Machine Learning Project: Stroke Prediction

Name: Okba Kharef

1.Context

- According to the World Health Organization (WHO) stroke is the 2nd leading cause of death globally, responsible for approximately 11% of total deaths. This dataset is used to predict whether a patient is likely to get stroke based on the input parameters like gender, age, various diseases, and smoking status. Each row in the data provides relavant information about the patient.

Attribute Information

1. **id**: unique identifier
2. **gender**: "Male", "Female" or "Other"
3. **age**: age of the patient
4. **hypertension**: 0 if the patient doesn't have hypertension, 1 if the patient has hypertension
5. **heart_disease**: 0 if the patient doesn't have any heart diseases, 1 if the patient has a heart disease
6. **ever_married**: "No" or "Yes"
7. **work_type**: "children", "Govt_jov", "Never_worked", "Private" or "Self-employed"
8. **Residence_type**: "Rural" or "Urban"
9. **avg_glucose_level**: average glucose level in blood
10. **bmi**: body mass index
11. **smoking_status**: "formerly smoked", "never smoked", "smokes" or "Unknown"*
12. **stroke**: 1 if the patient had a stroke or 0 if not

*Note: "Unknown" in smoking_status means that the information is unavailable for this patient

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

2. Project Objective

The goal of this project is to analyze a dataset of patient attributes to predict the likelihood of a stroke. We will perform a comprehensive analysis which includes:

1. **Exploratory Data Analysis (EDA):** To understand the data's structure, features, and identify potential challenges .
2. **Data Preprocessing:** To clean and prepare the data for machine learning models.
3. **Model Training:** To build and train several classification algorithms, including both basic models and more complex ensemble methods.
4. **Model Evaluation & Comparison:** To evaluate the models using appropriate metrics for an imbalanced dataset and to compare their performance to identify the most effective algorithm.

2. Setup: Importing Libraries

This first code block is for importing all the necessary Python libraries for the project.

```
# For data manipulation and analysis
import pandas as pd
import numpy as np

# For data visualization
import matplotlib.pyplot as plt
import seaborn as sns
import plotly
import plotly.express as px
import plotly.graph_objs as go
import plotly.offline as py
from plotly.offline import iplot
from plotly.subplots import make_subplots
import plotly.figure_factory as ff
import missingno as msno
from pywaffle import Waffle



# For splitting data and preprocessing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler # Optional: for
scaling numerical features
from sklearn.preprocessing import FunctionTransformer  # Transforming
of Data
from sklearn.preprocessing import OneHotEncoder # Data Encoding
from sklearn.preprocessing import StandardScaler # Data Scaling
from imblearn.over_sampling import RandomOverSampler # Data
OverSampling
from sklearn.decomposition import PCA # Principal Component Analysis

# For handling class imbalance
from imblearn.over_sampling import SMOTE

# Machine Learning Models
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import KFold, GridSearchCV
from scipy.stats import wilcoxon


import tensorflow
from tensorflow import keras
from tensorflow.keras import Sequential
from keras.models import Sequential
from keras.layers import Dense,Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import BinaryCrossentropy

# For evaluating model performance
from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score, f1_score, accuracy_score, precision_score,
recall_score,mean_squared_error, mean_absolute_error, make_scorer,
precision_score, recall_score, ConfusionMatrixDisplay



# To ignore warnings
import warnings
warnings.filterwarnings('ignore')
```

3. Loading the Data

Here, we load the dataset from the CSV file into a pandas DataFrame.

```
# Load the dataset
#dataset URL= https://www.kaggle.com/datasets/fedesoriano/stroke-
prediction-dataset
df = pd.read_csv('Data/healthcare-dataset-stroke-data.csv')
```

4. Exploratory Data Analysis (EDA)

In this section, we explore the data to understand its properties and find any issues that need to be addressed.

4.1. Initial Data Inspection

Let's look at the first few rows, the data types, and a statistical summary.

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

```
      id  gender   age  hypertension  heart_disease ever_married  \
0   9046    Male  67.0             0              1          Yes
1  51676  Female  61.0             0              0          Yes
2  31112    Male  80.0             0              1          Yes
3  60182  Female  49.0             0              0          Yes
4   1665  Female  79.0             1              0          Yes

       work_type Residence_type  avg_glucose_level   bmi
smoking_status  \
0        Private          Urban             228.69  36.6  formerly
smoked
1  Self-employed          Rural             202.21   NaN     never
smoked
2        Private          Rural             105.92  32.5     never
smoked
3        Private          Urban             171.23  34.4
smokes
4  Self-employed          Rural             174.12  24.0     never
smoked

   stroke
0       1
1       1
2       1
3       1
4       1
```

```
print (f' We have {df.shape[0]} instances with the {df.shape[1]-1}
features and 1 output variable')
```

```
 We have 5110 instances with the 11 features and 1 output variable
```

```
# Get a concise summary of the dataframe, including data types and
non-null values
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
```

- In our dataset, we have both numerical and categorical variables.
- It is essential to see whether columns are correctly inferred.
- The most important one to look for is our target variable 'stroke'
- 'Stroke' is detected as an integer, not as an object.
- Target variable is coded as 1 for positive cases (has a stroke) and 0 for negative cases (does not have a stroke)
- Both 'Hypertension' and 'heart disease" are detected as an integer, not as an object.
- Just remember from the data definition part, they are coded as 1 for the positive cases(has hypertension/heart disease)
- And 0 for the negative cases (does not have hypertension/heart disease)
- We don't need to change them, but it is good to see and be aware of it.
- In addition to them, we have 3 categorical variables, which we have to encode as numerical.

What Problem We Have?

- We have binary classification problem.
- We make prection on the target variable **STROKE**
- And we will build a model to get best prediction on the stroke variable.

Target Variable

- One of the first steps of exploratory data analysis should always be to look at what the values of y look like.

```
y = df['stroke']
print(f'Percentage of patient had a stroke: %
{round(y.value_counts(normalize=True)[1]*100,2)} -->
({y.value_counts()[1]} patient)\nPercentage of patient did not have a
stroke: % {round(y.value_counts(normalize=True)[0]*100,2)} -->
({y.value_counts()[0]} patient)')
```

```
Percentage of patient had a stroke: % 4.87 --> (249 patient)
Percentage of patient did not have a stroke: % 95.13 --> (4861
patient)
```

- Almost %95 of the instances of our target variable is 'No stroke'
- 4861 patient does not have a stroke
- %5 of the instances of our target variable is 'Stroke'

- 249 patient have a stroke.
- We have imbalanced data.

```
# Define color variables if not already defined
COLOR_NO_STROKE = "#512b58"  # Example: purple for no stroke
COLOR_STROKE = "#fe346e"     # Example: red for stroke

stroke_counts = df['stroke'].value_counts().reset_index()
stroke_counts.columns = ['stroke', 'count']
stroke_counts['stroke'] = stroke_counts['stroke'].astype(str) # For
discrete colors

fig = px.bar(stroke_counts, x="stroke", y="count", title='Distribution
of Target Variable: Stroke',
             width=500, height=400, # Slightly wider for better title
display
             color='stroke', # Use the 'stroke' column for color
mapping
             color_discrete_map={'0': COLOR_NO_STROKE, '1':
COLOR_STROKE},
             labels={'stroke': 'Stroke Occurrence', 'count': 'Number
of Patients'})
fig.update_layout(title_font_size=18,  # Set a fixed font size for the
title
                  font=dict(family="Serif"))  # Use a generic serif
font
fig.show()
```


```
{"config":{"plotlyServerURL":"https://plot.ly"},"data":
[{"hovertemplate":"Stroke Occurrence=%{x}<br>Number of Patients=%
{y}<extra></extra>","legendgroup":"0","marker":
{"color":"#512b58","pattern":
{"shape":""}},"name":"0","orientation":"v","showlegend":true,"textposi
tion":"auto","type":"bar","x":["0"],"xaxis":"x","y":{"bdata":"/
RI=","dtype":"i2"},"yaxis":"y"},{"hovertemplate":"Stroke Occurrence=%
{x}<br>Number of
Patients=%{y}<extra></extra>","legendgroup":"1","marker":
{"color":"#fe346e","pattern":
{"shape":""}},"name":"1","orientation":"v","showlegend":true,"textposi
tion":"auto","type":"bar","x":["1"],"xaxis":"x","y":
{"bdata":"+QA=","dtype":"i2"},"yaxis":"y"}],"layout":
{"barmode":"relative","font":{"family":"Serif"},"height":400,"legend":
{"title":{"text":"Stroke Occurrence"},"tracegroupgap":0},"template":
{"data":{"bar":[{"error_x":{"color":"#2a3f5f"},"error_y":
{"color":"#2a3f5f"},"marker":{"line":
{"color":"#E5ECF6","width":0.5},"pattern":
{"fillmode":"overlay","size":10,"solidity":0.2}},"type":"bar"}],"barpo
lar":[{"marker":{"line":{"color":"#E5ECF6","width":0.5},"pattern":
{"fillmode":"overlay","size":10,"solidity":0.2}},"type":"barpolar"}],"
```

```
carpet":[{"aaxis":
{"endlinecolor":"#2a3f5f","gridcolor":"white","linecolor":"white","min
orgridcolor":"white","startlinecolor":"#2a3f5f"},"baxis":
{"endlinecolor":"#2a3f5f","gridcolor":"white","linecolor":"white","min
orgridcolor":"white","startlinecolor":"#2a3f5f"},"type":"carpet"}],"ch
oropleth":[{"colorbar":
{"outlinewidth":0,"ticks":""},"type":"choropleth"}],"contour":
[{"colorbar":{"outlinewidth":0,"ticks":""},"colorscale":
[[0,"#0d0887"],[0.1111111111111111,"#46039f"],
[0.2222222222222222,"#7201a8"],[0.3333333333333333,"#9c179e"],
[0.4444444444444444,"#bd3786"],[0.5555555555555556,"#d8576b"],
[0.6666666666666666,"#ed7953"],[0.7777777777777778,"#fb9f3a"],
[0.8888888888888888,"#fdca26"],
[1,"#f0f921"]],"type":"contour"}],"contourcarpet":[{"colorbar":
{"outlinewidth":0,"ticks":""},"type":"contourcarpet"}],"heatmap":
[{"colorbar":{"outlinewidth":0,"ticks":""},"colorscale":
[[0,"#0d0887"],[0.1111111111111111,"#46039f"],
[0.2222222222222222,"#7201a8"],[0.3333333333333333,"#9c179e"],
[0.4444444444444444,"#bd3786"],[0.5555555555555556,"#d8576b"],
[0.6666666666666666,"#ed7953"],[0.7777777777777778,"#fb9f3a"],
[0.8888888888888888,"#fdca26"],
[1,"#f0f921"]],"type":"heatmap"}],"histogram":[{"marker":{"pattern":
{"fillmode":"overlay","size":10,"solidity":0.2}},"type":"histogram"}],
"histogram2d":[{"colorbar":{"outlinewidth":0,"ticks":""},"colorscale":
[[0,"#0d0887"],[0.1111111111111111,"#46039f"],
[0.2222222222222222,"#7201a8"],[0.3333333333333333,"#9c179e"],
[0.4444444444444444,"#bd3786"],[0.5555555555555556,"#d8576b"],
[0.6666666666666666,"#ed7953"],[0.7777777777777778,"#fb9f3a"],
[0.8888888888888888,"#fdca26"],
[1,"#f0f921"]],"type":"histogram2d"}],"histogram2dcontour":
[{"colorbar":{"outlinewidth":0,"ticks":""},"colorscale":
[[0,"#0d0887"],[0.1111111111111111,"#46039f"],
[0.2222222222222222,"#7201a8"],[0.3333333333333333,"#9c179e"],
[0.4444444444444444,"#bd3786"],[0.5555555555555556,"#d8576b"],
[0.6666666666666666,"#ed7953"],[0.7777777777777778,"#fb9f3a"],
[0.8888888888888888,"#fdca26"],
[1,"#f0f921"]],"type":"histogram2dcontour"}],"mesh3d":[{"colorbar":
{"outlinewidth":0,"ticks":""},"type":"mesh3d"}],"parcoords":[{"line":
{"colorbar":{"outlinewidth":0,"ticks":""}},"type":"parcoords"}],"pie":
[{"automargin":true,"type":"pie"}],"scatter":[{"fillpattern":
{"fillmode":"overlay","size":10,"solidity":0.2},"type":"scatter"}],"sc
atter3d":[{"line":{"colorbar":{"outlinewidth":0,"ticks":""}},"marker":
{"colorbar":
{"outlinewidth":0,"ticks":""}},"type":"scatter3d"}],"scattercarpet":
[{"marker":{"colorbar":
{"outlinewidth":0,"ticks":""}},"type":"scattercarpet"}],"scattergeo":
[{"marker":{"colorbar":
{"outlinewidth":0,"ticks":""}},"type":"scattergeo"}],"scattergl":
[{"marker":{"colorbar":
```

{"outlinewidth":0,"ticks":""}},"type":"scattergl"}],"scattermap":
[{"marker":{"colorbar":
{"outlinewidth":0,"ticks":""}},"type":"scattermap"}],"scattermapbox":
[{"marker":{"colorbar":
{"outlinewidth":0,"ticks":""}},"type":"scattermapbox"}],"scatterpolar"
:[{"marker":{"colorbar":
{"outlinewidth":0,"ticks":""}},"type":"scatterpolar"}],"scatterpolargl
":[{"marker":{"colorbar":
{"outlinewidth":0,"ticks":""}},"type":"scatterpolargl"}],"scatterterna
ry":[{"marker":{"colorbar":
{"outlinewidth":0,"ticks":""}},"type":"scatterternary"}],"surface":
[{"colorbar":{"outlinewidth":0,"ticks":""},"colorscale":
[[0,"#0d0887"],[0.1111111111111111,"#46039f"],
[0.2222222222222222,"#7201a8"],[0.3333333333333333,"#9c179e"],
[0.4444444444444444,"#bd3786"],[0.5555555555555556,"#d8576b"],
[0.6666666666666666,"#ed7953"],[0.7777777777777778,"#fb9f3a"],
[0.8888888888888888,"#fdca26"],
[1,"#f0f921"]],"type":"surface"}],"table":[{"cells":{"fill":
{"color":"#EBF0F8"},"line":{"color":"white"}},"header":{"fill":
{"color":"#C8D4E3"},"line":
{"color":"white"}},"type":"table"}]},"layout":{"annotationdefaults":
{"arrowcolor":"#2a3f5f","arrowhead":0,"arrowwidth":1},"autotypenumbers
":"strict","coloraxis":{"colorbar":
{"outlinewidth":0,"ticks":""}},"colorscale":{"diverging":
[[0,"#8e0152"],[0.1,"#c51b7d"],[0.2,"#de77ae"],[0.3,"#f1b6da"],
[0.4,"#fde0ef"],[0.5,"#f7f7f7"],[0.6,"#e6f5d0"],[0.7,"#b8e186"],
[0.8,"#7fbc41"],[0.9,"#4d9221"],[1,"#276419"]],"sequential":
[[0,"#0d0887"],[0.1111111111111111,"#46039f"],
[0.2222222222222222,"#7201a8"],[0.3333333333333333,"#9c179e"],
[0.4444444444444444,"#bd3786"],[0.5555555555555556,"#d8576b"],
[0.6666666666666666,"#ed7953"],[0.7777777777777778,"#fb9f3a"],
[0.8888888888888888,"#fdca26"],[1,"#f0f921"]],"sequentialminus":
[[0,"#0d0887"],[0.1111111111111111,"#46039f"],
[0.2222222222222222,"#7201a8"],[0.3333333333333333,"#9c179e"],
[0.4444444444444444,"#bd3786"],[0.5555555555555556,"#d8576b"],
[0.6666666666666666,"#ed7953"],[0.7777777777777778,"#fb9f3a"],
[0.8888888888888888,"#fdca26"],[1,"#f0f921"]]},"colorway":
["#636efa","#EF553B","#00cc96","#ab63fa","#FFA15A","#19d3f3","#FF6692"
,"#B6E880","#FF97FF","#FECB52"],"font":{"color":"#2a3f5f"},"geo":
{"bgcolor":"white","lakecolor":"white","landcolor":"#E5ECF6","showlake
s":true,"showland":true,"subunitcolor":"white"},"hoverlabel":
{"align":"left"},"hovermode":"closest","mapbox":
{"style":"light"},"paper_bgcolor":"white","plot_bgcolor":"#E5ECF6","po
lar":{"angularaxis":
{"gridcolor":"white","linecolor":"white","ticks":""},"bgcolor":"#E5ECF
6","radialaxis":
{"gridcolor":"white","linecolor":"white","ticks":""}},"scene":
{"xaxis":
{"backgroundcolor":"#E5ECF6","gridcolor":"white","gridwidth":2,"lineco

```
lor":"white","showbackground":true,"ticks":"","zerolinecolor":"white"}
,"yaxis":
{"backgroundcolor":"#E5ECF6","gridcolor":"white","gridwidth":2,"lineco
lor":"white","showbackground":true,"ticks":"","zerolinecolor":"white"}
,"zaxis":
{"backgroundcolor":"#E5ECF6","gridcolor":"white","gridwidth":2,"lineco
lor":"white","showbackground":true,"ticks":"","zerolinecolor":"white"}
},"shapedefaults":{"line":{"color":"#2a3f5f"}},"ternary":{"aaxis":
{"gridcolor":"white","linecolor":"white","ticks":""},"baxis":
{"gridcolor":"white","linecolor":"white","ticks":""},"bgcolor":"#E5ECF
6","caxis":
{"gridcolor":"white","linecolor":"white","ticks":""}},"title":
{"x":5.0e-2},"xaxis":
{"automargin":true,"gridcolor":"white","linecolor":"white","ticks":"",
"title":
{"standoff":15},"zerolinecolor":"white","zerolinewidth":2},"yaxis":
{"automargin":true,"gridcolor":"white","linecolor":"white","ticks":"",
"title":
{"standoff":15},"zerolinecolor":"white","zerolinewidth":2}}},"title":
{"font":{"size":18},"text":"Distribution of Target Variable:
Stroke"},"width":500,"xaxis":{"anchor":"y","categoryarray":
["0","1"],"categoryorder":"array","domain":[0,1],"title":
{"text":"Stroke Occurrence"}},"yaxis":{"anchor":"x","domain":
[0,1],"title":{"text":"Number of Patients"}}}}
```


