold cross-validation outcomes are consistent with the training results, indicating reliability. The training metrics are impressive, boasting a 96.37% accuracy rate, 96% precision, 97% recall, and 96% F1-score. Additionally, the confusion matrix highlights how well the model performed, with 2905 TN and 3461 TP. The model demonstrates exceptional proficiency in accurately classifying both stroke and no-stroke instances, with minimal errors evident in 52 FP and 188 FN, Accounting for only 3.64% of the confusion matrix.

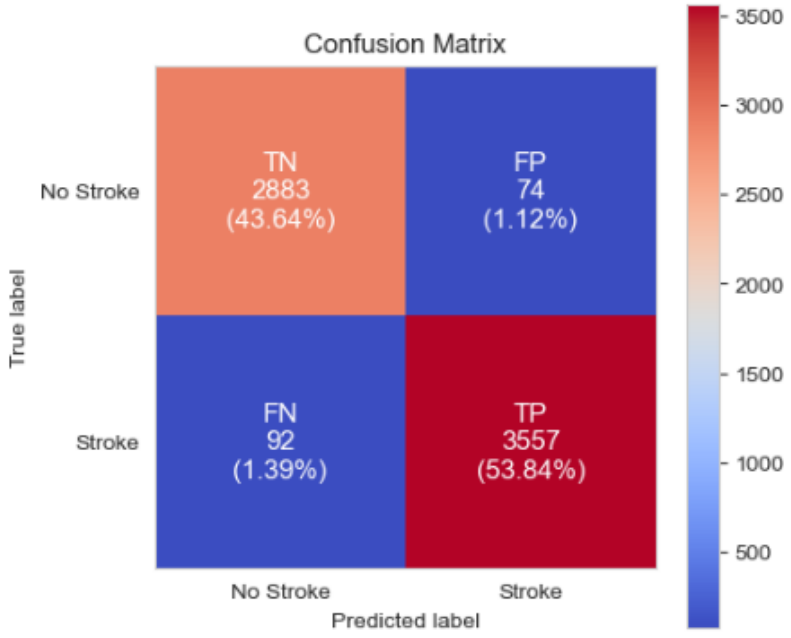
```
Stratified K-Fold Cross-Validation Results for Testing Data (k=5):
==================================================
Mean Accuracy: 93.92%
Mean Precision: 0.47
Mean Recall: 0.50
Mean F1-score: 0.48

--------------------------------------------------

Test Result:
==================================================
Accuracy Score: 87.84%
Precision: 0.54
Recall: 0.56
F1 Score: 0.55

--------------------------------------------------
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.95      0.92      0.93       958
           1       0.14      0.19      0.16        62

    accuracy                           0.88      1020
   macro avg       0.54      0.56      0.55      1020
weighted avg       0.90      0.88      0.89      1020

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



Support Vector Machine (SVM) model evaluation on testing data: Again, the stratified k-fold cross-validation results don't line up with the test results, this is because of the terrible prediction rate of stroke (1) observed in the classification report and the data distribution in test: 958 vs 62 support. Test metrics are decent at 87.84% accuracy, 54% precision, 56% recall, 55% F1-score. The classification report tells a story: The model is great at predicting no stroke (0), however, falls completely short in predicting stroke. Weighted averages of precision, recall and F1 are great in contrast of the macro averages, solely because of how well the model predicts no stroke. The confusion matrix paints a clearer picture of just how vastly different the model performs in predicting no stroke vs stroke: 884 TN and only 12 TP; with a rate of 12.15% incorrect classifications (7.25% FP & 4.90% FN). In contrast of the previous model logistic regression, we do see a 2% increase in testing accuracy.

```
#Supervised Machine Learning Algorithm: K-Nearest Neighbour(KNN)
knn = KNeighborsClassifier()
#Fitting the model with training data
knn.fit(X_train_std, y_train)

#Evaluate the model on the training data
evaluation(knn, X_train_std, y_train, X_test_std, y_test, train=True)
#Evaluate the model on the testing data
evaluation(knn, X_train_std, y_train, X_test_std, y_test, train=False)
```

```
Stratified K-Fold Cross-Validation Results for Training Data (k=5):
=====================================================
Mean Accuracy: 96.41%
Mean Precision: 0.96
Mean Recall: 0.96
Mean F1-score: 0.96
_____________________________________________________

Train Result:
=====================================================
Accuracy Score: 97.49%
Precision: 0.97
Recall: 0.97
F1 Score: 0.97
_____________________________________________________
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      2957
           1       0.98      0.97      0.98      3649

    accuracy                           0.97      6606
   macro avg       0.97      0.97      0.97      6606
weighted avg       0.97      0.97      0.97      6606

_____________________________________________________
```



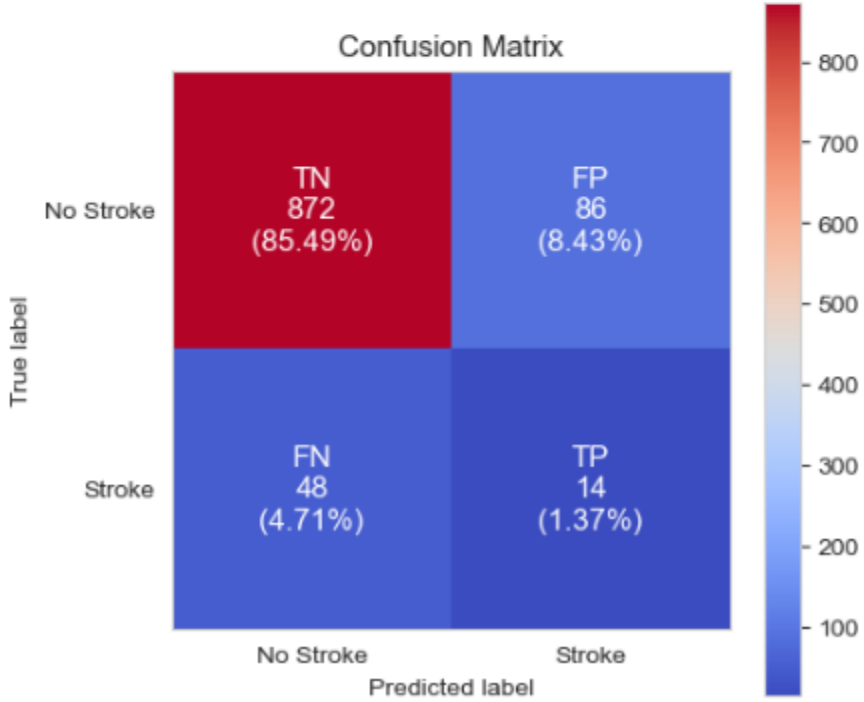
K-Nearest Neighbor (KNN) model evaluation on training data: The results from stratified k-fold cross-validation align consistently with the training outcomes, showcasing the consistency of the model. The training metrics are impressive, showing a high accuracy rate of 97.49%, along with precision, recall, and F1-score all at 97%. The confusion matrix further illustrates the models effectiveness in accurately classifying instances, with 2883 TN and 3557 TP.
It shows exceptional performance in distinguishing no stroke and stroke classifications, with only a small number of misclassifications: 74 FP and 92 FN, making up just 2.51% of the confusion matrix.

```
Stratified K-Fold Cross-Validation Results for Testing Data (k=5):
=====================================================
Mean Accuracy: 93.43%
Mean Precision: 0.47
Mean Recall: 0.50
Mean F1-score: 0.48
```

```
Test Result:
=====================================================
Accuracy Score: 86.86%
Precision: 0.54
Recall: 0.57
F1 Score: 0.55
```

CLASSIFICATION REPORT:

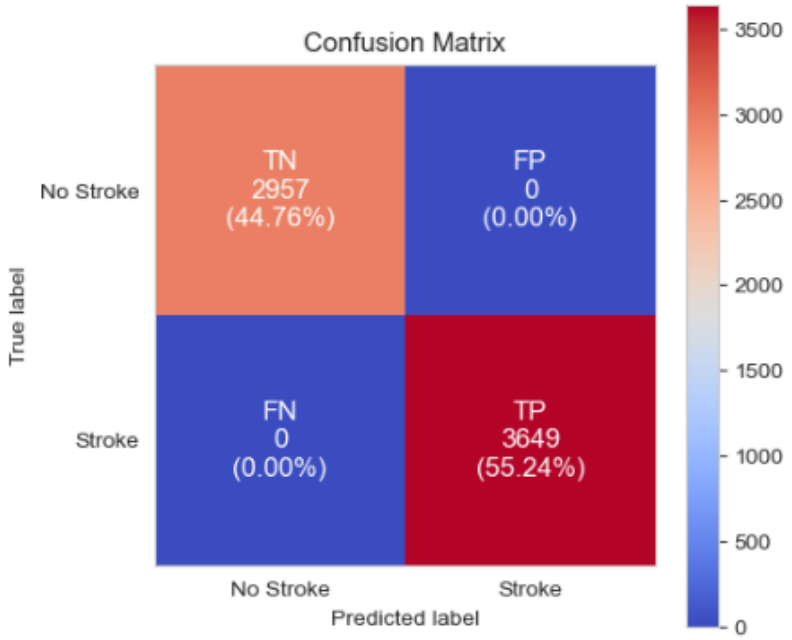
	precision	recall	f1-score	support
0	0.95	0.91	0.93	958
1	0.14	0.23	0.17	62
accuracy			0.87	1020
macro avg	0.54	0.57	0.55	1020
weighted avg	0.90	0.87	0.88	1020



K-Nearest Neighbor (KNN) model evaluation on testing data: Once more, the results from stratified k-fold cross-validation diverge from those of the test set, primarily due to the awful performance in predicting stroke cases observed in the classification report, as well as the stark data distribution in the test set: 958 no stroke & 62 stroke support count. Despite the test metrics reflecting good performance with an 86.86% accuracy rate, the precision, recall, and F1-score are subpar in comparison at 54%, 57%, and 55% respectively. The classification report showcases this discrepancy, highlighting the models proficiency in classifying no stroke while revealing significant shortcomings in classifying stroke. Although the weighted averages of precision, recall, and F1-score present a better picture, this is primarily caused by the models strong performance in predicting no stroke. An examination of the confusion matrix further displays the significant disparity in the models predictive capabilities between no stroke and stroke cases: 872 TN compared to only 14 TP, accompanied by 86 FP and 48 FN, resulting in 13.14% misclassifications.