- Our stroke dataset is an example of a so-called imbalanced dataset.
- There are 19 times more people who didn't have stroke in our data than who had, and we say that the non-stroke class dominates the stroke class.
- We can clearly see that: the stroke rate in our data is 0.048
- Which is a strong indicator of class imbalance

Imbalance Data

- Instances across classes are imbalanced, like in our dataset, we have imbalance data.
- The problem is, most of the machine learning algorithm do not work well with the imbalanced data.
- Some of the metrics (like accuracy) give us misleading results.
- Most of the time in classification problems our interest is to get better predict on the minority class.
- In our example: People had a stroke is minority class.
- Otherwise our machine learning algorithm falsely predicts majority class.
- In our example: No stroke is majority class.

There are a few different ways to handle imbalanced datasets, such as *undersampling* the majority class, or *oversampling* the minority one. We could also use more advanced algorithms, such as *SMOTE*, to generate synthetic samples from the minority class.

Decide the Metric

- This is the first step when approaching a machine learning problem: decide the metric!
- My reasoning is that minimizing false negatives, and therefore maximizing sensitivity/recall, is the priority in this context, as classifying someone as without STROKE when they have it would cause delayed or missed critical treatment, leading to significantly worse patient outcomes, permanent disability, or even death. False positives are undesirable, as people would take unnecessary precautions, but not as important as minimizing false negatives.
- The choice of the wrong metric can mean choosing the wrong algorithm.
- Rather than the basic *accuracy* metric, we will use *balanced accuracy*, which is suited for imbalanced data. *Balanced accuracy* is defined as the arithmetic mean of *accuracy* and *recall*:

$$\text{Balanced Accuracy} = \frac{sensitivity + specificity}{2}$$

Where *sensitivity* is the proportion of actual positives that are correctly identified as such, *specificity* is the proportion of actual negatives that are correctly identified, and *precision* quantifies the number of correct positive predictions made out of positive predictions made by the model.

$$ \text{Sensitivity} = \frac{TP}{TP + FN} \hspace{6 mm} \text{Specificity} = \frac{TN}{TN + FP} \hspace{6 mm}

$$

```
# Get descriptive statistics for numerical columns
df.describe()
```

```
                 id          age  hypertension  heart_disease  \
count   5110.000000  5110.000000   5110.000000    5110.000000
mean   36517.829354    43.226614      0.097456       0.054012
std    21161.721625    22.612647      0.296607       0.226063
min       67.000000     0.080000      0.000000       0.000000
25%    17741.250000    25.000000      0.000000       0.000000
50%    36932.000000    45.000000      0.000000       0.000000
75%    54682.000000    61.000000      0.000000       0.000000
max    72940.000000    82.000000      1.000000       1.000000

       avg_glucose_level          bmi       stroke
count        5110.000000  4909.000000  5110.000000
mean          106.147677    28.893237     0.048728
```

```
std            45.283560     7.854067     0.215320
min            55.120000    10.300000     0.000000
25%            77.245000    23.500000     0.000000
50%            91.885000    28.100000     0.000000
75%           114.090000    33.100000     0.000000
max           271.740000    97.600000     1.000000
```

```
# This helps check for any rare categories.

categorical_cols = df.select_dtypes(include=['object']).columns

for col in categorical_cols:
    print(f"Value counts for column: {col}")
    print(df[col].value_counts())
    print("-" * 30)
```

```
Value counts for column: gender
gender
Female    2994
Male      2115
Other        1
Name: count, dtype: int64
------------------------------
Value counts for column: ever_married
ever_married
Yes    3353
No     1757
Name: count, dtype: int64
------------------------------
Value counts for column: work_type
work_type
Private          2925
Self-employed     819
children          687
Govt_job          657
Never_worked       22
Name: count, dtype: int64
------------------------------
Value counts for column: Residence_type
Residence_type
Urban    2596
Rural    2514
Name: count, dtype: int64
------------------------------
Value counts for column: smoking_status
smoking_status
never smoked       1892
Unknown            1544
formerly smoked     885
smokes              789
```

```
Name: count, dtype: int64
------------------------------
```

4.2. Handling Missing Values

-In our initial data inspection with **.info()**, we identified that the bmi column is the only one with missing values. Before we can train our models, we must address these missing entries through imputation, which means filling in the blanks with a substituted value.

```
df.isnull().sum()
```

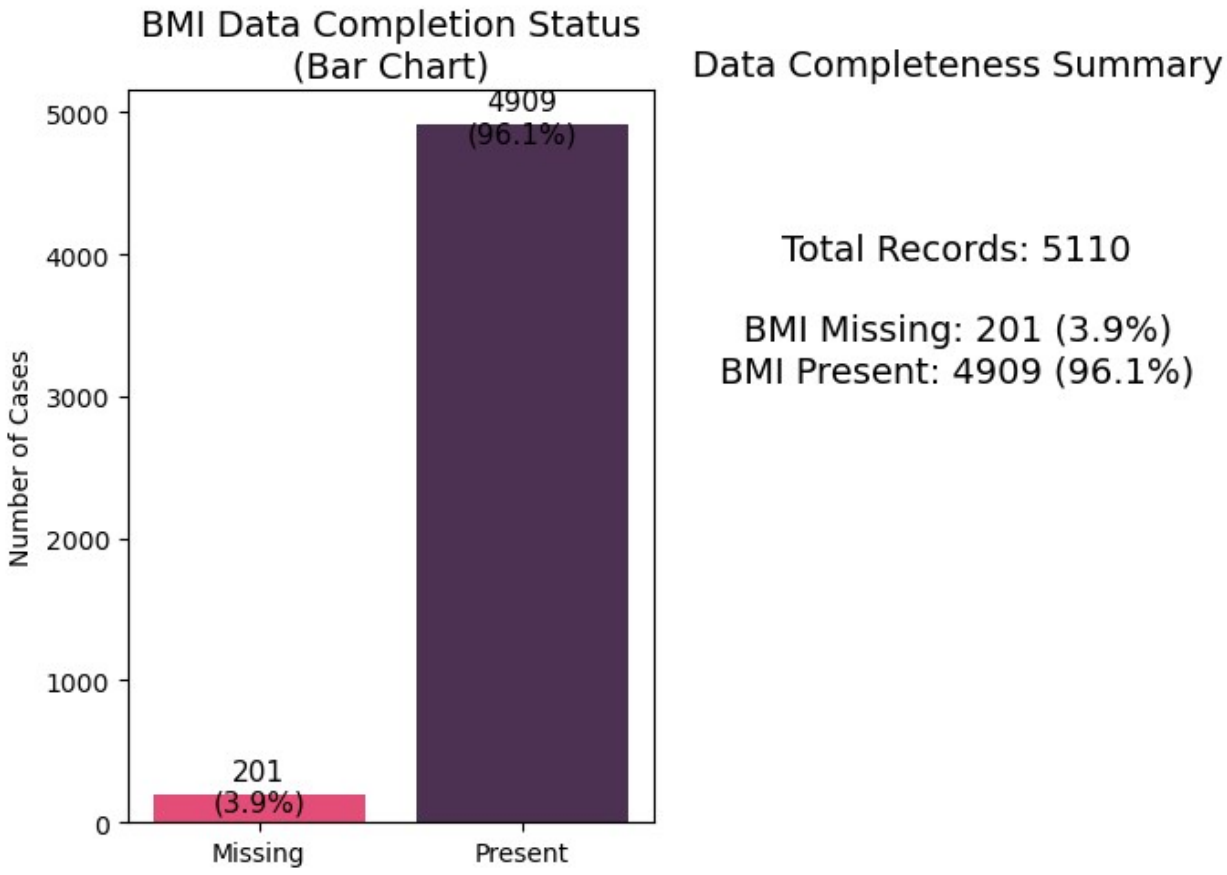
```
id                     0
gender                 0
age                    0
hypertension           0
heart_disease          0
ever_married           0
work_type              0
Residence_type         0
avg_glucose_level      0
bmi                  201
smoking_status         0
stroke                 0
dtype: int64
```

```
missing_bmi_count = df['bmi'].isnull().sum()
total_rows = len(df)
missing_percentage = (missing_bmi_count / total_rows) * 100

print(f"Number of missing BMI values: {missing_bmi_count}")
print(f"Percentage of missing BMI values: {missing_percentage:.2f}%")
```

```
Number of missing BMI values: 201
Percentage of missing BMI values: 3.93%
```

```
# Visualize missing values in the dataset
# Calculate missing values
total = len(df)
missing = df['bmi'].isna().sum()
present = total - missing
missing_pct = round(missing/total * 100, 1)

# Define colors for missing and present values
COLOR_MISSING = "#fe346e"   # red (same as COLOR_STROKE)
COLOR_PRESENT = "#512b58"   # purple (same as COLOR_NO_STROKE)

# Create visualization
plt.figure(figsize=(7, 5))

# Bar plot
```

```
plt.subplot(121)
ax = sns.barplot(x=['Missing', 'Present'],
                 y=[missing, present],
                palette=[COLOR_MISSING, COLOR_PRESENT])
plt.title('BMI Data Completion Status\n(Bar Chart)', fontsize=14)
plt.ylabel('Number of Cases')

# Add annotations
for i, v in enumerate([missing, present]):
    pct = f'{missing_pct}%' if i == 0 else f'{100-missing_pct}%'
    ax.text(i, v + 50, f'{v}\n({pct})',
            ha='center',
            va='center',
            fontsize=11)

# Text-based visualization
plt.subplot(122)
plt.text(0.5, 0.7,
         f"Total Records: {total}\n\nBMI Missing: {missing}
({missing_pct}%)\nBMI Present: {present} ({100-missing_pct}%)",
         fontsize=14,
         ha='center',
         va='center')
plt.axis('off')
plt.title('Data Completeness Summary', fontsize=14)

plt.tight_layout()
plt.show()

# Print detailed summary
print(f"Detailed Summary:")
print(f"Total records: {total}")
print(f"Missing BMI: {missing} ({missing_pct}%)")
print(f"Available BMI: {present} ({100-missing_pct}%)")
```




```
Detailed Summary:
Total records: 5110
Missing BMI: 201 (3.9%)
Available BMI: 4909 (96.1%)
```

4.2.1.Choosing the Right Imputation Strategy: Mean vs. Median

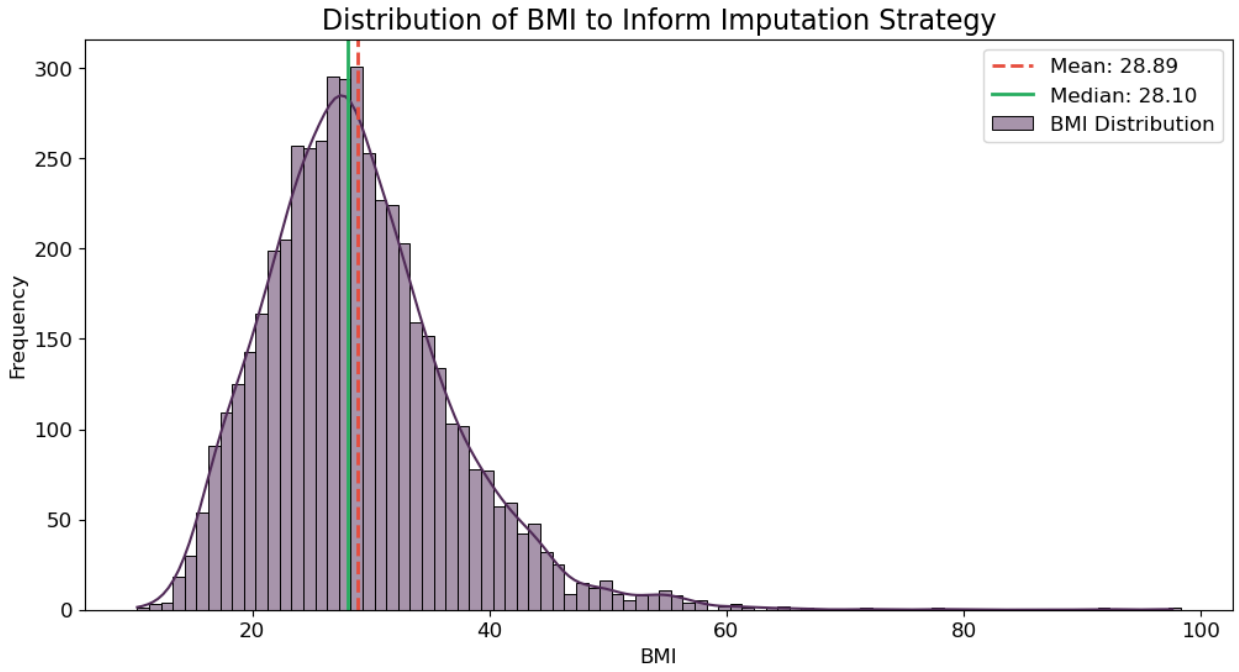
With approximately **4% of BMI** values missing, simply deleting these rows would cause a loss of valuable data. A better approach is to impute them. The two most common strategies are using the mean or the median value of the column.

- **Mean**: The average value. It is simple but **can be easily influenced by extremely high or low values (outliers)**.
- **Median**: The middle value of the sorted data. It is known to be **robust to outliers**.