```
#Supervised Machine Learning Algorithm: Decision Tree
dt = DecisionTreeClassifier()
#Fitting the model with training data
dt.fit(X_train_std, y_train)

#Evaluate the model on the training data
evaluation(dt, X_train_std, y_train, X_test_std, y_test, train=True)
#Evaluate the model on the testing data
evaluation(dt, X_train_std, y_train, X_test_std, y_test, train=False)
```

```
Stratified K-Fold Cross-Validation Results for Training Data (k=5):
======================================================
Mean Accuracy: 95.84%
Mean Precision: 0.96
Mean Recall: 0.96
Mean F1-score: 0.96
_____________________________________________________

Train Result:
======================================================
Accuracy Score: 100.00%
Precision: 1.00
Recall: 1.00
F1 Score: 1.00
_____________________________________________________
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2957
           1       1.00      1.00      1.00      3649

    accuracy                           1.00      6606
   macro avg       1.00      1.00      1.00      6606
weighted avg       1.00      1.00      1.00      6606

_____________________________________________________
```



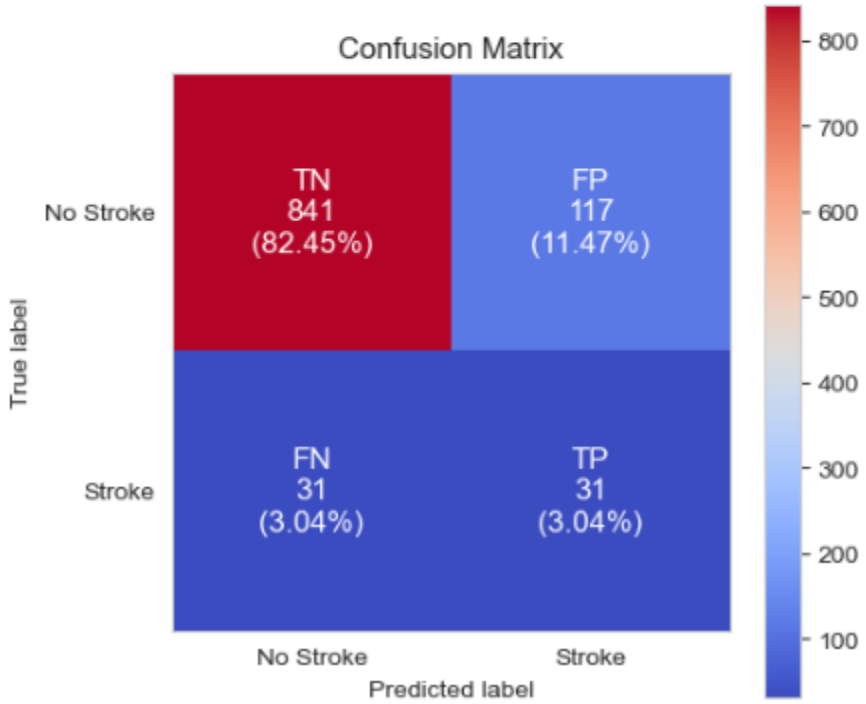
Decision Tree Classifier model evaluation on training data: The results from stratified k-fold cross-validation fall in line with the training metrics, indicating the reliability of the model. The training results are perfect, displaying a perfect accuracy rate of 100.00%, along with precision, recall, and F1-score all at 100%. The confusion matrix further highlights the models perfection in accurately classifying, with 2957 TN and 3649 TP. It demonstrates outstanding performance in distinguishing between classifications of no stroke and stroke, with no misclassifications observed: 0 FP and 0 FN.

```
Stratified K-Fold Cross-Validation Results for Testing Data (k=5)
=====================================================
Mean Accuracy: 89.02%
Mean Precision: 0.55
Mean Recall: 0.58
Mean F1-score: 0.56
```

```
Test Result:
=====================================================
Accuracy Score: 85.49%
Precision: 0.59
Recall: 0.69
F1 Score: 0.61
```

CLASSIFICATION REPORT:

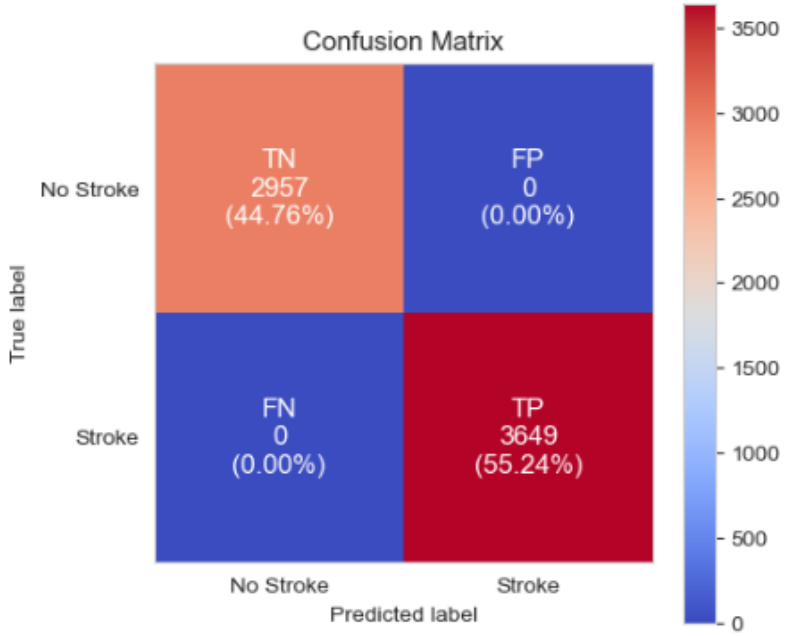
	precision	recall	f1-score	support
0	0.96	0.88	0.92	958
1	0.21	0.50	0.30	62
accuracy			0.85	1020
macro avg	0.59	0.69	0.61	1020
weighted avg	0.92	0.85	0.88	1020



Decision Tree Classifier model evaluation on testing data: The results from stratified k-fold cross-validation show some deviation from those of the test set, largely attributed to the model's substandard performance in predicting stroke cases, as indicated by the classification report. Additionally, the imbalance in data distribution within the test set, with 958 instances of no stroke and only 62 instances of stroke, contributes to this disparity. Despite achieving a respectable accuracy rate of 85.49%, the precision, recall, and F1-score are comparatively lower, standing at 59%, 69%, and 61% respectively. The classification report contrasts the model's proficiency in classifying no stroke cases but reveals notable flaws in classifying stroke cases. The weighted averages of precision, recall, and F1-score in comparison are way better, this is because it is predominantly driven by the models excellent performance in predicting no stroke. A more detailed analysis of the confusion matrix highlights the substantial difference in the model's ability to predict between instances of no stroke and stroke: with 841 TN contrasted with merely 31 TP; along with 117 FP and 31 FN. This leads to a total of 148 misclassifications, representing a ratio of 14.51%.

```
#Supervised Machine Learning Algorithm: Random Forest Classifier
rfc = RandomForestClassifier()
#Fitting the model with training data
rfc.fit(X_train_std, y_train)

#Evaluate the model on the training data
evaluation(rfc, X_train_std, y_train, X_test_std, y_test, train=True)
#Evaluate the model on the testing data
evaluation(rfc, X_train_std, y_train, X_test_std, y_test, train=False)
```

```
Stratified K-Fold Cross-Validation Results for Training Data (k=5):
======================================================
Mean Accuracy: 97.65%
Mean Precision: 0.98
Mean Recall: 0.98
Mean F1-score: 0.98
_____________________________________________________

Train Result:
======================================================
Accuracy Score: 100.00%
Precision: 1.00
Recall: 1.00
F1 Score: 1.00
_____________________________________________________
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2957
           1       1.00      1.00      1.00      3649

    accuracy                           1.00      6606
   macro avg       1.00      1.00      1.00      6606
weighted avg       1.00      1.00      1.00      6606

_____________________________________________________
```



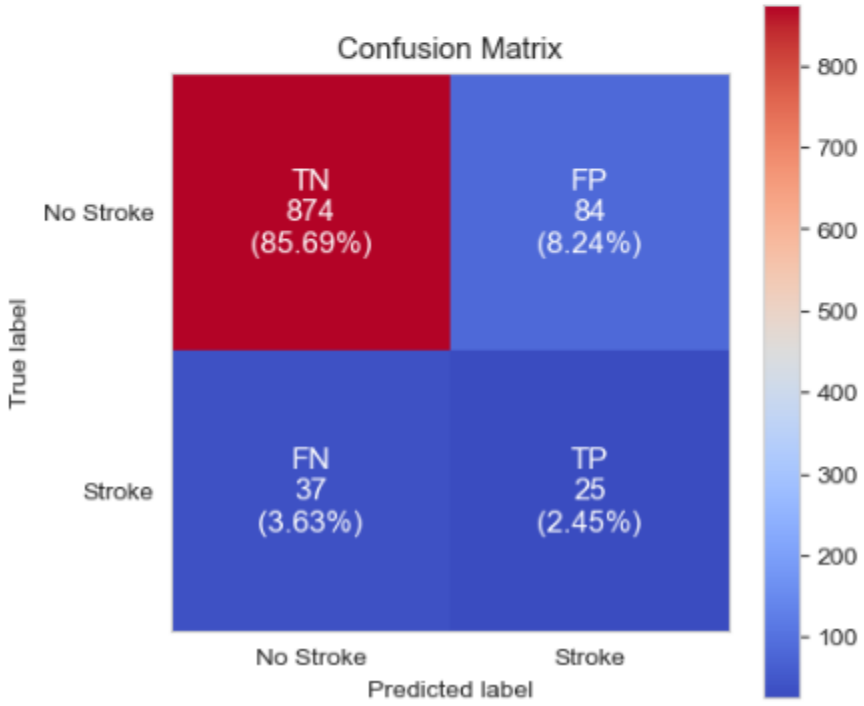
Random Forest Classifier model evaluation on training data: The outcomes from stratified k-fold cross-validation coincide with the training metrics, affirming the model's reliability. The training results are flawless, with a 100.00% accuracy rate, alongside precision, recall, and F1-score all reaching 100%. Moreover, the confusion matrix spotlights the model's impeccable performance in classification, with 2957 TN and 3649 true positives TP. It exhibits exceptional proficiency in distinguishing between no stroke and stroke instances, with no misclassifications present: 0 FP and 0 FN — an impeccable model.

```
Stratified K-Fold Cross-Validation Results for Testing Data (k=5):
======================================================
Mean Accuracy: 93.73%
Mean Precision: 0.52
Mean Recall: 0.51
Mean F1-score: 0.50
```

```
Test Result:
======================================================
Accuracy Score: 88.14%
Precision: 0.59
Recall: 0.66
F1 Score: 0.61
```

CLASSIFICATION REPORT:

	precision	recall	f1-score	support
0	0.96	0.91	0.94	958
1	0.23	0.40	0.29	62
accuracy			0.88	1020
macro avg	0.59	0.66	0.61	1020
weighted avg	0.92	0.88	0.90	1020



Random Forest Classifier model evaluation on testing data:
The results from stratified k-fold cross-validation shows variance compared to the test results; due to the models inadequate performance in classifying stroke, as shown in the classification report. Along with the imbalance in data distribution within the test set, with 958 vs 62 no stroke and stroke support samples. The test results are the best yet in terms of accuracy with a rate of 88.14%, the precision, recall, and F1-score are lower in contrast at 59%, 66%, and 61%. While the model is really good at classifying no stroke correctly, it's the complete opposite at classifying stroke. The weighted averages of precision, recall, and F1-score are much better, driven mainly by the model's significant performance in predicting no stroke. A thorough look at the confusion matrix signifies the drastic difference in the model's predictive ability on instances of no stroke and stroke: 874 TN compared to only 25 TP, accompanied by 84 FP and 37 FN. This results in a total of 121 misclassifications, or proportionally speaking equal to 11.87% of the confusion matrix.

Now we're going to run some code to summarize the results of the modelling phase. However, we are only going to be focusing on test results, as emphasized before, the test results are far more important as they represent real-world scenarios were we are conducting our models to classify stroke or no stroke on patients in unseen data.

```
#Calculate scores for each model
test_score_lr = round(accuracy_score(y_test, lr.predict(X_test_std)) * 100, 2)
test_score_svm = round(accuracy_score(y_test, svm.predict(X_test_std)) * 100, 2)
test_score_knn = round(accuracy_score(y_test, knn.predict(X_test_std)) * 100, 2)
test_score_dt = round(accuracy_score(y_test, dt.predict(X_test_std)) * 100, 2)
test_score_rfc = round(accuracy_score(y_test, rfc.predict(X_test_std)) * 100, 2)

test_precision_lr = round(precision_score(y_test, lr.predict(X_test_std), average='macro') * 100, 2)
test_precision_svm = round(precision_score(y_test, svm.predict(X_test_std), average='macro') * 100, 2)
test_precision_knn = round(precision_score(y_test, knn.predict(X_test_std), average='macro') * 100, 2)
test_precision_dt = round(precision_score(y_test, dt.predict(X_test_std), average='macro') * 100, 2)
test_precision_rfc = round(precision_score(y_test, rfc.predict(X_test_std), average='macro') * 100, 2)

test_recall_lr = round(recall_score(y_test, lr.predict(X_test_std), average='macro') * 100, 2)
test_recall_svm = round(recall_score(y_test, svm.predict(X_test_std), average='macro') * 100, 2)
test_recall_knn = round(recall_score(y_test, knn.predict(X_test_std), average='macro') * 100, 2)
test_recall_dt = round(recall_score(y_test, dt.predict(X_test_std), average='macro') * 100, 2)
test_recall_rfc = round(recall_score(y_test, rfc.predict(X_test_std), average='macro') * 100, 2)

test_f1_lr = round(f1_score(y_test, lr.predict(X_test_std), average='macro') * 100, 2)
test_f1_svm = round(f1_score(y_test, svm.predict(X_test_std), average='macro') * 100, 2)
test_f1_knn = round(f1_score(y_test, knn.predict(X_test_std), average='macro') * 100, 2)
test_f1_dt = round(f1_score(y_test, dt.predict(X_test_std), average='macro') * 100, 2)
test_f1_rfc = round(f1_score(y_test, rfc.predict(X_test_std), average='macro') * 100, 2)

##Variable to display model results
models = {
    'Test Accuracy': [test_score_lr, test_score_svm, test_score_knn, test_score_dt, test_score_rfc],
    'Test Precision': [test_precision_lr, test_precision_svm, test_precision_knn, test_precision_dt, test_precision_rfc],
    'Test Recall': [test_recall_lr, test_recall_svm, test_recall_knn, test_recall_dt, test_recall_rfc],
    'Test F1 Score': [test_f1_lr, test_f1_svm, test_f1_knn, test_f1_dt, test_f1_rfc]
}

models = pd.DataFrame(models, index=['Logistic Regression', 'Support Vector Machine', 'K-Nearest Neighbour',
                                     'Decision Tree Classifier', 'Random Forest Classifier'])

#Display results
models.head()
```

	Test Accuracy	Test Precision	Test Recall	Test F1 Score
Logistic Regression	85.20	57.71	66.47	59.41
Support Vector Machine	87.84	54.30	55.82	54.83
K-Nearest Neighbour	86.86	54.39	56.80	55.07
Decision Tree Classifier	85.49	58.70	68.89	60.72
Random Forest Classifier	88.14	59.44	65.78	61.38

By observing the table above, we can ascertain that random forest classifier is currently the best performing model before hyperparameter tuning at 88.14% accuracy. Logistic Regression is currently the worst performing model at 85.20% accuracy. Random forest classifier is only ahead of SVM by 0.30%, results could possibly change after hyperparameter tuning.

Hyperparameter Tuning

We will conduct hyperparameter tuning on our models with GridSearchCV. GridSearchCV operates by searching a predefined grid of hyperparameters you have specified, assessing each and every combination through cross-validation (5 folds), and ultimately selecting the configuration that yields the highest performance based on the chosen metric, in our case: accuracy. Essentially, it streamlines the process of finding the optimal model configuration, eliminating the need for manual experimentation, making it extremely convenient.

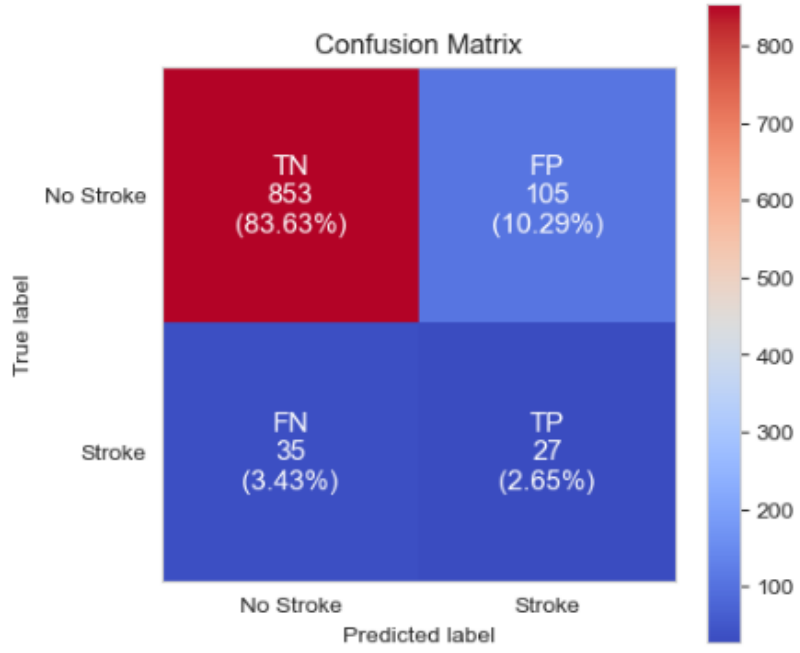
```
#Define the hyperparameter grid
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'penalty': ['l2'],
    'solver': ['liblinear', 'lbfgs', 'sag', 'saga', 'newton-cg', 'newton-cholesky'],
    'class_weight': ['balanced', None]
}