To make an informed decision, we must visualize the distribution of the bmi column. This will show us if the data is symmetric or if it is "skewed" by the presence of outliers.

```
# Original: color='skyblue', mean color='red', median color='green'
# Suggested:
plt.figure(figsize=(12, 6)) # Consistent size
# plt.title('Distribution of BMI (with Mean and Median)',
fontsize=TITLE_FONT_ARGS['fontsize']) # Using global rcParams
```

```
# plt.xlabel('BMI', fontsize=AXIS_LABEL_FONT_ARGS['fontsize'])
# plt.ylabel('Frequency', fontsize=AXIS_LABEL_FONT_ARGS['fontsize'])

# Define colors for mean and median if not already defined
COLOR_MEAN = "#e74c3c"    # red
COLOR_MEDIAN = "#27ae60"  # green

# Define font argument dictionaries if not already defined
TITLE_FONT_ARGS = {'fontsize': 18}
AXIS_LABEL_FONT_ARGS = {'fontsize': 14}
LEGEND_FONT_ARGS = {'fontsize': 12}
TICK_LABEL_FONT_ARGS = {'fontsize': 12}

sns.histplot(df['bmi'].dropna(), kde=True, color=COLOR_PRESENT,
binwidth=1, label='BMI Distribution')
plt.axvline(df['bmi'].mean(), color=COLOR_MEAN, linestyle='--',
linewidth=2, label=f"Mean: {df['bmi'].mean():.2f}")
plt.axvline(df['bmi'].median(), color=COLOR_MEDIAN, linestyle='-',
linewidth=2, label=f"Median: {df['bmi'].median():.2f}")
plt.legend(fontsize=LEGEND_FONT_ARGS['fontsize'])
plt.title('Distribution of BMI to Inform Imputation Strategy',
fontdict={'fontsize': TITLE_FONT_ARGS['fontsize']-2}) # Use args if
not fully relying on rcParams
plt.xlabel('BMI', fontdict={'fontsize':
AXIS_LABEL_FONT_ARGS['fontsize']-2})
plt.ylabel('Frequency', fontdict={'fontsize':
AXIS_LABEL_FONT_ARGS['fontsize']-2})
plt.xticks(fontsize=TICK_LABEL_FONT_ARGS['fontsize'])
plt.yticks(fontsize=TICK_LABEL_FONT_ARGS['fontsize'])
plt.show()
```



4.2.2. Analysis of the Distribution and Final Decision

The visualization provides critical insights for our decision:

1. **Right-Skewed Distribution**: The graph is not symmetrical. There is a "tail" extending to the right, which indicates the presence of a minority of patients with very high BMI values **(outliers).**
2. **Mean vs. Median Position**: While the two values are numerically close (Mean: 28.89, Median: 28.10), their positions are telling. The Mean (red line) is positioned slightly to the right of the Median (green line).

Conclusion: This slight difference is visual proof that the outliers in the tail are pulling the Mean towards them. Even though the effect is small, it confirms the Mean is being influenced by extreme values. The Median, located at the center of the main population hump, is unaffected. For this reason, **the Median is the more statistically robust and honest representation of a 'typical' patient.** Therefore, it is the best choice for imputation.

We will now fill the missing BMI values using the calculated median.

```
# Calculate the median value of the 'bmi' column
bmi_median = df['bmi'].median()
print(f"The calculated median BMI to be used for imputation is:
{bmi_median:.2f}")

# Impute the missing values in 'bmi' with the median.
# The inplace=True argument modifies the DataFrame directly, so we
don't need to re-assign it.
df['bmi'].fillna(bmi_median, inplace=True)
```

```
The calculated median BMI to be used for imputation is: 28.10
```

4.2.3.Verification

```
# Check the number of missing values in the 'bmi' column one last time
final_missing_count = df['bmi'].isnull().sum()

print(f"Number of missing BMI values after imputation:
{final_missing_count}")

if final_missing_count == 0:
    print("Verification successful: The 'bmi' column has no missing
values.")
else:
    print("Warning: Missing values still exist in the 'bmi' column.")
```

```
Number of missing BMI values after imputation: 0
Verification successful: The 'bmi' column has no missing values.
```

4.3. Data Visualization

Next, we want to explore the data.

fundamentally we need to understand the relationships within the data.

- Which features seem to be important for predicting a stroke?
- What do the characteristics of patients who have strokes look like?
- Does age makes one more likely to suffer a stroke? What about gender? Or BMI?

These are all questions that can be explored and answered with some data visulization.

First, let's look at the numeric/continuous variable distribtion

Numerical and Categorical Features

Features Identification

```
# Create lists to hold the names of columns of different data types
numerical_features = []
categorical_features = []

# Loop through all the columns in the DataFrame
for col in df.columns:
    # Check if the column is of a numeric data type and has more than
2 unique values
    # (We consider columns with only 2 unique values, like
'hypertension', as categorical)
    if pd.api.types.is_numeric_dtype(df[col]) :  #and
df[col].nunique() > 2
        numerical_features.append(col)
    # All other columns are treated as categorical
    else:
        categorical_features.append(col)
```

```
# Print the lists to see the results
print("--- Column Types ---")
print("\nNumerical Features:")
for feature in numerical_features:
    print(f"  - {feature}")
print("\nCategorical Features:")
for feature in categorical_features:
    print(f"  - {feature}")
```

```
--- Column Types ---

Numerical Features:
  - id
  - age
  - hypertension
  - heart_disease
  - avg_glucose_level
  - bmi
  - stroke

Categorical Features:
  - gender
  - ever_married
  - work_type
  - Residence_type
  - smoking_status
```

For feature extraction, binning was applied for all the continous values, binning values are taken from follow articles.

- body mass index binning
- Age binning
- average glucose binning

```
df['bmi_cat'] = pd.cut(df['bmi'], bins = [0, 19, 25,30,10000], labels
= ['Underweight', 'Ideal', 'Overweight', 'Obesity'])
df['age_cat'] = pd.cut(df['age'], bins = [0,13,18, 45,60,200], labels
= ['Children', 'Teens', 'Adults','Mid Adults','Elderly'])
df['glucose_cat'] = pd.cut(df['avg_glucose_level'], bins =
[0,90,160,230,500], labels = ['Low', 'Normal', 'High', 'Very High'])
```

analysis of continuous features

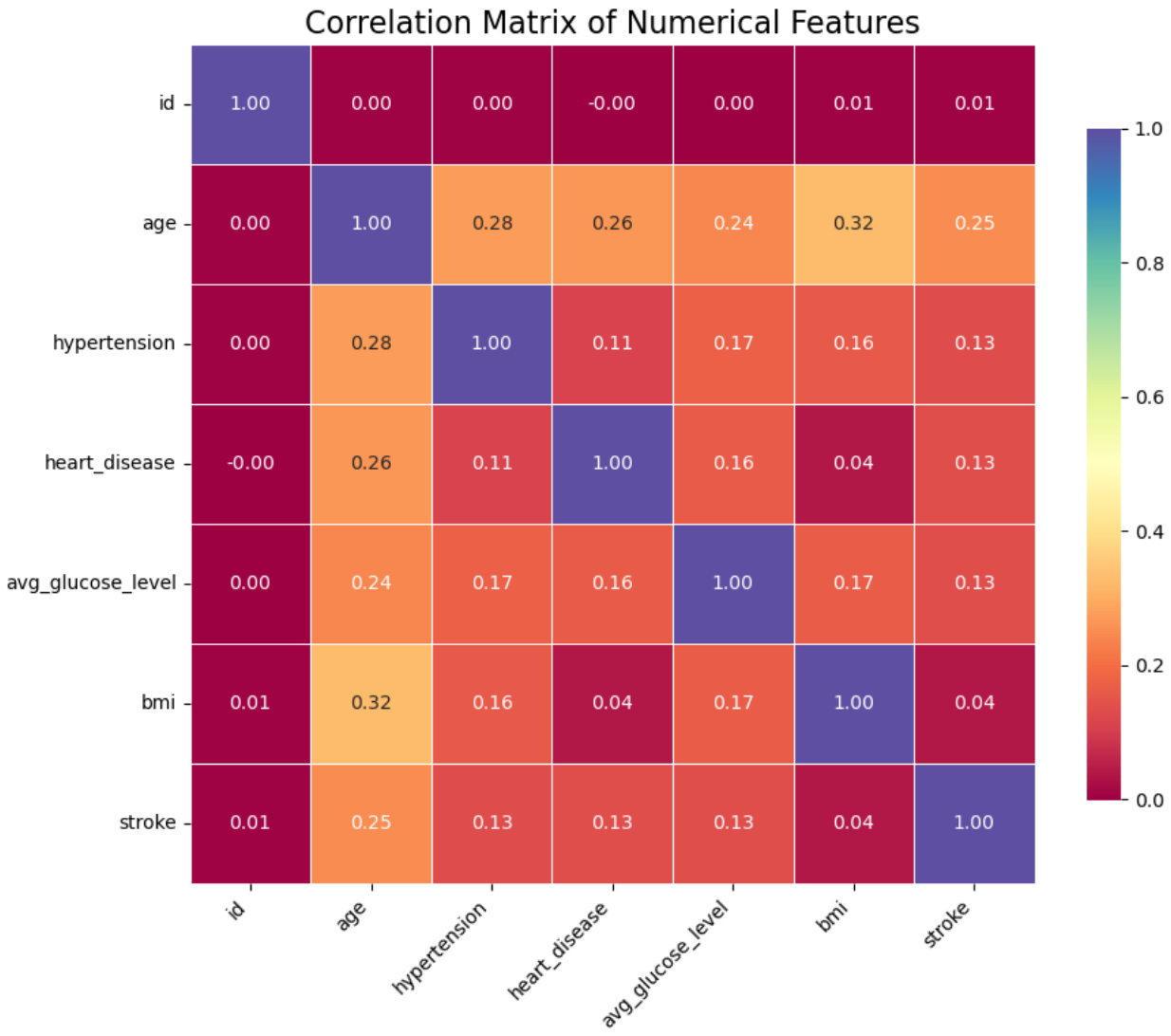
Correlation matrix

So we've gained some understanding on the distributiona of our numeric variables, but we can add more information to this plot.

Let's see how the distribution of our numeric variables is different for those that have strokes, and those that do not.

This could be important for modelling later on

```
# Correlation matrix (numeric columns only)
corr = df.select_dtypes(include=[np.number]).corr()

# Define the colormap to use for the correlation heatmap
CORRELATION_CMAP = "Spectral"

plt.figure(figsize=(10, 8)) # Slightly adjusted for readability
sns.heatmap(data=corr, annot=True, cmap=CORRELATION_CMAP, fmt=".2f", #
Add fmt for consistent decimal places
            linewidths=.5, cbar_kws={"shrink": .8}) # Optional: visual
tweaks
plt.title("Correlation Matrix of Numerical Features",
fontdict={'fontsize': TITLE_FONT_ARGS['fontsize']-2})
plt.xticks(fontsize=TICK_LABEL_FONT_ARGS['fontsize']-2, rotation=45,
ha="right")
plt.yticks(fontsize=TICK_LABEL_FONT_ARGS['fontsize']-2)
plt.show()
```



Each square shows the correlation between the variables on each axis. Correlation ranges from -1 to +1. Values closer to zero means there is no linear trend between the two variables. The close to 1 the correlation is the more positively correlated they are; that is as one increases so does the other and the closer to 1 the stronger this relationship is. A correlation closer to -1 is similar, but instead of both increasing one variable will decrease as the other increases. The diagonals are all 1/dark green because those squares are correlating each variable to itself (so it's a perfect correlation). For the rest the larger the number and darker the color the higher the correlation between the two variables. The plot is also symmetrical about the diagonal since the same two variables are being paired together in those squares.

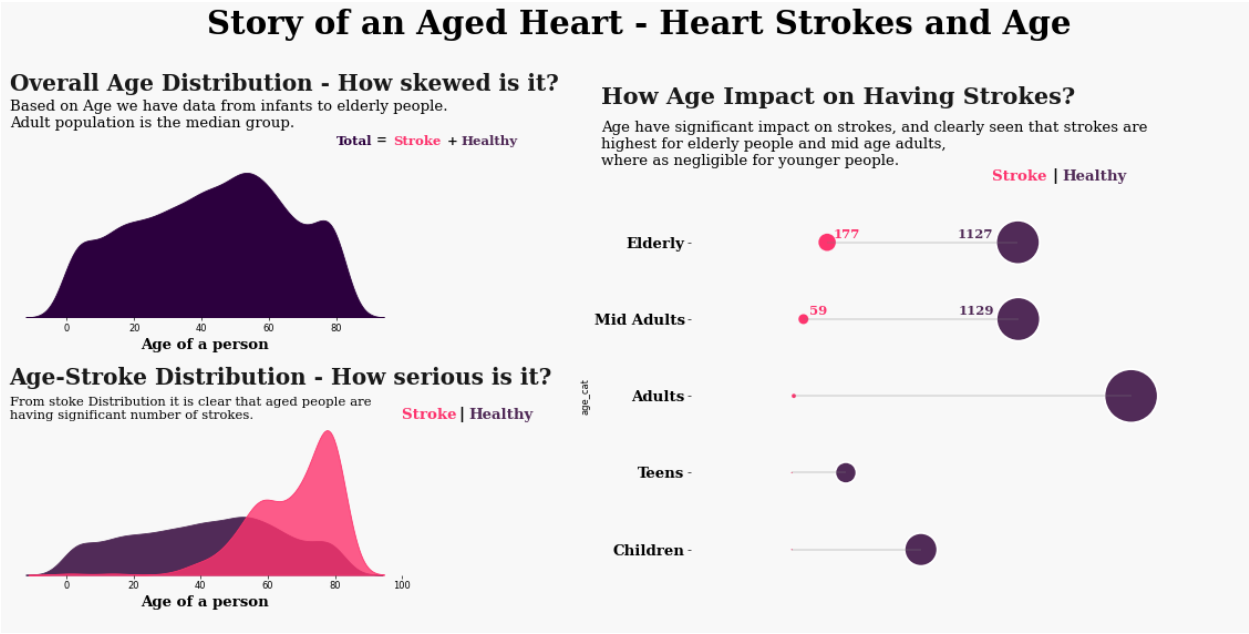
Story of an Aged Heart - Heart Strokes and Age

```
COLOR_BACKGROUND_FIG = "#f8f8f8"  # or any color you prefer

fig = plt.figure(figsize = (24,10), dpi = 60)

gs = fig.add_gridspec(10,24)
```

```
gs.update(wspace = 1, hspace = 0.05)


ax2 = fig.add_subplot(gs[1:4,0:8]) #distribution plot
ax3 = fig.add_subplot(gs[6:9, 0:8]) #hue distribution plot
ax1 = fig.add_subplot(gs[1:10,13:]) #dumbbell plot

# axes list
axes = [ ax1,ax2, ax3]

# setting of axes; visibility of axes and spines turn off
for ax in axes:
    ax.axes.get_yaxis().set_visible(False)
    ax.set_facecolor(COLOR_BACKGROUND_FIG)

    for loc in ['left', 'right', 'top', 'bottom']:
        ax.spines[loc].set_visible(False)

fig.patch.set_facecolor(COLOR_BACKGROUND_FIG)

ax1.axes.get_xaxis().set_visible(False)
ax1.axes.get_yaxis().set_visible(True)

# dumbbell plot of stoke and healthy people

stroke_age = df[df['stroke'] == 1].age_cat.value_counts()
healthy_age = df[df['stroke'] == 0].age_cat.value_counts()

ax1.hlines(y = ['Children', 'Teens', 'Adults', 'Mid Adults',
'Elderly'], xmin = [644,270,1691,1129,1127],
          xmax = [1,1,11,59,177], color = 'grey',**{'linewidth':0.5})


sns.scatterplot(y = stroke_age.index, x = stroke_age.values, s =
stroke_age.values*2, color = '#fe346e', ax= ax1, alpha = 1)
sns.scatterplot(y = healthy_age.index, x = healthy_age.values, s =
healthy_age.values*2, color = '#512b58', ax= ax1, alpha = 1)

ax1.axes.get_xaxis().set_visible(False)
ax1.set_xlim(xmin = -500, xmax = 2250)
ax1.set_ylim(ymin = -1,ymax = 5)

ax1.set_yticklabels( labels = ['Children', 'Teens', 'Adults', 'Mid
Adults', 'Elderly'],fontdict = {'font':'Serif',
'fontsize':16,'fontweight':'bold', 'color':'black'})

ax1.text(-950,5.8, 'How Age Impact on Having Strokes?' ,{'font':
'Serif', 'fontsize': 25,'weight':'bold', 'color':'black'},alpha = 0.9)
ax1.text(1000,4.8, 'Stroke ', {'font':
'Serif','weight':'bold','fontsize':
```

```
16,'weight':'bold','style':'normal', 'color':'#fe346e'})
ax1.text(1300,4.8, '|', {'color':'black' , 'size':'16', 'weight':
'bold'})
ax1.text(1350,4.8, 'Healthy', {'font': 'Serif','weight':'bold',
'fontsize': 16,'style':'normal', 'weight':'bold','color':'#512b58'})
ax1.text(-950,5., 'Age have significant impact on strokes, and clearly
seen that strokes are \nhighest for elderly people and mid age adults,
\nwhere as negligible for younger people.',
        {'font':'Serif', 'size':'16','color': 'black'})

ax1.text(stroke_age.values[0] + 30,4.05, stroke_age.values[0],
{'font':'Serif', 'fontsize':14, 'weight':'bold', 'color':'#fe346e'})
ax1.text(healthy_age.values[2] - 300,4.05, healthy_age.values[2],
{'font':'Serif', 'fontsize':14, 'weight':'bold', 'color':'#512b58'})

ax1.text(stroke_age.values[1] + 30,3.05, stroke_age.values[1],
{'font':'Serif', 'fontsize':14, 'weight':'bold', 'color':'#fe346e'})
ax1.text(healthy_age.values[1] - 300,3.05, healthy_age.values[1],
{'font':'Serif', 'fontsize':14, 'weight':'bold', 'color':'#512b58'})



# distribution plots ---- only single variable

sns.kdeplot(data = df, x = 'age', ax = ax2, shade = True, color =
'#2c003e', alpha = 1, )
ax2.set_xlabel('Age of a person', fontdict = {'font':'Serif', 'color':
'black', 'size': 16,'weight':'bold' })
ax2.text(-17,0.025,'Overall Age Distribution - How skewed is it?',
{'font':'Serif', 'color': 'black','weight':'bold','size':24}, alpha =
0.9)
ax2.text(-17,0.021, 'Based on Age we have data from infants to elderly
people.\nAdult population is the median group.',
        {'font':'Serif', 'size':'16','color': 'black'})
ax2.text(80,0.019, 'Total',{'font':'Serif', 'size':'14','color':
'#2c003e','weight':'bold'})
ax2.text(92,0.019, '=',{'font':'Serif', 'size':'14','color':
'black','weight':'bold'})
ax2.text(97,0.019, 'Stroke',{'font':'Serif', 'size':'14','color':
'#fe346e','weight':'bold'})
ax2.text(113,0.019, '+',{'font':'Serif', 'size':'14','color':
'black','weight':'bold'})
ax2.text(117,0.019, 'Healthy',{'font':'Serif', 'size':'14','color':
'#512b58','weight':'bold'})



# distribution plots with hue of strokes
```

```
sns.kdeplot(data = df[df['stroke'] == 0], x = 'age',ax = ax3, shade = True,  alpha = 1, color = '#512b58' )
sns.kdeplot(data = df[df['stroke'] == 1], x = 'age',ax = ax3, shade = True,  alpha = 0.8, color = '#fe346e')

ax3.set_xlabel('Age of a person', fontdict = {'font':'Serif', 'color': 'black', 'weight':'bold','size': 16})

ax3.text(-17,0.0525,'Age-Stroke Distribution - How serious is it?', {'font':'Serif', 'weight':'bold','color': 'black', 'size':24}, alpha= 0.9)
ax3.text(-17,0.043,'From stoke Distribution it is clear that aged people are \nhaving significant number of strokes.', {'font':'Serif', 'color': 'black', 'size':14})
ax3.text(100,0.043, 'Stroke ', {'font': 'Serif','weight':'bold','fontsize': 16,'weight':'bold','style':'normal', 'color':'#fe346e'})
ax3.text(117,0.043, '|', {'color':'black' , 'fontsize':'16', 'weight': 'bold'})
ax3.text(120,0.043, 'Healthy', {'font': 'Serif','weight':'bold', 'fontsize': 16,'style':'normal', 'weight':'bold','color':'#512b58'})

fig.text(0.25,1,'Story of an Aged Heart - Heart Strokes and Age', {'font':'Serif', 'weight':'bold','color': 'black', 'size':35})
fig.show()
```



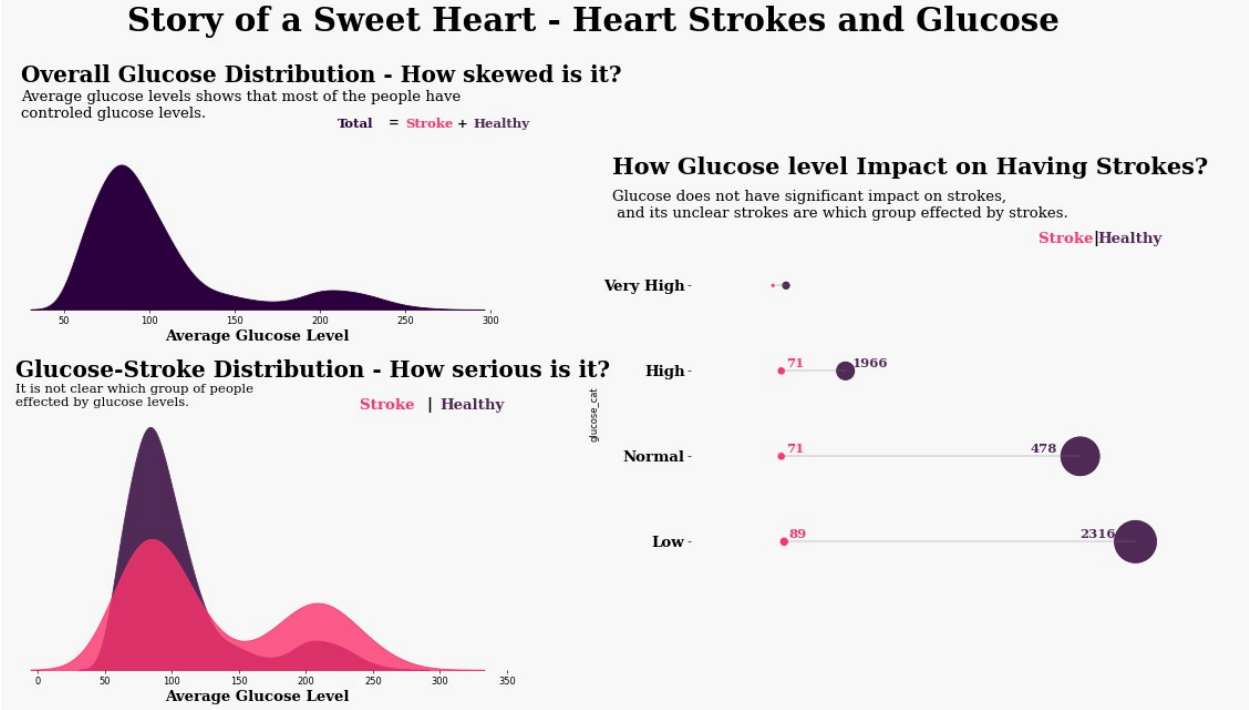
Story of a Sweet Heart - Heart Strokes and Glucose

```
# sugar distribution plots
```

```
fig = plt.figure(figsize = (24,10), dpi = 60)

gs = fig.add_gridspec(10,24)
gs.update(wspace = 1, hspace = 0.05)


ax2 = fig.add_subplot(gs[0:3,0:10]) #distribution plot
ax3 = fig.add_subplot(gs[5:10, 0:10]) #hue distribution plot
ax1 = fig.add_subplot(gs[0:,13:]) #dumbbell plot

# axes list
axes = [ ax1,ax2, ax3]

# setting of axes; visibility of axes and spines turn off
for ax in axes:
    ax.axes.get_yaxis().set_visible(False)
    ax.set_facecolor(COLOR_BACKGROUND_FIG)

    for loc in ['left', 'right', 'top', 'bottom']:
        ax.spines[loc].set_visible(False)

fig.patch.set_facecolor(COLOR_BACKGROUND_FIG)

ax1.axes.get_xaxis().set_visible(False)
ax1.axes.get_yaxis().set_visible(True)

# dumbbell plot of stoke and healthy people

stroke_glu = df[df['stroke'] == 1].glucose_cat.value_counts()
healthy_glu = df[df['stroke'] == 0].glucose_cat.value_counts()

ax1.hlines(y = ['Low', 'Normal', 'High', 'Very High'], xmin =
[2316,1966,478,101],
          xmax = [89,71,71,18], color = 'grey',**{'linewidth':0.5})


sns.scatterplot(y = stroke_glu.index, x = stroke_glu.values, s =
stroke_glu.values, color = '#fe346e', ax= ax1, alpha = 1)
sns.scatterplot(y = healthy_glu.index, x = healthy_glu.values, s =
healthy_glu.values, color = '#512b58', ax= ax1, alpha = 1)

ax1.axes.get_xaxis().set_visible(False)
ax1.set_xlim(xmin = -500, xmax = 3000)
ax1.set_ylim(ymin = -1.5,ymax = 4.5)

ax1.set_yticklabels( labels = ['Low', 'Normal', 'High', 'Very
High'],fontdict = {'font':'Serif', 'fontsize':16,'fontweight':'bold',
'color':'black'})

ax1.text(-1000,4.3, 'How Glucose level Impact on Having Strokes?' ,
```

```
{'font': 'Serif', 'fontsize': 25,'weight':'bold', 'color':'black'})
ax1.text(1700,3.5, 'Stroke ', {'font': 'Serif','weight':'bold',
'fontsize': 16,'weight':'bold','style':'normal', 'color':'#fe346e'})
ax1.text(2050,3.5, '|', {'color':'black' , 'size':'16', 'weight':
'bold'})
ax1.text(2075,3.5, 'Healthy', {'font': 'Serif','weight':'bold',
'fontsize': 16,'style':'normal', 'weight':'bold','color':'#512b58'})
ax1.text(-1000,3.8, 'Glucose does not have significant impact on
strokes,\n and its unclear strokes are which group effected by
strokes.',
        {'font':'Serif', 'fontsize':'16','color': 'black'})

ax1.text(stroke_glu.values[0] + 30,0.05, stroke_glu.values[0],
{'font':'Serif', 'fontsize':14, 'weight':'bold', 'color':'#fe346e'})
ax1.text(healthy_glu.values[0] + -355,0.05, healthy_glu.values[0],
{'font':'Serif', 'fontsize':14, 'weight':'bold', 'color':'#512b58'})

ax1.text(stroke_glu.values[2] + 30,1.05, stroke_glu.values[2],
{'font':'Serif', 'fontsize':14, 'weight':'bold', 'color':'#fe346e'})
ax1.text(healthy_glu.values[2] + 1170,1.05, healthy_glu.values[2],
{'font':'Serif', 'fontsize':14, 'weight':'bold', 'color':'#512b58'})

ax1.text(stroke_glu.values[1] + 30,2.05, stroke_glu.values[1],
{'font':'Serif', 'fontsize':14, 'weight':'bold', 'color':'#fe346e'})
ax1.text(healthy_glu.values[1] - 1450,2.05, healthy_glu.values[1],
{'font':'Serif', 'fontsize':14, 'weight':'bold', 'color':'#512b58'})



# distribution plots ---- only single variable

sns.kdeplot(data = df, x = 'avg_glucose_level', ax = ax2, shade =
True, color = '#2c003e', alpha = 1, )
ax2.set_xlabel('Average Glucose Level', fontdict = {'font':'Serif',
'color': 'black', 'size': 16,'weight':'bold' })
ax2.text(25,0.025,'Overall Glucose Distribution - How skewed is it?',
{'font':'Serif', 'color': 'black','weight':'bold','size':24})
ax2.text(25,0.021, 'Average glucose levels shows that most of the
people have \ncontroled glucose levels.',
        {'font':'Serif', 'size':'16','color': 'black'})
ax2.text(210,0.020, 'Total',{'font':'Serif', 'size':'14','color':
'#2c003e','weight':'bold'})
ax2.text(240,0.02, '=',{'font':'Serif', 'size':'14','color':
'black','weight':'bold'})
ax2.text(250,0.02, 'Stroke',{'font':'Serif', 'size':'14','color':
'#fe346e','weight':'bold'})
ax2.text(280,0.02, '+',{'font':'Serif', 'size':'14','color':
'black','weight':'bold'})
ax2.text(290,0.02, 'Healthy',{'font':'Serif', 'size':'14','color':
'#512b58','weight':'bold'})
```

```
# distribution plots with hue of strokes


sns.kdeplot(data = df[df['stroke'] == 0], x = 'avg_glucose_level',ax =
ax3, shade = True,  alpha = 1, color = '#512b58' )
sns.kdeplot(data = df[df['stroke'] == 1], x = 'avg_glucose_level',ax =
ax3, shade = True,  alpha = 0.8, color = '#fe346e')

ax3.set_xlabel('Average Glucose Level', fontdict = {'font':'Serif',
'color': 'black', 'weight':'bold','size': 16})

ax3.text(-17,0.0195,'Glucose-Stroke Distribution - How serious is
it?', {'font':'Serif', 'weight':'bold','color': 'black', 'size':24})
ax3.text(-17,0.0176,'It is not clear which group of people \neffected
by glucose levels.', {'font':'Serif', 'color': 'black', 'size':14})
ax3.text(240,0.0174, 'Stroke ', {'font': 'Serif','weight':'bold',
'fontsize': 16,'weight':'bold','style':'normal', 'color':'#fe346e'})
ax3.text(290,0.0174, '|', {'color':'black' , 'size':'16', 'weight':
'bold'})
ax3.text(300,0.0174, 'Healthy', {'font': 'Serif','weight':'bold',
'fontsize': 16,'style':'normal', 'weight':'bold','color':'#512b58'})


fig.text(0.2,1.07,'Story of a Sweet Heart - Heart Strokes and
Glucose',{'font':'Serif', 'weight':'bold','color': 'black',
'size':35})

fig.show()
```