#Perform grid search with cross-validation
grid_search = GridSearchCV(LogisticRegression(max_iter=1500), param_grid, cv=5, scoring='accuracy')
#Fit grid search with training data
grid_search.fit(X_train_std, y_train)

#Get the best hyperparameters
best_params = grid_search.best_params_

#Print the best hyperparameters
print("Best Hyperparameters:")
print(best_params)
```

```
Best Hyperparameters:
{'C': 0.1, 'class_weight': 'balanced', 'penalty': 'l2', 'solver': 'lbfgs'}
```

Performing hyperparameter tuning with GridSearchCV for logistic regression. We will solely be focusing on model scores during hyperparameter tuning.

```python
#Tuned logistic regression model with the best hyperparameters
tuned_lr = LogisticRegression(**best_params, max_iter=1500)

#Fit the model to your training data
tuned_lr.fit(X_train_std, y_train)

#Evaluate the model on the testing data
evaluation(tuned_lr, X_train_std, y_train, X_test_std, y_test, train=False)
```

```
Stratified K-Fold Cross-Validation Results for Testing Data (k=5):
==================================================
Mean Accuracy: 72.25%
Mean Precision: 0.56
Mean Recall: 0.73
Mean F1-score: 0.54
--------------------------------------------------

Test Result:
==================================================
Accuracy Score: 86.27%
Precision: 0.58
Recall: 0.66
F1 Score: 0.60
--------------------------------------------------
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.96      0.89      0.92       958
           1       0.20      0.44      0.28        62

    accuracy                           0.86      1020
   macro avg       0.58      0.66      0.60      1020
weighted avg       0.91      0.86      0.88      1020

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



Hyperparameter Tuned Logistic regression model evaluation on testing data: Overall, there is a slight improvement in results over the untuned logistic regression model:

Tuned Logistic Regression model:

- 86.27% accuracy, 58% precision, 66% recall, and 60% F1-score. 853 TN, 27 TP, 105 FP, and 35 FN.

Untuned Logistic Regression model:

- 85.20% accuracy, 58% precision, 66% recall, and 59% F1-score. 841 TN, 28 TP, 117 FP, and 34 FN.

Our tuned logistic regression model has less errors (misclassifications), a higher number of correct predictions, and has better accuracy and f1 score over the untuned logistic regression model.

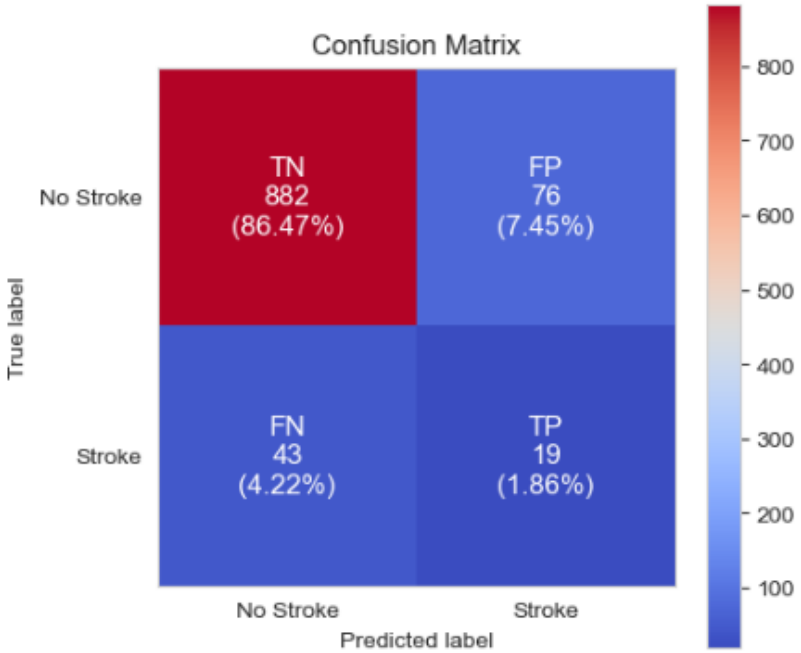
```
#Define the hyperparameter grid
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
    'gamma': [0.1, 'scale', 'auto'],
    'class_weight': ['balanced', None]
}

#Perform grid search with cross-validation
grid_search = GridSearchCV(SVC(), param_grid, cv=5, scoring='accuracy')
#Fit grid search with training data
grid_search.fit(X_train_std, y_train)

#Get the best hyperparameters
best_params = grid_search.best_params_

#Print the best hyperparameters
print("Best Hyperparameters:")
print(best_params)
```

```
Best Hyperparameters:
{'C': 10, 'class_weight': 'balanced', 'gamma': 0.1, 'kernel': 'rbf'}
```

Performing hyperparameter tuning with GridSearchCV for SVM.

```
#Tuned support vector machine model with the best hyperparameters
tuned_svm = SVC(**best_params)

#Fit the model with training data
tuned_svm.fit(X_train_std, y_train)

#Evaluate the model on the testing data
evaluation(tuned_svm, X_train_std, y_train, X_test_std, y_test, train=False)
```

```
Stratified K-Fold Cross-Validation Results for Testing Data (k=5):
==================================================
Mean Accuracy: 78.92%
Mean Precision: 0.50
Mean Recall: 0.50
Mean F1-score: 0.48
--------------------------------------------------

Test Result:
==================================================
Accuracy Score: 88.33%
Precision: 0.58
Recall: 0.61
F1 Score: 0.59
--------------------------------------------------
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.95      0.92      0.94       958
           1       0.20      0.31      0.24        62

    accuracy                           0.88      1020
   macro avg       0.58      0.61      0.59      1020
weighted avg       0.91      0.88      0.89      1020

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



Hyperparameter Tuned Support Vector Machine (SVM) model evaluation on testing data: Again, there is a small improvement in the results over the untuned SVM model:

Tuned SVM model:

- 88.33% accuracy, 58% precision, 61% recall, and 59% F1-score. 882 TN, 19 TP, 76 FP, and 43 FN.

Untuned SVM model:

- 87.84% accuracy, 54% precision, 56% recall, and 55% F1-score. 884 TN, 12 TP, 74 FP, and 50 FN.

Our tuned SVM model has less misclassifications, a bigger number of correct classifications, and contains better accuracy, precision, recall, and f1 score over the untuned SVM model.

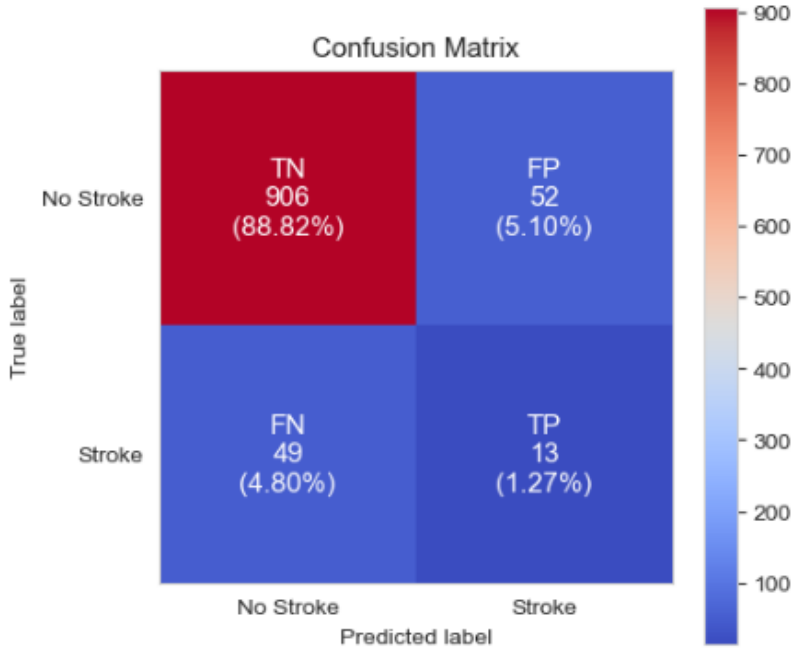
```
#Define the hyperparameter grid
param_grid = {
    'n_neighbors': list(range(1, 31)),  #Range of neighbors from 1 to 30
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski', 'hamming']

}

#Perform grid search with cross-validation
grid_search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5, scoring='accuracy')
#Fit grid search with training data
grid_search.fit(X_train_std, y_train)

#Get the best hyperparameters
best_params = grid_search.best_params_

#Print the best hyperparameters
print("Best Hyperparameters:")
print(best_params)
```

```
Best Hyperparameters:
{'metric': 'manhattan', 'n_neighbors': 2, 'weights': 'uniform'}
```

Performing hyperparameter tuning with GridSearchCV on KNN.