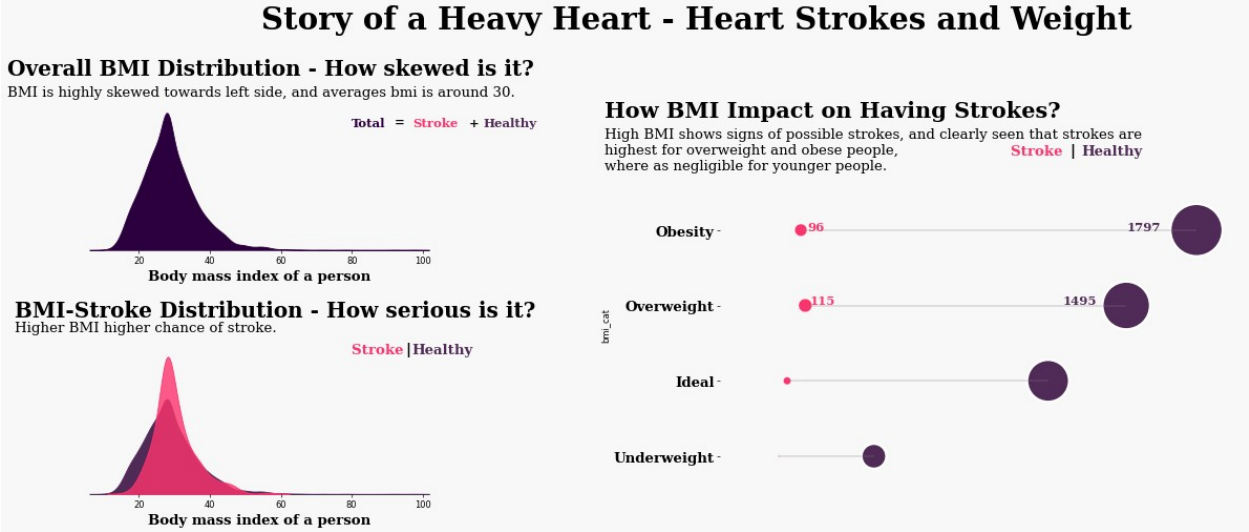


Story of a Heavy Heart - Heart Strokes and Weight

```
import matplotlib.pyplot as plt
import seaborn as sns

fig = plt.figure(figsize = (24,10),dpi = 60)

gs = fig.add_gridspec(10,24)
gs.update(wspace = 1, hspace = 0.05)


ax2 = fig.add_subplot(gs[1:4,0:8]) #distribution plot
ax3 = fig.add_subplot(gs[6:9, 0:8]) #hue distribution plot
ax1 = fig.add_subplot(gs[2:9,13:]) #dumbbell plot

# axes list
axes = [ ax1,ax2, ax3]

# setting of axes; visibility of axes and spines turn off
for ax in axes:
    ax.axes.get_yaxis().set_visible(False)
    ax.set_facecolor(COLOR_BACKGROUND_FIG)

    for loc in ['left', 'right', 'top', 'bottom']:
        ax.spines[loc].set_visible(False)

fig.patch.set_facecolor(COLOR_BACKGROUND_FIG)

ax1.axes.get_xaxis().set_visible(False)
ax1.axes.get_yaxis().set_visible(True)
```

```
ax1.set_xlim(xmin = -250,xmax = 2000)
ax1.set_ylim(ymin = -1,ymax =3.5)

# dumbbell plot of stoke and healthy people

stroke_bmi = df[df['stroke'] == 1].bmi_cat.value_counts()
healthy_bmi = df[df['stroke'] == 0].bmi_cat.value_counts()

ax1.hlines(y = ['Obesity', 'Overweight', 'Ideal', 'Underweight'], xmin
= [96,115,37,1],
          xmax = [1797,1495,1159,410], color =
'grey',**{'linewidth':0.5})


sns.scatterplot(y = stroke_bmi.index, x = stroke_bmi.values, s =
stroke_bmi.values*2, color = '#fe346e', ax= ax1, alpha = 1)
sns.scatterplot(y = healthy_bmi.index, x = healthy_bmi.values, s =
healthy_bmi.values*2, color = '#512b58', ax= ax1, alpha = 1)

ax1.set_yticklabels( labels = ['Obesity', 'Overweight', 'Ideal',
'Underweight'],fontdict = {'font':'Serif',
'fontsize':16,'fontweight':'bold', 'color':'black'})

ax1.text(-750,-1.5, 'How BMI Impact on Having Strokes?' ,{'font':
'Serif', 'size': 25,'weight':'bold', 'color':'black'})
ax1.text(1000,-1., 'Stroke ', {'font': 'Serif','weight':'bold','size':
16,'weight':'bold','style':'normal', 'color':'#fe346e'})
ax1.text(1250,-1, '|', {'color':'black' , 'size':16, 'weight':
'bold'})
ax1.text(1300,-1, 'Healthy', {'font': 'Serif','weight':'bold', 'size':
16,'style':'normal', 'weight':'bold','color':'#512b58'})
ax1.text(-750,-0.8, 'High BMI shows signs of possible strokes, and
clearly seen that strokes are \nhighest for overweight and obese
people, \nwhere as negligible for younger people.',
        {'font':'Serif', 'size':'16','color': 'black'})



ax1.text(stroke_bmi.values[0] + 20 , 0.98, stroke_bmi.values[0],
{'font':'Serif', 'size':14, 'weight':'bold', 'color':'#fe346e'})
ax1.text(healthy_bmi.values[1] - 275 ,0.98, healthy_bmi.values[1],
{'font':'Serif', 'size':14, 'weight':'bold', 'color':'#512b58'})

ax1.text(stroke_bmi.values[1] + 30,0, stroke_bmi.values[1],
{'font':'Serif', 'size':14, 'weight':'bold', 'color':'#fe346e'})
ax1.text(healthy_bmi.values[0] - 300,0, healthy_bmi.values[0],
{'font':'Serif', 'size':14, 'weight':'bold', 'color':'#512b58'})
```

```
# distribution plots ---- only single variable

sns.kdeplot(data = df, x = 'bmi', ax = ax2, shade = True, color =
'#2c003e', alpha = 1, )
ax2.set_xlabel('Body mass index of a person', fontdict =
{'font':'Serif', 'color': 'black', 'size': 16,'weight':'bold' })
ax2.text(-17,0.085,'Overall BMI Distribution - How skewed is it?',
{'font':'Serif', 'color': 'black','weight':'bold','size':24})
ax2.text(-17,0.075, 'BMI is highly skewed towards left side, and
averages bmi is around 30.',
        {'font':'Serif', 'size':'16','color': 'black'})
ax2.text(80,0.06, 'Total',{'font':'Serif', 'size':'14','color':
'#2c003e','weight':'bold'})
ax2.text(92,0.06, '=',{'font':'Serif', 'size':'14','color':
'black','weight':'bold'})
ax2.text(97,0.06, 'Stroke',{'font':'Serif', 'size':'14','color':
'#fe346e','weight':'bold'})
ax2.text(113,0.06, '+',{'font':'Serif', 'size':'14','color':
'black','weight':'bold'})
ax2.text(117,0.06, 'Healthy',{'font':'Serif', 'size':'14','color':
'#512b58','weight':'bold'})


# distribution plots with hue of strokes


sns.kdeplot(data = df[df['stroke'] == 0], x = 'bmi',ax = ax3, shade =
True,  alpha = 1, color = '#512b58' )
sns.kdeplot(data = df[df['stroke'] == 1], x = 'bmi',ax = ax3, shade =
True,  alpha = 0.8, color = '#fe346e')

ax3.set_xlabel('Body mass index of a person', fontdict =
{'font':'Serif', 'color': 'black', 'weight':'bold','size': 16})

ax3.text(-15,0.12,'BMI-Stroke Distribution - How serious is it?',
{'font':'Serif', 'weight':'bold','color': 'black', 'size':24})
ax3.text(-15,0.11,'Higher BMI higher chance of stroke.',
{'font':'Serif', 'color': 'black', 'size':16})
ax3.text(80,0.095, 'Stroke ', {'font': 'Serif','weight':'bold','size':
16,'weight':'bold','style':'normal', 'color':'#fe346e'})
ax3.text(95,0.095, '|', {'color':'black' , 'size':16, 'weight':
'bold'})
ax3.text(97,0.095, 'Healthy', {'font': 'Serif','weight':'bold',
'size': 16,'style':'normal', 'weight':'bold','color':'#512b58'})

fig.text(0.25,0.925,'Story of a Heavy Heart - Heart Strokes and
Weight',{'font':'Serif', 'weight':'bold','color': 'black', 'size':35})

fig.show()
```



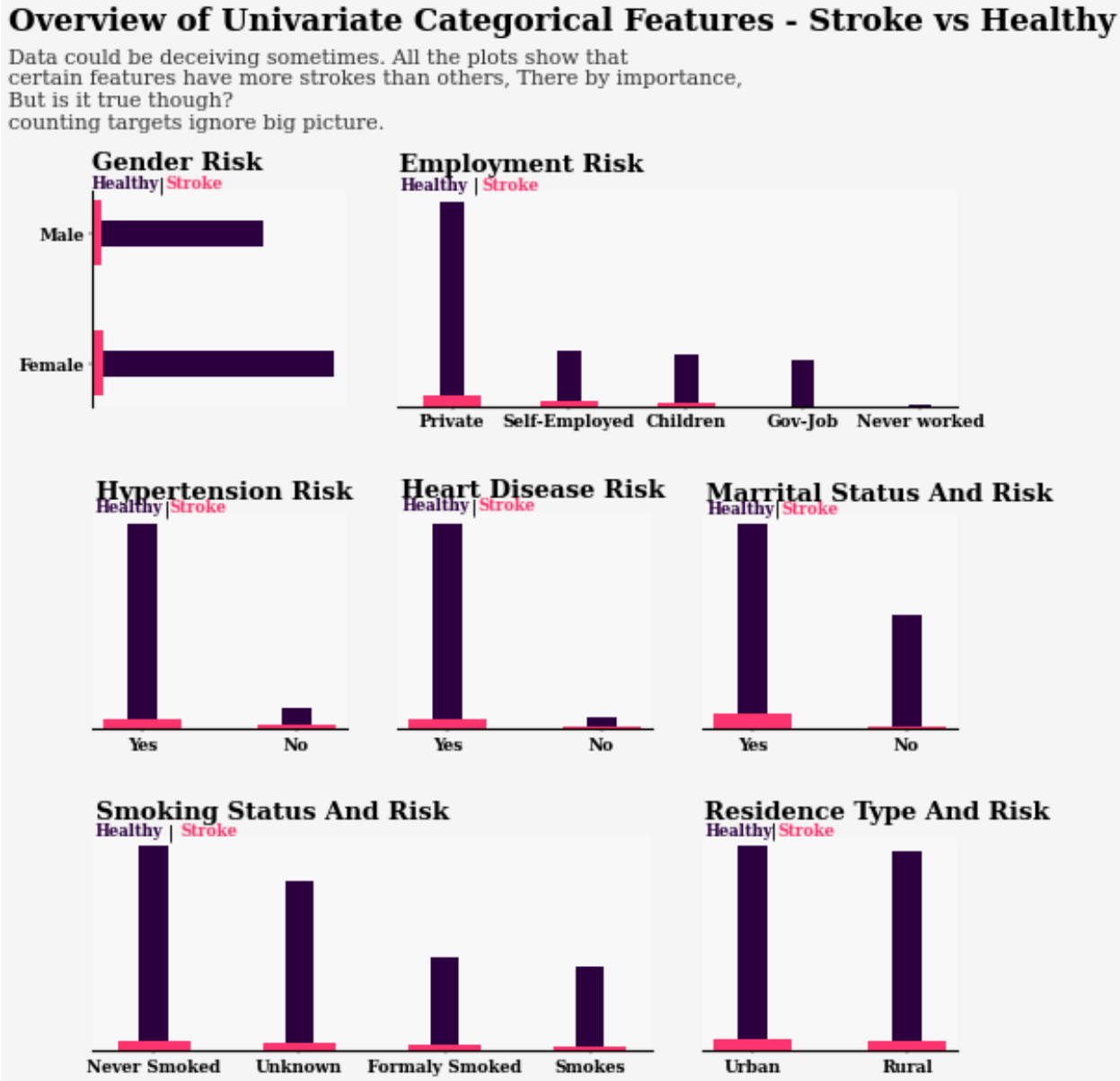
Overview of categorical features

```
fig = plt.figure(figsize = (15,15),dpi = 40)

gs = fig.add_gridspec(3,3)
gs.update(wspace = 0.2, hspace = 0.5)


ax1 = fig.add_subplot(gs[0,0])
ax2 = fig.add_subplot(gs[0,1:])
ax3 = fig.add_subplot(gs[1,0])
ax4 = fig.add_subplot(gs[1,1])
ax5 = fig.add_subplot(gs[1,2])
ax6 = fig.add_subplot(gs[2,0:2])
ax7 = fig.add_subplot(gs[2,2])

axes = [ax1, ax2, ax3, ax4, ax5, ax6, ax7]

fig.patch.set_facecolor('#f5f5f5')

# setting of axes; visibility of axes and spines turn off
for ax in axes:
    ax.axes.get_yaxis().set_visible(False)
    ax.set_facecolor('#f8f8f8')
    ax.spines['bottom'].set_linewidth(2)
    for loc in ['left', 'right', 'top']:
        ax.spines[loc].set_visible(False)
        ax.spines[loc].set_linewidth(2)


title_args = {'font':'Serif', 'weight':'bold','color': 'black',
'size':24}
font_dict = {'size':16, 'family':'Serif', 'color':'black',
'weight':'bold'}
health_dict = {'font':'Serif', 'color': '#2c003e', 'size':15,
```

```
'weight':'bold'}
dash_dict = {'font':'Serif', 'color': 'black',
'size':15,'weight':'bold'}
stroke_dict = {'font':'Serif', 'color': '#fe346e',
'size':15,'weight':'bold'}

stroke_col = '#fe346e'
healthy_col = '#2c003e'

# Ax1: Gender- stroke distributions
healthy_gen = df[df['stroke'] == 0].gender.value_counts()
stroke_gen = df[df['stroke'] == 1].gender.value_counts()

ax1.barh( stroke_gen.index , width = healthy_gen.values[0:2], height =
0.2, color = healthy_col)
ax1.barh( np.arange(len(stroke_gen.index)) , width =
stroke_gen.values, height = 0.5, color = stroke_col)
ax1.set_yticklabels(stroke_gen.index, **font_dict)

ax1.axes.get_yaxis().set_visible(True)
ax1.axes.get_xaxis().set_visible(False)
ax1.spines['bottom'].set_visible(False)
ax1.spines['left'].set_visible(True)
ax1.text(0,1.5, 'Gender Risk',**title_args)
ax1.text(0,1.35, 'Healthy',**health_dict)
ax1.text(790,1.35, '|',**dash_dict)
ax1.text(870,1.35, 'Stroke',**stroke_dict)

# Ax2: work type - stroke distributions
healthy_gen = df[df['stroke'] == 0].work_type.value_counts()
stroke_gen = df[df['stroke'] == 1].work_type.value_counts()

ax2.bar( healthy_gen.index , height = healthy_gen.values, width = 0.2,
color = healthy_col)
ax2.bar( np.arange(len(stroke_gen.index)) , height =
stroke_gen.values, width = 0.5, color= stroke_col)
ax2.set_xticklabels(['Private','Self-Employed','Children', 'Gov-
Job','Never worked'], **font_dict)


ax2.text(-0.45,3200, 'Employment Risk',**title_args)
ax2.text(-0.45,2950, 'Healthy',**health_dict)
ax2.text(0.18,2950, '|',**dash_dict)
ax2.text(0.25,2950, 'Stroke',**stroke_dict)

# Ax3: hypertension - stroke distributions

healthy_gen = df[df['stroke'] == 0].hypertension.value_counts()
stroke_gen = df[df['stroke'] == 1].hypertension.value_counts()
```

```
ax3.bar(['Yes','No'] , height = healthy_gen.values, width = 0.2,color
= healthy_col)
ax3.bar( stroke_gen.index, height = stroke_gen.values, width =
0.5,color= stroke_col)
ax3.set_xticklabels(['Yes','No'], **font_dict)

ax3.text(-0.3,5000, 'Hypertension Risk',**title_args)
ax3.text(-0.3,4700, 'Healthy',**health_dict)
ax3.text(0.14,4700, '|',**dash_dict)
ax3.text(0.18,4700, 'Stroke',**stroke_dict)


# Ax4: Heart Disease - stroke distributions

healthy_gen = df[df['stroke'] == 0].heart_disease.value_counts()
stroke_gen = df[df['stroke'] == 1].heart_disease.value_counts()

ax4.bar(['Yes','No'] , height = healthy_gen.values, width = 0.2,color
= healthy_col)
ax4.bar( stroke_gen.index, height = stroke_gen.values, width =
0.5,color= stroke_col)
ax4.set_xticklabels(['Yes', 'No'],**font_dict)

ax4.text(-0.3,5250, 'Heart Disease Risk',**title_args)
ax4.text(-0.3,4950, 'Healthy',**health_dict)
ax4.text(0.15,4950, '|',**dash_dict)
ax4.text(0.20,4950, 'Stroke',**stroke_dict)


# Ax5: Married - stroke distributions
healthy_gen = df[df['stroke'] == 0].ever_married.value_counts()
stroke_gen = df[df['stroke'] == 1].ever_married.value_counts()

ax5.bar( healthy_gen.index , height = healthy_gen.values, width =
0.2,color = healthy_col)
ax5.bar( np.arange(len(stroke_gen.index)) , height =
stroke_gen.values, width = 0.5,color= stroke_col )
ax5.set_xticklabels(healthy_gen.index, **font_dict)

ax5.text(-0.3,3500, 'Marrital Status And Risk',**title_args)
ax5.text(-0.3,3300, 'Healthy',**health_dict)
ax5.text(0.14,3300, '|',**dash_dict)
ax5.text(0.18,3300, 'Stroke',**stroke_dict)




# Ax6: Smoking status - stroke distributions

healthy_gen = df[df['stroke'] == 0].smoking_status.value_counts()
stroke_gen = df[df['stroke'] == 1].smoking_status.value_counts()
```

```
ax6.bar( healthy_gen.index, height = healthy_gen.values, width =
0.2,color = healthy_col)
ax6.bar( np.arange(len(stroke_gen.index)) , height =
stroke_gen.values, width = 0.5,color= stroke_col)
ax6.set_xticklabels(['Never Smoked', 'Unknown','Formaly
Smoked' ,'Smokes'], **font_dict)

ax6.text(-0.4,2050, 'Smoking Status And Risk',**title_args)
ax6.text(-0.4,1900, 'Healthy',**health_dict)
ax6.text(0.095,1900, '|',**dash_dict)
ax6.text(0.18,1900, 'Stroke',**stroke_dict)



# Ax7: Residence type - stroke distributions

healthy_gen = df[df['stroke'] == 0].Residence_type.value_counts()
stroke_gen = df[df['stroke'] == 1].Residence_type.value_counts()

ax7.bar( healthy_gen.index , height = healthy_gen.values, width =
0.2,color = healthy_col)
ax7.bar( np.arange(len(stroke_gen.index)) , height =
stroke_gen.values, width = 0.5,color= stroke_col)
ax7.set_xticklabels(healthy_gen.index, **font_dict)

ax7.text(-0.31,2800, 'Residence Type And Risk',**title_args)
ax7.text(-0.31,2600, 'Healthy',**health_dict)
ax7.text(0.12,2600,'|',**dash_dict)
ax7.text(0.165,2600, 'Stroke',**stroke_dict)

fig.text(0.05,1.025, 'Overview of Univariate Categorical Features -
Stroke vs Healthy', {'font':'Serif', 'color':'black','size':30,
'weight':'bold'})
fig.text(0.05,0.9375,'Data could be deceiving sometimes. All the plots
show that\ncertain features have more strokes than others, There by
importance, \nBut is it true though? \ncounting targets ignore big
picture.',{'font':'Serif', 'color':'black','size':20,
'weight':'normal'}, alpha = 0.8)

fig.show()
```