```
#Tuned K-Nearest Neighbor (KNN) with the best hyperparameters
tuned_knn = KNeighborsClassifier(**best_params)

#Fit the model with training data
tuned_knn.fit(X_train_std, y_train)

#Evaluate the model on the testing data
evaluation(tuned_knn, X_train_std, y_train, X_test_std, y_test, train=False)
```

```
Stratified K-Fold Cross-Validation Results for Testing Data (k=5):
==================================================
Mean Accuracy: 93.33%
Mean Precision: 0.47
Mean Recall: 0.50
Mean F1-score: 0.48
--------------------------------------------------

Test Result:
==================================================
Accuracy Score: 90.10%
Precision: 0.57
Recall: 0.58
F1 Score: 0.58
--------------------------------------------------
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.95      0.95      0.95       958
           1       0.20      0.21      0.20        62

    accuracy                           0.90      1020
   macro avg       0.57      0.58      0.58      1020
weighted avg       0.90      0.90      0.90      1020

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



Hyperparameter Tuned K-Nearest Neighbor (KNN) model evaluation on testing data: Currently, this is the most substantial improvement in tuning models so far. The test results:

Tuned KNN model:

- 90.10% accuracy, 57% precision, 58% recall, and 58% F1-score. 906 TN, 13 TP, 52 FP, and 49 FN.

Untuned KNN model:

- 86.86% accuracy, 54% precision, 57% recall, and 55% F1-score. 872 TN, 14 TP, 86 FP, and 48 FN.

As previously iterated, this is currently the biggest improvement in terms of accuracy: a 3.24% increase. This optimized KNN model demonstrates reduced misclassifications, increased correct classifications, and improved accuracy, precision, and F1-score compared to the original KNN model.

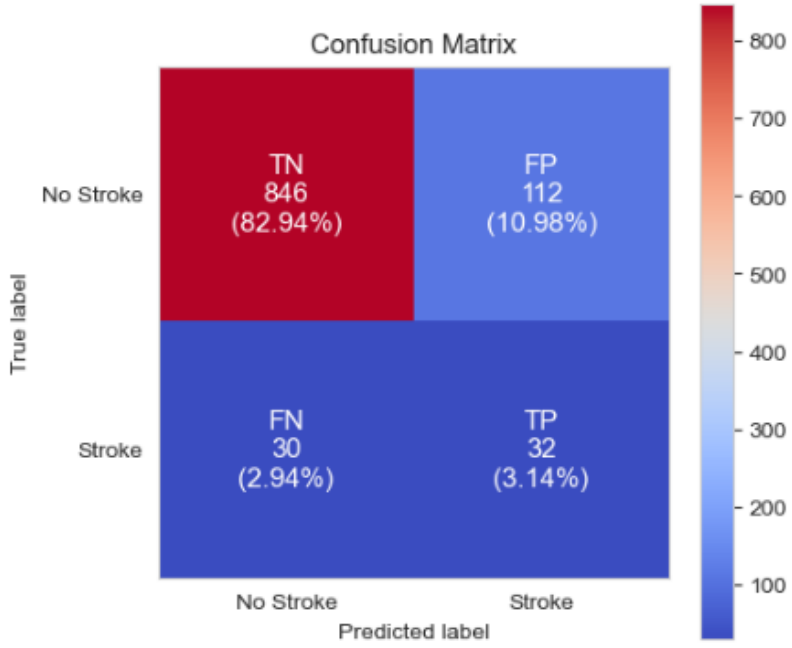
```
#Define the hyperparameter grid
param_grid = {
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 4, 8],
    'max_features': ['auto', 'sqrt', 'log2', None],
    'class_weight': ['balanced', None]
}

#Perform grid search with cross-validation
grid_search = GridSearchCV(DecisionTreeClassifier(), param_grid, cv=5, scoring='accuracy')
#Fit grid search with training data
grid_search.fit(X_train_std, y_train)

#Get the best hyperparameters
best_params = grid_search.best_params_

#Print the best hyperparameters
print("Best Hyperparameters:")
print(best_params)
```

```
Best Hyperparameters:
{'class_weight': None, 'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2}
```

Performing hyperparameter tuning on the decision tree classifier with GridSearchCV.

```
#Tuned Decision Tree Classifier with the best hyperparameters
tuned_dt = DecisionTreeClassifier(**best_params)

#Fit the model with training data
tuned_dt.fit(X_train_std, y_train)

#Evaluate the model on the testing data
evaluation(tuned_dt, X_train_std, y_train, X_test_std, y_test, train=False)
```

```
Stratified K-Fold Cross-Validation Results for Testing Data (k=5):
==================================================
Mean Accuracy: 89.61%
Mean Precision: 0.56
Mean Recall: 0.57
Mean F1-score: 0.56
--------------------------------------------------

Test Result:
==================================================
Accuracy Score: 86.08%
Precision: 0.59
Recall: 0.70
F1 Score: 0.62
--------------------------------------------------
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.97      0.88      0.92       958
           1       0.22      0.52      0.31        62

    accuracy                           0.86      1020
   macro avg       0.59      0.70      0.62      1020
weighted avg       0.92      0.86      0.89      1020

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



Hyperparameter Tuned Decision Tree Classifier model evaluation on testing data: Once again, there is a marginal improvement in the results over the untuned decision tree model:

Tuned Decision Tree model:

- 86.08% accuracy, 59% precision, 70% recall, and 62% F1-score. 846 TN, 32 TP, 112 FP, and 30 FN.

Untuned Decision Tree model:

- 85.49% accuracy, 59% precision, 69% recall, and 61% F1-score. 841 TN, 31 TP, 117 FP, and 31 FN.

The optimized decision tree model exhibits fewer misclassifications, a higher count of accurate classifications, and establishes slightly superior accuracy, recall, and F1 score compared to the original, untuned decision tree model.

```python
#Define the hyperparameter grid
param_grid = {
    'n_estimators': [100, 200, 300, 400],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 4, 8],
    'max_features': ['auto', 'sqrt', 'log2']
}

#Perform grid search with cross-validation
grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=5, scoring='accuracy')
#Fit grid search with training data
grid_search.fit(X_train_std, y_train)

#Get the best hyperparameters
best_params = grid_search.best_params_

#Print the best hyperparameters
print("Best Hyperparameters:")
print(best_params)
```

```
Best Hyperparameters:
{'max_depth': 20, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 400}
```

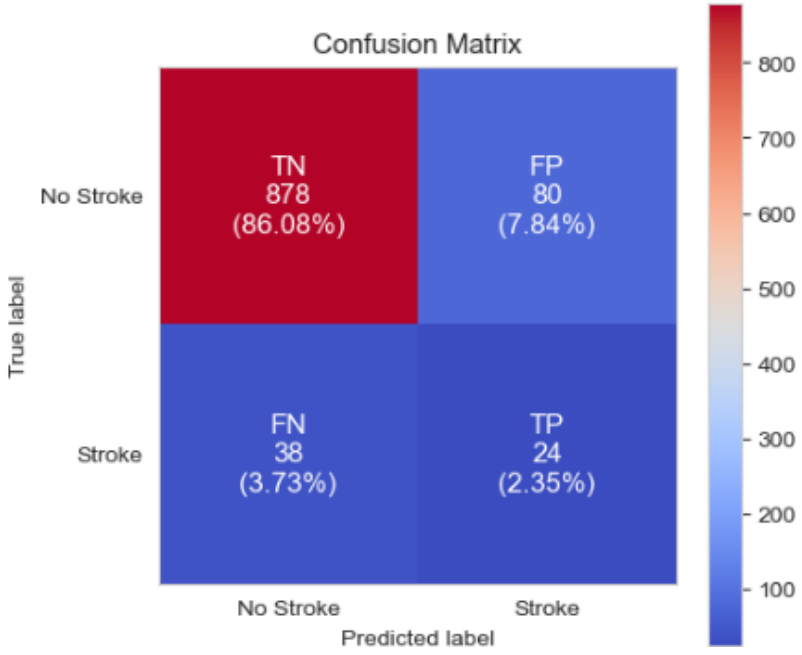
Performing hyperparameter tuning with GridSearchCV on Random Forest Classifier.

```
#Tuned Random Forest Classifier with the best hyperparameters
tuned_rfc = RandomForestClassifier(**best_params)

#Fit the model with training data
tuned_rfc.fit(X_train_std, y_train)

#Evaluate the model on the testing data
evaluation(tuned_rfc, X_train_std, y_train, X_test_std, y_test, train=False)
```

```
Stratified K-Fold Cross-Validation Results for Testing Data (k=5):
==================================================
Mean Accuracy: 93.73%
Mean Precision: 0.57
Mean Recall: 0.51
Mean F1-score: 0.50
```

```
Test Result:
==================================================
Accuracy Score: 88.43%
Precision: 0.59
Recall: 0.65
F1 Score: 0.61
```

```
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.96      0.92      0.94       958
           1       0.23      0.39      0.29        62

    accuracy                           0.88      1020
   macro avg       0.59      0.65      0.61      1020
weighted avg       0.91      0.88      0.90      1020
```



Hyperparameter Tuned Random Forest Classifier model evaluation on testing data: Once more, there is a slight enhancement observed in the outcomes compared to the untuned random forest classifier model.

Tuned Random Forest Classifier model:

- 88.43% accuracy, 59% precision, 65% recall, and 61% F1-score. 878 TN, 24 TP, 80 FP, and 38 FN.

Untuned Random Forest Classifier model:

- 88.14% accuracy, 59% precision, 66% recall, and 61% F1-score. 874 TN, 25 TP, 84 FP, and 37 FN.

The refined random forest classifier model showcases a reduction in misclassifications, a slightly increased number of correct classifications, and marginally better accuracy when contrasted with the initial, untuned decision tree model.

Run code to display hyperparameter tuned results in a tabular form.

```
#Calculate scores for each tuned model
test_score_tuned_lr = round(accuracy_score(y_test, tuned_lr.predict(X_test_std)) * 100, 2)
test_score_tuned_svm = round(accuracy_score(y_test, tuned_svm.predict(X_test_std)) * 100, 2)
test_score_tuned_knn = round(accuracy_score(y_test, tuned_knn.predict(X_test_std)) * 100, 2)
test_score_tuned_dt = round(accuracy_score(y_test, tuned_dt.predict(X_test_std)) * 100, 2)
test_score_tuned_rfc = round(accuracy_score(y_test, tuned_rfc.predict(X_test_std)) * 100, 2)

test_precision_tuned_lr = round(precision_score(y_test, tuned_lr.predict(X_test_std), average='macro') * 100, 2)
test_precision_tuned_svm = round(precision_score(y_test, tuned_svm.predict(X_test_std), average='macro') * 100, 2)
test_precision_tuned_knn = round(precision_score(y_test, tuned_knn.predict(X_test_std), average='macro') * 100, 2)
test_precision_tuned_dt = round(precision_score(y_test, tuned_dt.predict(X_test_std), average='macro') * 100, 2)
test_precision_tuned_rfc = round(precision_score(y_test, tuned_rfc.predict(X_test_std), average='macro') * 100, 2)

test_recall_tuned_lr = round(recall_score(y_test, tuned_lr.predict(X_test_std), average='macro') * 100, 2)
test_recall_tuned_svm = round(recall_score(y_test, tuned_svm.predict(X_test_std), average='macro') * 100, 2)
test_recall_tuned_knn = round(recall_score(y_test, tuned_knn.predict(X_test_std), average='macro') * 100, 2)
test_recall_tuned_dt = round(recall_score(y_test, tuned_dt.predict(X_test_std), average='macro') * 100, 2)
test_recall_tuned_rfc = round(recall_score(y_test, tuned_rfc.predict(X_test_std), average='macro') * 100, 2)

test_f1_tuned_lr = round(f1_score(y_test, tuned_lr.predict(X_test_std), average='macro') * 100, 2)
test_f1_tuned_svm = round(f1_score(y_test, tuned_svm.predict(X_test_std), average='macro') * 100, 2)
test_f1_tuned_knn = round(f1_score(y_test, tuned_knn.predict(X_test_std), average='macro') * 100, 2)
test_f1_tuned_dt = round(f1_score(y_test, tuned_dt.predict(X_test_std), average='macro') * 100, 2)
test_f1_tuned_rfc = round(f1_score(y_test, tuned_rfc.predict(X_test_std), average='macro') * 100, 2)

#Create a DataFrame for tuned model results
tuned_results = {
    'Test Accuracy': [test_score_tuned_lr, test_score_tuned_svm, test_score_tuned_knn, test_score_tuned_dt, test_score_tuned_rfc]
    'Test Precision': [test_precision_tuned_lr, test_precision_tuned_svm, test_precision_tuned_knn, test_precision_tuned_dt, test
    'Test Recall': [test_recall_tuned_lr, test_recall_tuned_svm, test_recall_tuned_knn, test_recall_tuned_dt, test_recall_tuned_r
    'Test F1 Score': [test_f1_tuned_lr, test_f1_tuned_svm, test_f1_tuned_knn, test_f1_tuned_dt, test_f1_tuned_rfc]
}

#Index for model names
model_names = ['Tuned Logistic Regression', 'Tuned Support Vector Machine', 'Tuned K-Nearest Neighbour',
                                'Tuned Decision Tree Classifier', 'Tuned Random Forest Classifier']

#Create the DataFrame
tuned_models = pd.DataFrame(tuned_results, index=model_names)

#Define colors for each row
colors = [
    'skyblue',
    'lightgreen',
    'lightcoral',
    'lightcyan',
    'lightsalmon'
]

#Apply background colors to the DataFrame
styled_tuned_models = tuned_models.style.format("{:.2f}").apply(lambda x: [f'background-color: {colors[i]}' for i in range(len(m

#Display the styled results
styled_tuned_models
```

	Test Accuracy	Test Precision	Test Recall	Test F1 Score
Tuned Logistic Regression	86.27	58.26	66.29	60.13
Tuned Support Vector Machine	88.33	57.68	61.36	58.94
Tuned K-Nearest Neighbour	90.10	57.43	57.77	57.60
Tuned Decision Tree Classifier	86.08	59.40	69.96	61.66
Tuned Random Forest Classifier	88.43	59.46	65.18	61.31

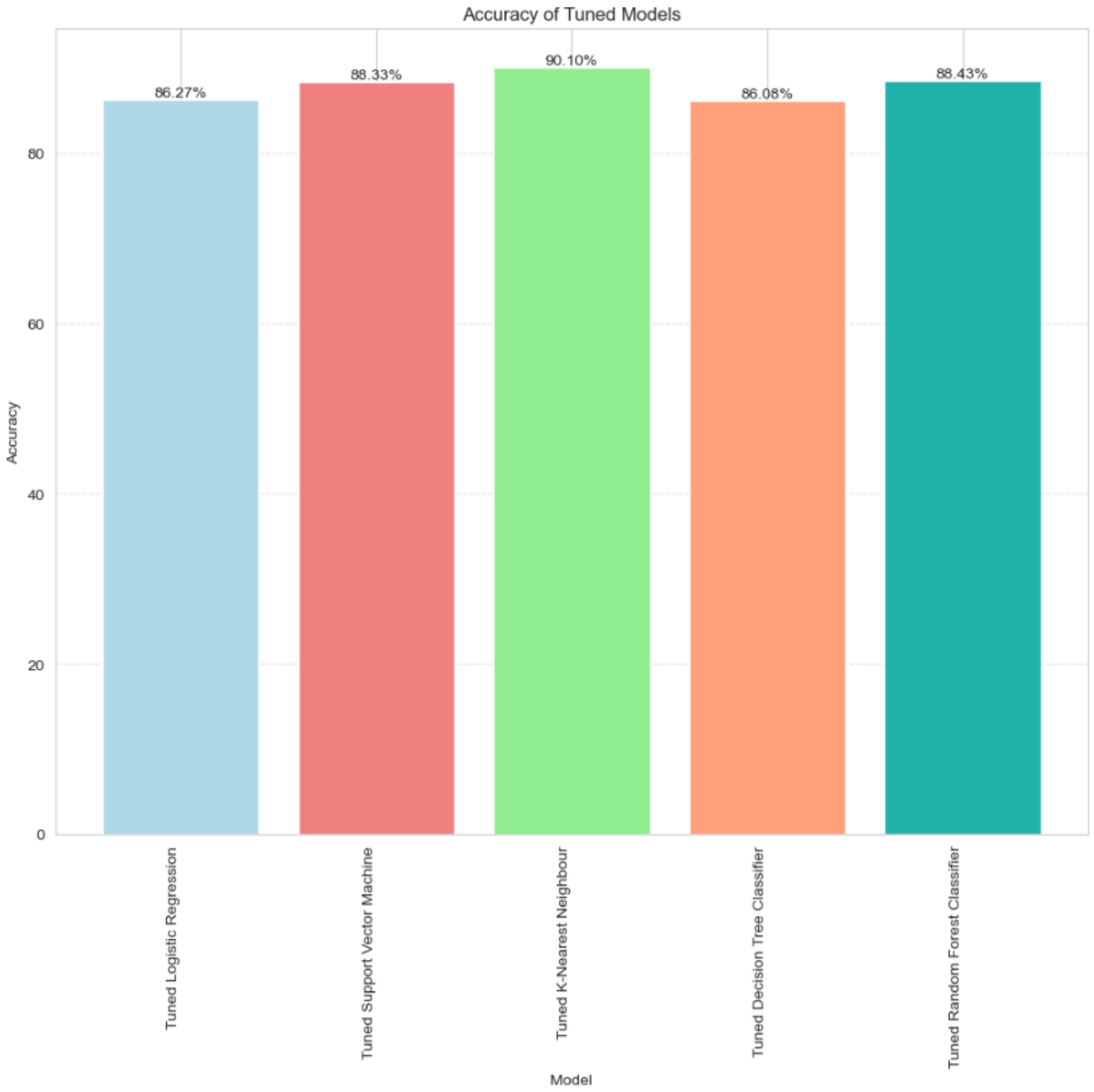
The leader in results has changed: initially RFC was the best performing model however, after hyperparameter tuning KNN has taken the lead at 90.10% accuracy. KNN had the most improvement in terms of metrics after hyperparameter tuning. With these results in mind if we had to choose right now, we would recommend the Tuned KNN model to predict strokes — here's why. For starters, currently it has the highest modelling

performance across the 5 different models, and for obvious reasons we want to use the model with the highest accuracy rate. Its high metrics indicate its effectiveness in imbalanced datasets, this is because KNN does not assume any underlying distribution of the data, making it well-suited for imbalanced datasets where the classes are not evenly distributed. However, things may change possibly, and we are not yet certain of the model we are going to recommend until we evaluate all models on a ROC Curve with AUC scores.

RESULTS