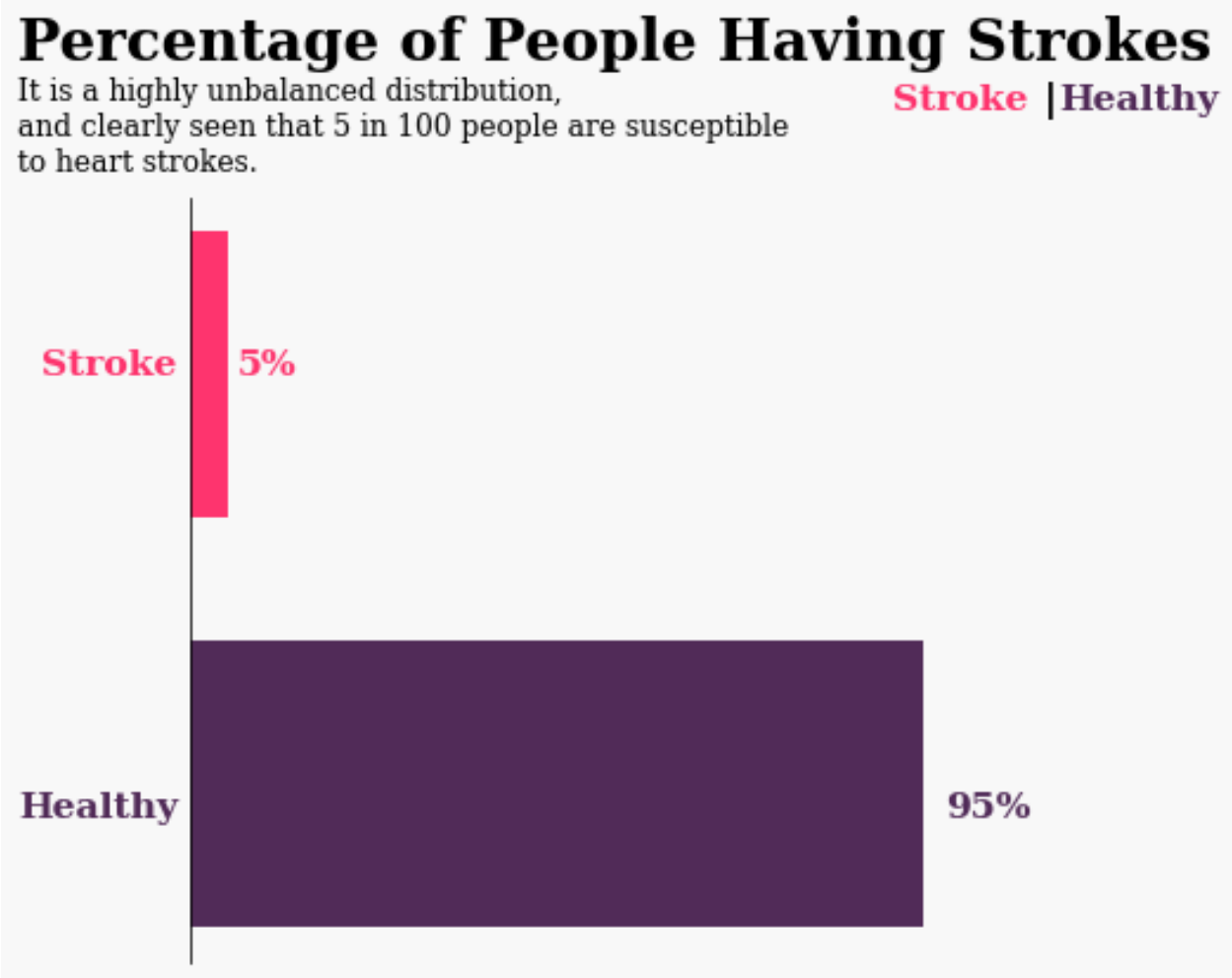
Target Variable: `stroke`

This plot will show us the distribution of our target variable and reveal if we have a class imbalance problem.

```
x = pd.DataFrame( df.groupby(['stroke'])['stroke'].count())

# plot
fig, ax = plt.subplots(figsize = (6,6), dpi = 70)
ax.barh([1], x.stroke[1], height = 0.7, color = '#fe346e')
plt.text(-1150, -0.08, 'Healthy', fontname='Serif', weight='bold',
fontsize=16, style='normal', color='#512b58')
plt.text(5000, -0.08, '95%', fontname='Serif', weight='bold',
fontsize=16, color='#512b58')
```

```
ax.barh([0], x.stroke[0], height=0.7, color='#512b58')
plt.text(-1000, 1, 'Stroke', fontname='Serif', weight='bold',
fontsize=16, style='normal', color='#fe346e')
plt.text(300, 1, '5%', fontname='Serif', weight='bold', fontsize=16,
color='#fe346e')

fig.patch.set_facecolor(COLOR_BACKGROUND_FIG)
ax.set_facecolor(COLOR_BACKGROUND_FIG)

plt.text(-1150, 1.77, 'Percentage of People Having Strokes',
fontname='Serif', fontsize=25, weight='bold', color='black')
plt.text(4650, 1.65, 'Stroke ', fontname='Serif', weight='bold',
fontsize=16, style='normal', color='#fe346e')
plt.text(5650, 1.65, '|', color='black', fontsize=16, weight='bold')
plt.text(5750, 1.65, 'Healthy', fontname='Serif', weight='bold',
fontsize=16, style='normal', color='#512b58')
plt.text(-1150, 1.5, 'It is a highly unbalanced distribution,\nand
clearly seen that 5 in 100 people are susceptible \nto heart
strokes.',
         fontname='Serif', fontsize=12.5, color='black')

ax.axes.get_xaxis().set_visible(False)
ax.axes.get_yaxis().set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['left'].set_visible(True)
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
# Calculate and print the exact percentage
stroke_percentage = df['stroke'].value_counts(normalize=True) * 100
print("Percentage of each class in the 'stroke' column:")
print(stroke_percentage)

print("\nObservation: The dataset is highly imbalanced, with strokes
accounting for less than 5% of all cases.")
print("This justifies our later use of SMOTE to balance the training
data.")
```

```
Percentage of each class in the 'stroke' column:
stroke
0    95.127202
1     4.872798
Name: proportion, dtype: float64

Observation: The dataset is highly imbalanced, with strokes accounting
for less than 5% of all cases.
This justifies our later use of SMOTE to balance the training data.
```




```
df.drop(columns=['glucose_cat', 'bmi_cat', 'age_cat'], inplace=True)
```

5. Data Preprocessing

5.1. Handle Rare Categorical Values

The gender column has one "Other" value. It's cleanest to remove this single row before proceeding

```
print("Original shape of the dataframe:", df.shape)

# Step 1: Drop the 'id' column as it's not a predictive feature
df = df.drop('id', axis=1)

# Step 2: Remove the 'Other' gender category if it exists
if 'Other' in df['gender'].unique():
    df = df[df['gender'] != 'Other']
```

```
#print("Shape after dropping ID and 'Other' gender:", df.shape)
df.head()
```

```
Original shape of the dataframe: (5110, 12)

   gender   age  hypertension  heart_disease ever_married
work_type  \
0    Male  67.0             0              1          Yes
Private
1  Female  61.0             0              0          Yes  Self-
employed
2    Male  80.0             0              1          Yes
Private
3  Female  49.0             0              0          Yes
Private
4  Female  79.0             1              0          Yes  Self-
employed

  Residence_type  avg_glucose_level   bmi   smoking_status  stroke
0          Urban             228.69  36.6  formerly smoked       1
1          Rural             202.21  28.1     never smoked       1
2          Rural             105.92  32.5     never smoked       1
3          Urban             171.23  34.4           smokes       1
4          Rural             174.12  24.0     never smoked       1
```

5.2. Categorical Feature Encoding

Machine learning models require all input features to be numeric. We will use **One-Hot Encoding** to convert categorical columns into a numerical format. The `id` column is just an identifier and is not useful for prediction, so we will drop it.

```
#strategy 1
# Use pandas get_dummies to perform one-hot encoding on categorical
features
# Convert Marital Status, Residence and Gender into 0's and 1's
from sklearn.preprocessing import LabelEncoder
lb=LabelEncoder()
cat_columns=['gender','ever_married', 'Residence_type',]
for col in cat_columns:
    df[col]=lb.fit_transform(df[col])

df.head()
# used One Hot encoding smoking_status, work_type

data_dummies = df[['smoking_status','work_type']]
data_dummies=pd.get_dummies(data_dummies ,dtype=int)
data_dummies.astype('int64')
df.drop(columns=['smoking_status','work_type'],inplace=True)

y=df['stroke']
```

```
df.drop(columns=['stroke'],inplace=True)
x=df.merge(data_dummies,left_index=True, right_index=True,how='left')
df_processed=x.merge(y,left_index=True, right_index=True,how='left')

print("\nShape of the dataframe after one-hot encoding:",
df_processed.shape)
print("\nFirst 5 rows of the processed dataframe:")
df_processed.head()
#df_processed.info()


#stategy 2:
'''train_data_cat = df.select_dtypes("object")
train_data_num = df.select_dtypes("number")

train_data_cata_encoded=pd.get_dummies(train_data_cat,
columns=train_data_cat.columns.to_list())
train_data_cata_encoded.head()
df_processed=pd.concat([train_data_cata_encoded,train_data_num],axis=1
,join="outer")
df_processed'''


Shape of the dataframe after one-hot encoding: (5109, 18)

First 5 rows of the processed dataframe:

'train_data_cat = df.select_dtypes("object")\ntrain_data_num =
df.select_dtypes("number")\n\
ntrain_data_cata_encoded=pd.get_dummies(train_data_cat,
columns=train_data_cat.columns.to_list())\
ntrain_data_cata_encoded.head()\
ndf_processed=pd.concat([train_data_cata_encoded,train_data_num],axis=
1,join="outer")\ndf_processed'
```

5.3. Separating Features (X) and Target (y)

```
# 5.2. Define Features (X) and Target (y)

X = df_processed.drop('stroke', axis=1)
y = df_processed['stroke']

print("Shape of our features (X):", X.shape)
print("Shape of our target (y):", y.shape)

#Scaling the features
sc = StandardScaler()
X=pd.DataFrame(sc.fit_transform(X), columns=X.columns)
```

```
Shape of our features (X): (5109, 17)
Shape of our target (y): (5109,)
```

5.4. Train-Test Split

We split the data into a training set (to build the model) and a testing set (to evaluate the model on unseen data).

```
# Split the data, with 80% for training and 20% for testing
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,    # 20% of data will be for testing
    random_state=42,  # Ensures the split is the same every time we
run
    stratify=y        # ESSENTIAL: Keeps the stroke proportion the
same in train and test
)

print("--- Data Split ---")
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)

# Get the value representing the positive class (originally 1 before
scaling)
positive_class = y.max()
print("\nProportion of stroke cases in original data:   ",
y.value_counts(normalize=True)[positive_class])
print("Proportion of stroke cases in training data: ",
y_train.value_counts(normalize=True)[positive_class])
print("Proportion of stroke cases in testing data:  ",
y_test.value_counts(normalize=True)[positive_class])
```

```
--- Data Split ---
X_train shape: (4087, 17)
X_test shape: (1022, 17)

Proportion of stroke cases in original data:    0.04873752201996477
Proportion of stroke cases in training data:  0.04869097137264497
Proportion of stroke cases in testing data:   0.04892367906066536
```

6. Modeling Approaches for Imbalanced Data

- In this core part of our project, we will explore different strategies to build predictive models on our imbalanced stroke dataset. We'll first attempt to train models without explicit oversampling techniques, relying on model parameters or algorithms robust to imbalance. Subsequently, we will apply the SMOTE (Synthetic Minority Over-sampling Technique) to the training data to create a balanced dataset and compare the performance. We will also investigate the critical issue of data leakage when

oversampling is misapplied. Finally, we'll develop a neural network model. Our process for each classification model will generally involve:

1. Initializing the model.
2. Training the model on the appropriately prepared training data (either original or with SMOTE).
3. Testing its performance on the original, unbalanced test data (X_test, y_test), as this represents the real-world scenario.
4. Evaluating it using our chosen metrics suitable for imbalanced datasets: F1-Score, Recall, Precision, and ROC-AUC Score.
5. Storing the results for a final comparison.

6.1. Model Training and Evaluation (NO SMOTE)

In this first modeling approach, we will train several classification algorithms without applying SMOTE to the training data. The objective is to establish a baseline and see how well standard models perform on the imbalanced dataset. We may explore using class weights if the algorithm supports it or choose models that are inherently more robust to class imbalance.

```
# Create a dictionary to store the results of each model
results = {}
```

Utility functions

```
import matplotlib.pyplot as plt
from sklearn.metrics import (
    confusion_matrix,
    ConfusionMatrixDisplay,
    classification_report,
    f1_score,
    precision_score,
    recall_score,
    roc_auc_score,
    RocCurveDisplay,
    balanced_accuracy_score
)


def plot_confusion(y_true, y_pred, ax=None):
    """
    Plot a confusion matrix.

    Parameters
    ----------
    y_true : array-like
        True class labels.
    y_pred : array-like
        Predicted class labels.
    labels : list of str, optional
        Names of the target classes (default: inferred).
```

```
    ax : matplotlib Axes, optional
        Axes object to draw the plot into, otherwise creates a new
one.

    Returns
    -------
    ax : matplotlib Axes
    """
    labels=labels=["No Stroke", "Stroke"]
    cm = confusion_matrix(y_true, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=labels)
    return disp.plot(ax=ax)


def plot_roc(y_true, y_score, ax=None):
    """
    Plot ROC curve with AUC.

    Parameters
    ----------
    y_true : array-like
        True binary labels.
    y_score : array-like
        Target scores, can either be probability estimates of the
positive class,
        confidence values, or non-thresholded measure of decisions.
    ax : matplotlib Axes, optional
        Axes object to draw the plot into, otherwise creates a new
one.

    Returns
    -------
    ax : matplotlib Axes
    """
    roc_disp = RocCurveDisplay.from_predictions(y_true, y_score,
ax=ax)
    return ax or plt.gca()


def evaluate_model(
    clf,
    X_test,
    y_test,
    name: str = None,
    results: dict = None,
    labels: list = None,
    show_plots: bool = True
):
    """
```

```
    Evaluate a fitted classifier on test data and optionally store
metrics.

    Calculates balanced accuracy, classification report, confusion
matrix,
    ROC AUC, precision, recall, and F1-score.

    Parameters
    ----------
    clf : estimator
        Fitted classifier with predict and predict_proba methods.
    X_test : array-like
        Test features.
    y_test : array-like
        True labels for X_test.
    name : str, optional
        Key under which to store metrics in results.
    results : dict, optional
        Dictionary to update with computed metrics.
    labels : list of str, optional
        Class names for display (default: inferred).
    show_plots : bool, default=True
        Whether to display confusion matrix and ROC curve.

    Returns
    -------
    metrics : dict
        Dictionary with keys 'Balanced Accuracy', 'Precision',
'Recall', 'F1', 'ROC AUC'.
    """
    y_pred = clf.predict(X_test)
    y_proba = clf.predict_proba(X_test)[:, 1] if hasattr(clf,
"predict_proba") else None

    bal_acc = balanced_accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred)

    metrics = {
        'Balanced Accuracy': bal_acc,
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1': f1_score(y_test, y_pred),
        'ROC AUC': roc_auc_score(y_test, y_proba) if y_proba is not
None else None
    }

    # Print results
    title = f"Evaluation: {name or clf.__class__.__name__}"
    print(f"--- {title} ---")
    print(f"Balanced Accuracy: {bal_acc:.4f}")
```

```
    print(report)

    # Plot
    if show_plots:
        fig, axes = plt.subplots(1, 2, figsize=(12, 5))
        plot_confusion(y_test, y_pred, ax=axes[0])
        if y_proba is not None:
            plot_roc(y_test, y_proba, ax=axes[1])
        plt.tight_layout()
        plt.show()

    # Store
    if results is not None and name is not None:
        results[name] = metrics

    return metrics


def evaluate_grid_search(
    grid_search,
    X_test,
    y_test,
    results: dict = None,
    name: str = None,

    show_plots: bool = True
):
    """
    Evaluate a GridSearchCV result on test data.

    Prints best parameters, cross-validation score, and evaluates on
test set.

    Parameters
    ----------
    grid_search : GridSearchCV
        Fitted GridSearchCV instance.
    X_test : array-like
        Test features.
    y_test : array-like
        True labels.
    results : dict, optional
        Dictionary to update with computed metrics.
    name : str, optional
        Name under which to store metrics in results.
    labels : list of str, optional
        Class names for display.
    show_plots : bool, default=True
        Whether to display plots.
```

```
    Returns
    -------
    metrics : dict
    """
    print(f"Best parameters: {grid_search.best_params_}")
    print(f"CV mean score: {grid_search.best_score_:.4f}")
    return evaluate_model(
        clf=grid_search.best_estimator_,
        X_test=X_test,
        y_test=y_test,
        name=name or grid_search.best_estimator_.__class__.__name__,
        results=results,

        show_plots=show_plots
    )
```

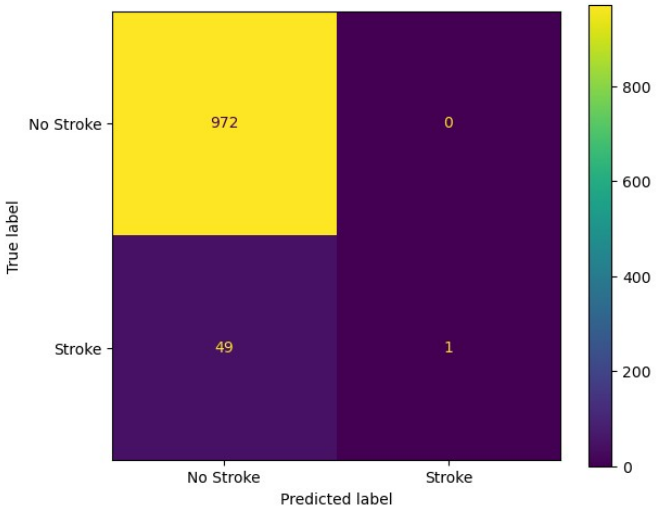
6.1.1. Model 1: Logistic Regression (Baseline)

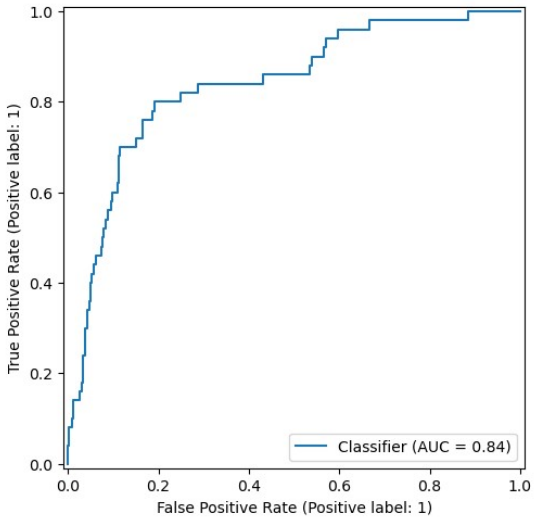
Logistic Regression is a linear model that is an excellent starting point for any classification task. It estimates probabilities and is very efficient, making it the perfect baseline to compare against more complex models.

```
from sklearn.metrics import ConfusionMatrixDisplay

# Initialize the Logistic Regression model
# We set max_iter=1000 to ensure the model has enough iterations to
converge, preventing a common warning.
log_reg = LogisticRegression(random_state=42, max_iter=1000)
log_reg.fit(X_train, y_train)

# Make predictions on the original, unbalanced test set
y_pred_log_reg = log_reg.predict(X_test)

# Predict probabilities, which are needed for the ROC-AUC score
y_pred_proba_log_reg = log_reg.predict_proba(X_test)[:, 1] # We only
need the probabilities for the 'positive' class (stroke=1)

# Print the Classification Report
# This report gives a great summary of precision, recall, and f1-
score.
metrics = evaluate_model(
    clf=log_reg,
    X_test=X_test,
    y_test=y_test,
```

```
    name="LogisticRegression",
    results=results,
    show_plots=True                    # set False if running in non-
interactive mode
)
print(results)
```

```
--- Evaluation: LogisticRegression ---
Balanced Accuracy: 0.5100
              precision    recall  f1-score   support

           0       0.95      1.00      0.98       972
           1       1.00      0.02      0.04        50

    accuracy                           0.95      1022
   macro avg       0.98      0.51      0.51      1022
weighted avg       0.95      0.95      0.93      1022
```






```
{'LogisticRegression': {'Balanced Accuracy': 0.51, 'Precision': 1.0,
'Recall': 0.02, 'F1': 0.0392156862745098, 'ROC AUC':
0.8390946502057613}}
```

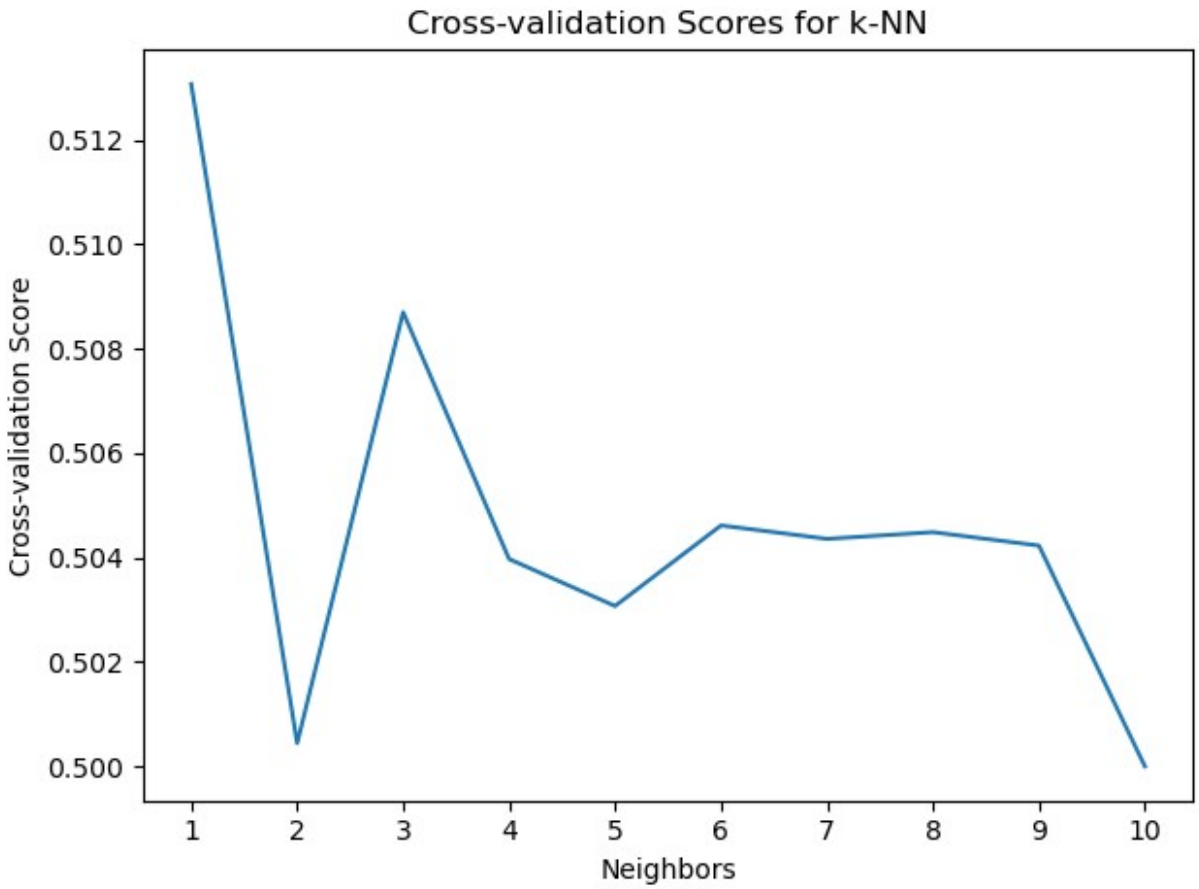
6.1.2. Model 2: K-Nearest Neighbors (KNN)

Now, follow the same 4 steps as above for the KNN model.

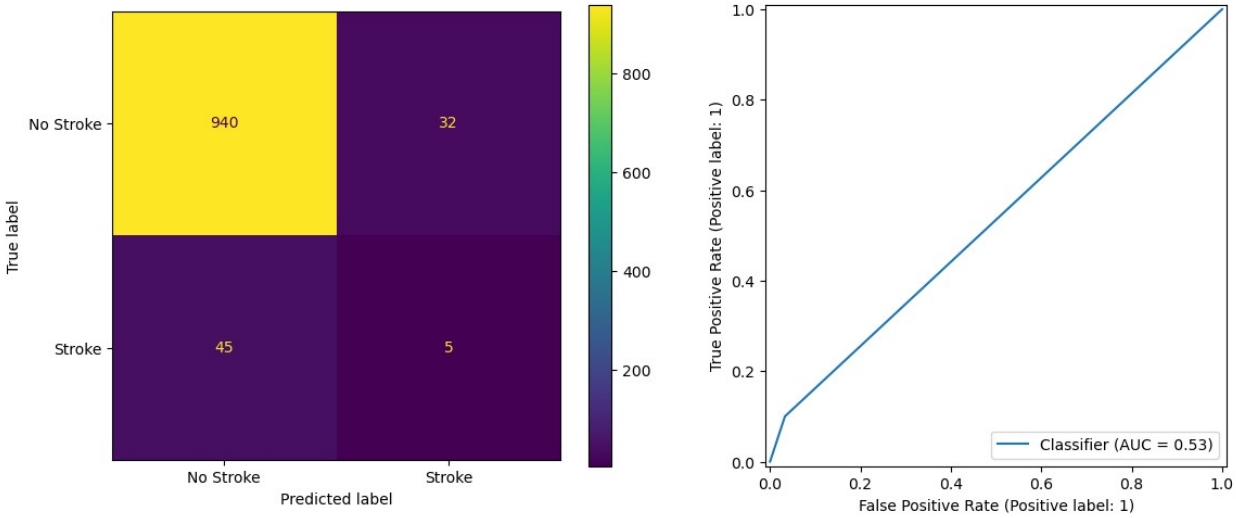
```
from sklearn.model_selection import cross_val_score

k_values = list(range(1, 11))

# Use cross-validation to evaluate the performance of each k value
```

```
cv_scores = []
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train, cv=5,
scoring='balanced_accuracy')
    cv_scores.append(scores.mean())

# Find the k value that gives the best performance
best_k = k_values[np.argmax(cv_scores)]
print("Best k value:", best_k)

plt.plot(k_values, cv_scores);
plt.title('Cross-validation Scores for k-NN');
plt.xticks(k_values);
plt.xlabel('Neighbors');
plt.ylabel('Cross-validation Score');
plt.tight_layout();
plt.show();
print(f"""

we can see ({best_k}) has the best score

""")




params = {
    "n_neighbors": list(range(1, 11)),
    "weights": ["uniform", "distance"],
    "metric": ["euclidean", "chebyshev", "minkowski", "manhattan"]
}

knn_clf = KNeighborsClassifier()

knn = GridSearchCV(knn_clf, params, scoring="balanced_accuracy")
knn.fit(X_train, y_train)



# Evaluate best-found estimator on the test set
evaluate_grid_search(
    grid_search=knn,
    X_test=X_test,
    y_test=y_test,
    results=results,
    name="K-Nearest Neighbors",
    show_plots=True
)
```

```
Best k value: 1
```




```
we can see (1) has the best score


Best parameters: {'metric': 'manhattan', 'n_neighbors': 1, 'weights':
'uniform'}
CV mean score: 0.5247
--- Evaluation: K-Nearest Neighbors ---
Balanced Accuracy: 0.5335
              precision    recall  f1-score   support

           0       0.95      0.97      0.96       972
           1       0.14      0.10      0.11        50

    accuracy                           0.92      1022
   macro avg       0.54      0.53      0.54      1022
weighted avg       0.91      0.92      0.92      1022
```



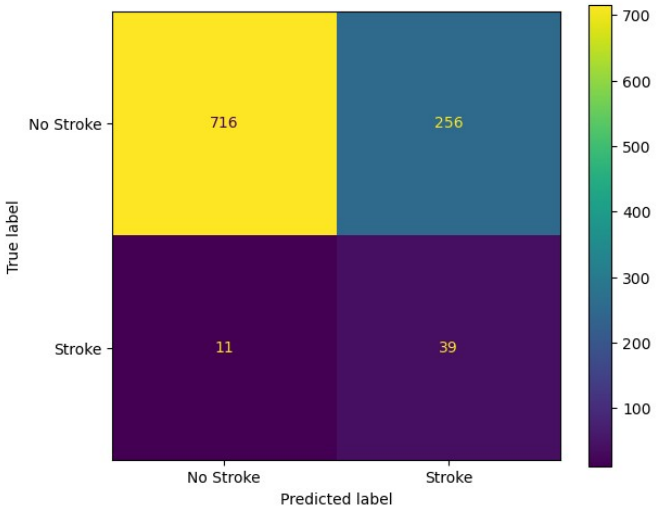

```
{'Balanced Accuracy': 0.5335390946502058,
 'Precision': 0.13513513513513514,
 'Recall': 0.1,
 'F1': 0.11494252873563218,
 'ROC AUC': 0.5335390946502059}
```