```
#Define the models and their corresponding test accuracy scores
models = ['Tuned Logistic Regression', 'Tuned Support Vector Machine', 'Tuned K-Nearest Neighbour',
          'Tuned Decision Tree Classifier', 'Tuned Random Forest Classifier']
accuracy_scores = [test_score_tuned_lr, test_score_tuned_svm, test_score_tuned_knn,
                   test_score_tuned_dt, test_score_tuned_rfc]

#Define colors for each bar
light_colors = ['lightblue', 'lightcoral', 'lightgreen', 'lightsalmon', 'lightseagreen']

plt.figure(figsize=(10, 10))

#Create bar plot
bars = plt.bar(models, accuracy_scores, color=light_colors)

#Annotate each bar with its accuracy score
for bar, score in zip(bars, accuracy_scores):
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width() / 2, height + 0.02, f'{score:.2f}%',
             ha='center', va='bottom', fontsize=10)

plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Accuracy of Tuned Models')
plt.xticks(rotation=90, ha='right')
plt.grid(axis='y', linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

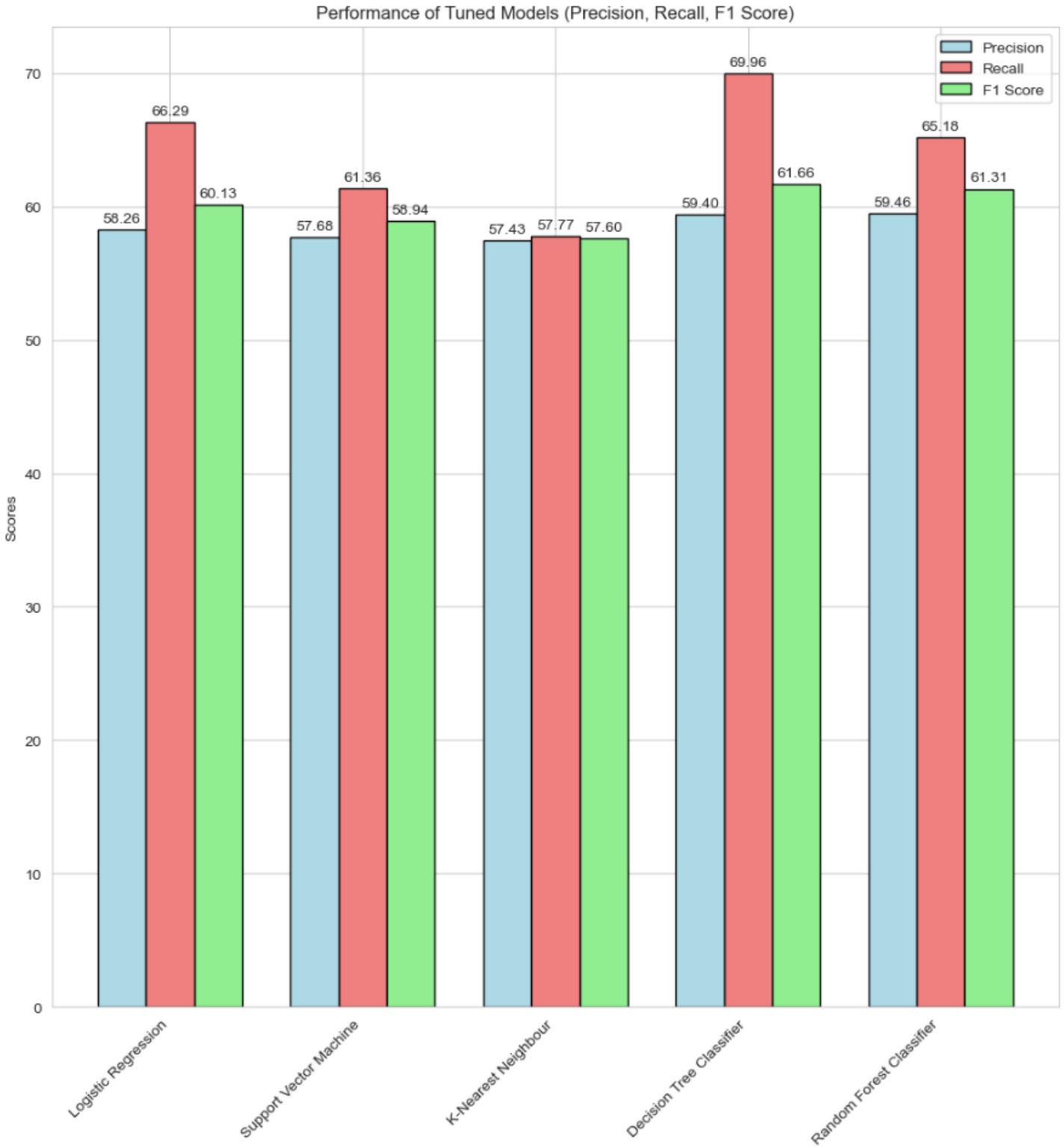
Running the above code to represent a bar plot to visualize test accuracy of each and every tuned model.



This bar plot displays the accuracy scores of each and every tuned model. Visualizing a clearer picture of the best performing models. We can observe that tuned KNN is the best performing model: boasting 90.10% accuracy. In contrast, tuned decision tree is just the worst performing model at 86.08% accuracy. Overall, the test accuracy scores for every model are excellent and the performance differences are marginal.

```
#Define the tuned models and their corresponding scores
tuned_models = ['Logistic Regression', 'Support Vector Machine', 'K-Nearest Neighbour',
                'Decision Tree Classifier', 'Random Forest Classifier']
precision_scores = [test_precision_tuned_lr, test_precision_tuned_svm, test_precision_tuned_knn,
                    test_precision_tuned_dt, test_precision_tuned_rfc]
recall_scores = [test_recall_tuned_lr, test_recall_tuned_svm, test_recall_tuned_knn,
                 test_recall_tuned_dt, test_recall_tuned_rfc]
f1_scores = [test_f1_tuned_lr, test_f1_tuned_svm, test_f1_tuned_knn,
             test_f1_tuned_dt, test_f1_tuned_rfc]

#Set the width of the bars
bar_width = 0.25

#Set the position of the bars on the x-axis
r1 = np.arange(len(tuned_models))
r2 = [x + bar_width for x in r1]
r3 = [x + bar_width for x in r2]

#Grouped bar plot depicting tuned models metric scores
plt.figure(figsize=(10, 11))
bars_precision = plt.bar(r1, precision_scores, color='lightblue', width=bar_width, edgecolor='black', label='Precision')
bars_recall = plt.bar(r2, recall_scores, color='lightcoral', width=bar_width, edgecolor='black', label='Recall')
bars_f1 = plt.bar(r3, f1_scores, color='lightgreen', width=bar_width, edgecolor='black', label='F1 Score')

#Add annotations of ratios
def add_labels(bars):
    for bar in bars:
        height = bar.get_height()
        plt.annotate(f'{height:.2f}', xy=(bar.get_x() + bar.get_width() / 2, height),
                     xytext=(0, 3), textcoords='offset points', ha='center', va='bottom')

add_labels(bars_precision)
add_labels(bars_recall)
add_labels(bars_f1)

#Configure text
plt.xlabel('Model')
plt.ylabel('Scores')
plt.xticks([r + bar_width for r in range(len(tuned_models))], tuned_models, rotation=45, ha='right')
plt.title('Performance of Tuned Models (Precision, Recall, F1 Score)')
plt.legend()
plt.tight_layout()
plt.show()
```

Running the above code to visualize a group bar plot representing the precision, recall, and F1-scores of the hyperparameter tuned models.



This bar plot displays the test macro averages of precision, recall, and F1-score of each and every tuned model. Visualizing a clearer picture of the best performing models in this regard. We can observe that overall, tuned decision tree is the best performing model:

boasting 69.96 recall, 61.66 F1 score, and a 59.40 precision. In contrast, KNN is the worst performing model at 57.77 recall, 57.60 F1 score, and 57.43 precision. Which is strange considering it has the highest accuracy out of all models. However, we have to consider that these are the macro averages, and every model has struggled with the precision, recall, and f1-score of macro averages; the struggle is because of how bad every model performs when it comes to classifying stroke correctly, subsequently bringing the macro averages down tremendously, this is because of the massive support imbalance in the test data. Overall, obviously there is more to be desired with these results.

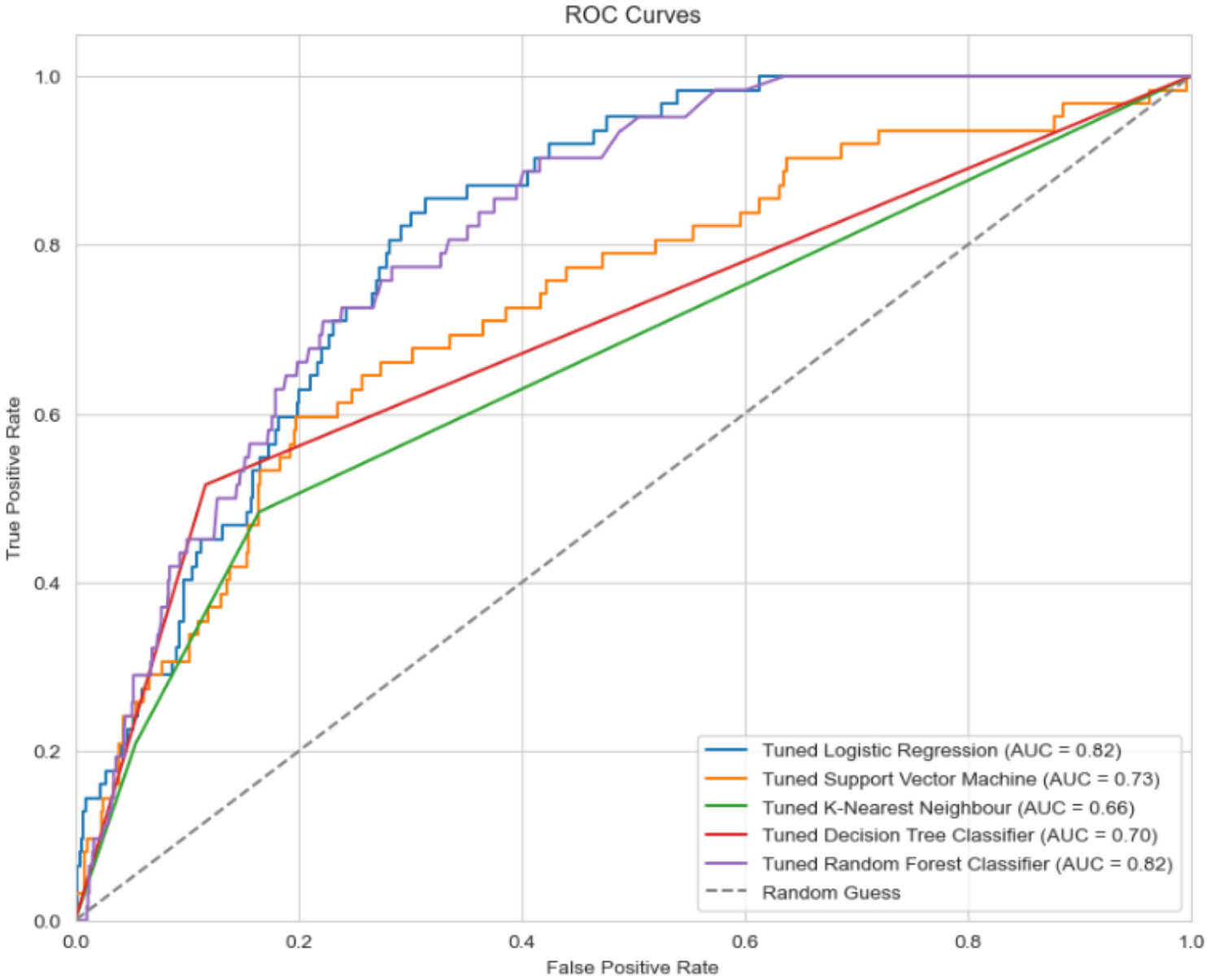
```
#Define tuned models and their corresponding predictions
tuned_models = [tuned_lr, tuned_svm, tuned_knn, tuned_dt, tuned_rfc]
model_names = ['Tuned Logistic Regression', 'Tuned Support Vector Machine', 'Tuned K-Nearest Neighbour',
               'Tuned Decision Tree Classifier', 'Tuned Random Forest Classifier']

plt.figure(figsize=(10, 8))

#Plot ROC curve for each tuned model
for model, name in zip(tuned_models, model_names):
    if hasattr(model, "decision_function"):
        y_score = model.decision_function(X_test_std)
    else:
        y_score = model.predict_proba(X_test_std)[:, 1]

    fpr, tpr, _ = roc_curve(y_test, y_score)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'{name} (AUC = {roc_auc:.2f})')

#Plot the random guess line
plt.plot([0, 1], [0, 1], linestyle='--', color='grey', label='Random Guess')

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

Running the above code to plot a receiver operating characteristic curve (ROC) with AUC (Area Under the ROC Curve) scores for each and every tuned model. The ROC curve is used to evaluate the performance of binary classification models. (which represents our situation: stroke the label we are classifying, 0=no stroke, 1 = stroke). This is how ROC and AUC Work:

- ROC Curve**:** The ROC curve is a plot of the true positive rate against the false positive rate. It shows how well the model is able to distinguish between the positive and negative classes.
- AUC**:** the overall performance of the classifier. It represents the area under the ROC curve, with a score of 1 indicating a perfect classifier and a score of 0.5 indicating a classifier that performs no better than random guessing.

- True Positive Rate (TPR)**:** TPR measures the proportion of actual positive cases that are correctly predicted by the model. It indicates how well the model identifies positive cases out of all actual positive cases.
- False Positive Rate**:** FPR measures the proportion of actual negative cases that are incorrectly predicted as positive by the model. It indicates how often the model incorrectly classifies negative cases as positive.

Basically, you want the ROC curve to be as close as possible to the upper-left corner of the plot, indicating high TPR and low FPR. Additionally, you want the AUC score to be as close to 1 as possible, which demonstrates excellent classifier performance.



This ROC curve highlights every tuned model ROC and AUC test scores. It displays perplexing results in contrast to the modelling phase. The tuned logistic regression and random forest classifier have the best AUC scores at 0.82 – which is excellent, especially for logistic regression which was one of the lower performing models initially. If we observe the ROC curve, we can see tuned logistic regression and tuned random forest classifier showing the best results, with logistic regression having a slightly better curve as it rises to the left the furthest. With these results in mind, we would recommend tuned logistic regression for stroke prediction. It has the best ROC Curve which subsequently means it also has the highest TPR and the lowest FPR indicating it is the best model in classifying strokes correctly. It also is among the models which have the highest AUC rate of 0.82 indicating an excellent classifier and obviously we would want the best performing model to predict strokes — it's a no brainer.

CONCLUSION

Vigorous testing and analysis went into finding the best model to predict stroke, and ultimately it ended up being tuned logistic regression. What sealed the deal for us on picking it was the results of the ROC Curve with AUC scores, despite it being one of the lower performing models initially, it all changed drastically once we observed the ROC and AUC. Which shows in an imbalanced dataset the traditional scores such as accuracy, precision, recall and F1 scores aren't as important as ROC and AUC, especially when you are experiencing a binary classification problem like this. Accuracy was really good in all models, although it was misleading because they were only so good due to the outstanding correct predictions on no stroke, meanwhile correct predictions for stroke were in the dump. As far as the macro averages of recall, precision and f1 scores go, they definitely weren't at a satisfactory level, despite following the best protocols when dealing with an imbalanced dataset; it was still a difficult task to build a model that could accurately predict whether a patient had a stroke to the level of no stroke, because the imbalances were way too vast. For the model to have better traditional scores such as accuracy, precision, recall, and F1 it requires a more significant amount of diverse data for training, specifically more stroke related data. This is necessary to balance out the data and capture the varied nature of human biology, behavior, and habits. In summary, logistic regression offers a combination of interpretability, flexibility, efficiency, and robustness that makes it an excellent choice for stroke event prediction. Its ability to provide probabilistic predictions, interpret model coefficients, handle diverse predictor variables, and maintain performance in large-scale datasets makes it a valuable tool for stroke risk assessment. Ultimately, tuned logistic regression is the best overall performing model at correctly predicting stroke, and I would recommend it to be deployed for stroke event prediction.

REFERENCES

Dataset

Fedesoriano. (2021). healthcare-dataset-stroke-data [Dataset; Csv]. *Stroke Prediction Dataset*. https://www.kaggle.com/datasets/fedesoriano/stroke-prediction-dataset/data

Code References

Durgance Gaur. *A Guide to any Classification Problem*. (2022, February 15). Kaggle. https://www.kaggle.com/code/durgancegaur/a-guide-to-any-classification-problem

Silvano Quarto. *From EDA to explainability.* (2024, March 26). Kaggle. https://www.kaggle.com/code/silvanoquarto31/from-eda-to-explainability

Zahra Rafei. *Highest Accuracy Achievable by Sequential Model.* (2024, March 29). Kaggle. https://www.kaggle.com/code/zahrarafieimelo/highest-accuracy-achievable-by-sequential-model

Menna Mahmoud. *Stroke Prediction with 99% accuracy*. (2024, March 21). Kaggle. https://www.kaggle.com/code/mennatallah77/stroke-prediction-with-99-accuracy/notebook

Umesh Khatiwada. *Using SMOTE for Oversampling Stroke data.* (2024, March 24) Kaggle. https://www.kaggle.com/code/umeshkhatiwada/using-smote-for-oversampling-stroke-data

sklearn.linear_model.LogisticRegression. Scikit. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

sklearn.svm.SVC. Scikit. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

sklearn.neighbors.KNeighborsClassifier. Scikit. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

sklearn.tree.DecisionTreeClassifier. Scikit. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

sklearn.ensemble.RandomForestClassifier. Scikit. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html