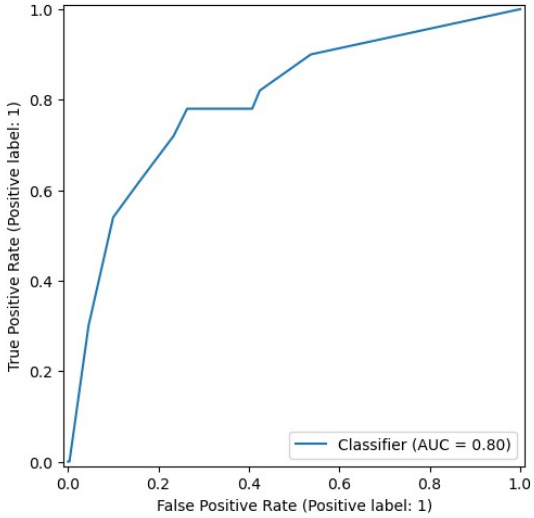
6.1.3. Model 3: Decision Tree

The Decision Tree is a non-linear model that works by splitting the data into branches based on feature values, creating a tree-like flow of decisions. It's highly interpretable and often forms the basis for more powerful ensemble models like Random Forest.

```
# Initialize the Decision Tree Classifier
# We set a random_state to ensure the results are the same every time we run the code.
# class_weight=balanced adjusts weights inversely proportional to class frequencies
tree_clf = DecisionTreeClassifier(
    max_depth=5, random_state=42, class_weight="balanced")
tree_clf.fit(X_train, y_train)

evaluate_model(
    clf=tree_clf,
    X_test=X_test,
    y_test=y_test,
    name="DecisionTree",
    results=results,
    show_plots=True
)


--- Evaluation: DecisionTree ---
Balanced Accuracy: 0.7583
```

	precision	recall	f1-score	support
0	0.98	0.74	0.84	972
1	0.13	0.78	0.23	50
accuracy			0.74	1022
macro avg	0.56	0.76	0.53	1022
weighted avg	0.94	0.74	0.81	1022



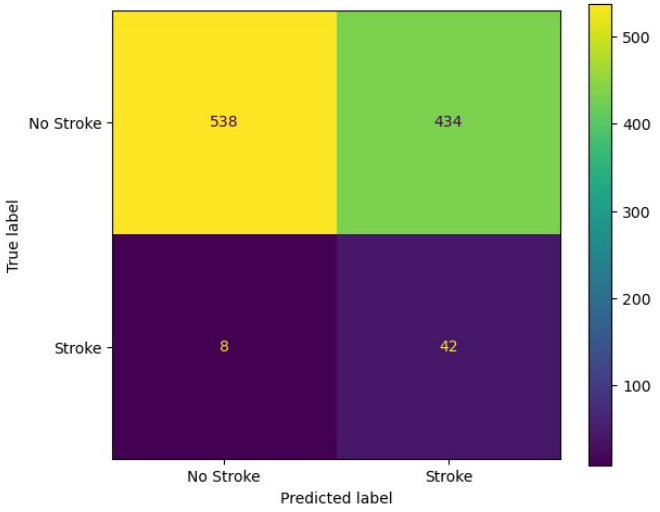


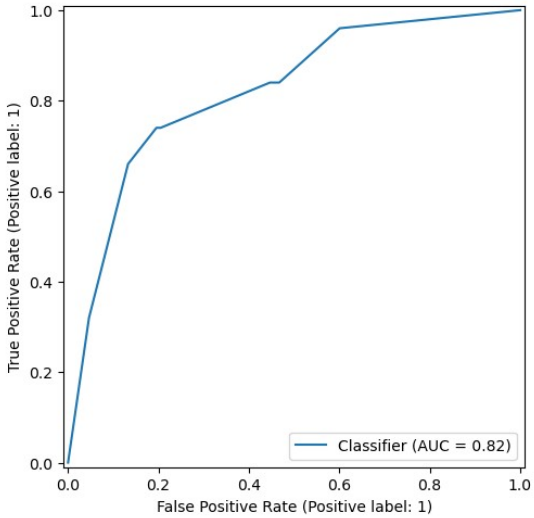

```
{'Balanced Accuracy': 0.7583127572016461,
 'Precision': 0.13220338983050847,
 'Recall': 0.78,
 'F1': 0.22608695652173913,
 'ROC AUC': 0.7985905349794238}
```

```
from sklearn.model_selection import GridSearchCV

params = {
    "criterion": ["gini", "entropy"],
    "max_depth": list(range(1, 10)),
    "max_features": [None, "log2", "sqrt"],
}


# class_weight=balanced adjusts weights inversely proportional to
class frequencies
tree_clf = DecisionTreeClassifier(random_state=42,
class_weight="balanced")
grid_tree_clf = GridSearchCV(tree_clf, params,
scoring="balanced_accuracy")
grid_tree_clf.fit(X_train, y_train)
evaluate_grid_search(
```

```
    grid_search=grid_tree_clf,
    X_test=X_test,
    y_test=y_test,
    results=results,
    name="DecisionTree-GridSearch",
    show_plots=True
)
```

```
Best parameters: {'criterion': 'entropy', 'max_depth': 3,
'max_features': None}
CV mean score: 0.7549
--- Evaluation: DecisionTree-GridSearch ---
Balanced Accuracy: 0.6967
              precision    recall  f1-score   support

           0       0.99      0.55      0.71       972
           1       0.09      0.84      0.16        50

    accuracy                           0.57      1022
   macro avg       0.54      0.70      0.43      1022
weighted avg       0.94      0.57      0.68      1022
```






```
{'Balanced Accuracy': 0.6967489711934156,
 'Precision': 0.08823529411764706,
 'Recall': 0.84,
 'F1': 0.1596958174904943,
 'ROC AUC': 0.8203806584362141}
```

6.1.4. Model 4: SVM

```
#@title SVM takes a lot of time skipped by default
execute_svm = False #@param {type:"boolean"}
```

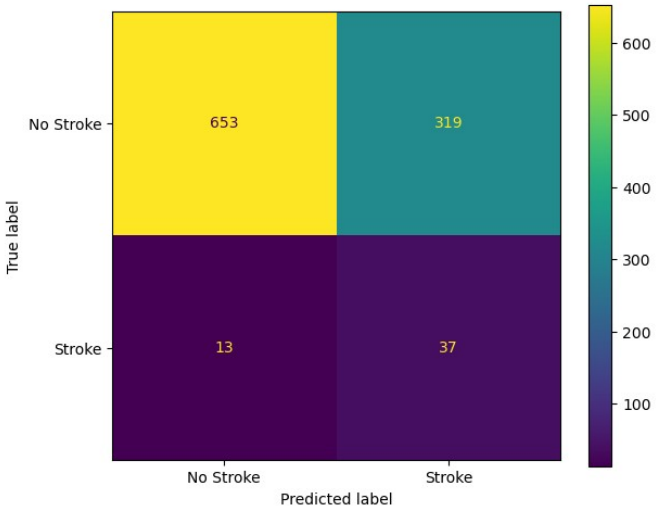
```
from sklearn.svm import SVC

if execute_svm:
  params = {
      "C": [1, 10, 50],
      "gamma": ["scale", 1, 0.1, 0.01],
      "kernel": ["linear", "rbf", "poly"],
      "degree": [2, 3, 4]
  }

  svm_clf = SVC(random_state=42, class_weight="balanced",
probability=True)

  grid_svm_clf = GridSearchCV(svm_clf, params,
scoring="balanced_accuracy")
  grid_svm_clf.fit(X_train, y_train)

  evaluate_grid_search(
          grid_search=grid_svm_clf,
          X_test=X_test,
          y_test=y_test,
          results=results,
          name="SVM",
          show_plots=True
      )
```

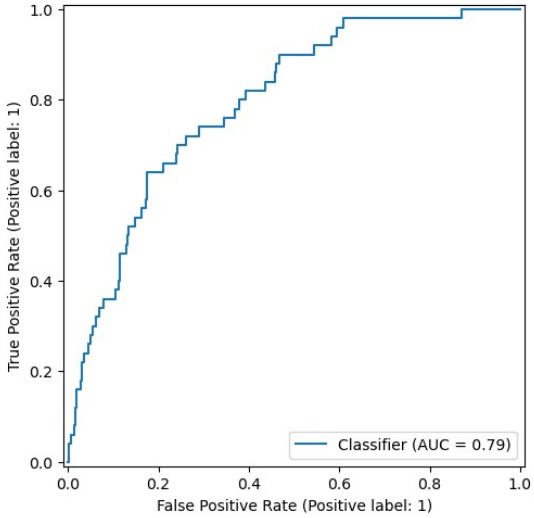
The resulting accuracy of the best model is high, but it's worth noticing that a considerable amount of time is required for training and cross-validation.

6.1.6. Model 6:Naive Bayes

```
params = {
    "var_smoothing": np.logspace(0, -9, num=300)
}

gnb_clf = GaussianNB()

grid_gnb_clf = GridSearchCV(gnb_clf, params,
scoring="balanced_accuracy")
grid_gnb_clf.fit(X_train, y_train)
evaluate_grid_search(
    grid_search=grid_gnb_clf,
    X_test=X_test,
    y_test=y_test,
    results=results,
    name="GaussianNB-GridSearch",
    show_plots=True
)
```

```
Best parameters: {'var_smoothing': 0.014584098829439856}
CV mean score: 0.7354
--- Evaluation: GaussianNB-GridSearch ---
Balanced Accuracy: 0.7059
              precision    recall  f1-score   support

           0       0.98      0.67      0.80       972
           1       0.10      0.74      0.18        50

    accuracy                           0.68      1022
   macro avg       0.54      0.71      0.49      1022
weighted avg       0.94      0.68      0.77      1022
```



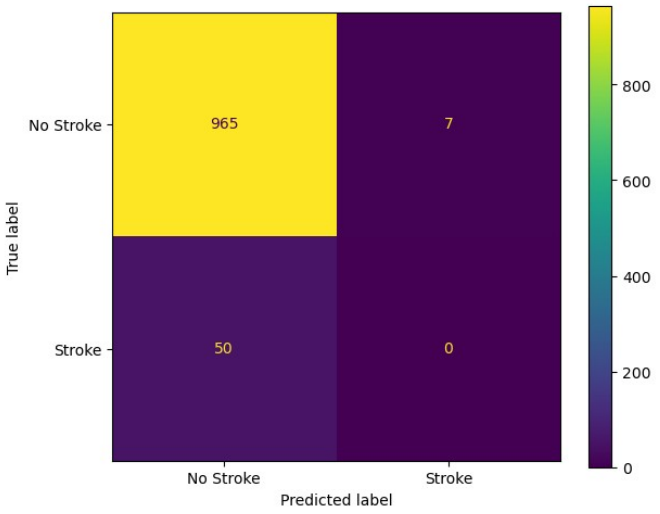


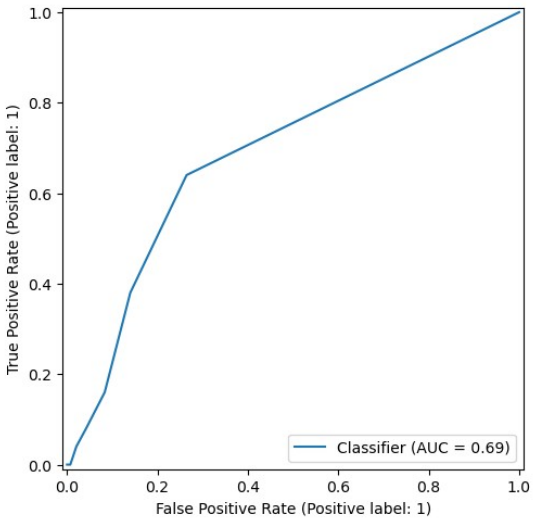

```
{'Balanced Accuracy': 0.7059053497942387,
 'Precision': 0.10393258426966293,
 'Recall': 0.74,
 'F1': 0.18226600985221675,
 'ROC AUC': 0.7945061728395062}
```

6.1.7. Model 7: Bagging (Ensemble)

```
from sklearn.ensemble import BaggingClassifier

bag_clf = BaggingClassifier(random_state=42)
bag_clf.fit(X_train, y_train)
evaluate_model(
    clf=bag_clf,
    X_test=X_test,
    y_test=y_test,
    name="BaggingClassifier",
    results=results,
```

```
    show_plots=True
)

--- Evaluation: BaggingClassifier ---
Balanced Accuracy: 0.4964
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       972
           1       0.00      0.00      0.00        50

    accuracy                           0.94      1022
   macro avg       0.48      0.50      0.49      1022
weighted avg       0.90      0.94      0.92      1022
```

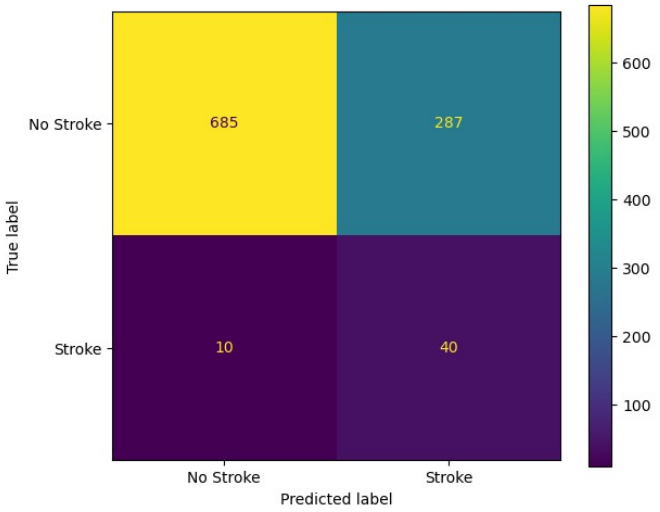


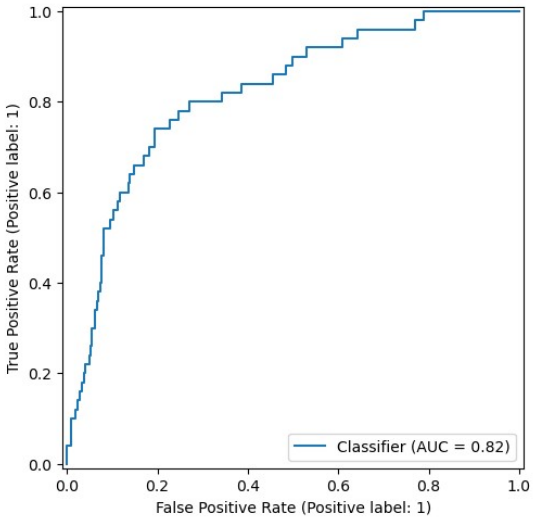



```
{'Balanced Accuracy': 0.4963991769547325,
 'Precision': 0.0,
 'Recall': 0.0,
 'F1': 0.0,
 'ROC AUC': 0.6884567901234568}
```

```
from imblearn.ensemble import BalancedBaggingClassifier

params = {
    "n_estimators": [100, 200, 500],
    "max_samples": [0.2, 0.5, 1.0],
    "max_features": [0.5, 1.0],
}

grid_bb_clf = GridSearchCV(
    BalancedBaggingClassifier(random_state=42),
    params,
    scoring="balanced_accuracy"
```

```
)

grid_bb_clf.fit(X_train, y_train)
evaluate_grid_search(
    grid_search=grid_bb_clf,
    X_test=X_test,
    y_test=y_test,
    results=results,
    name="BalancedBaggingClassifier-GridSearch",
    show_plots=True
)

Best parameters: {'max_features': 0.5, 'max_samples': 0.2,
'n_estimators': 500}
CV mean score: 0.7785
--- Evaluation: BalancedBaggingClassifier-GridSearch ---
Balanced Accuracy: 0.7524
              precision    recall  f1-score   support

           0       0.99      0.70      0.82       972
           1       0.12      0.80      0.21        50

    accuracy                           0.71      1022
   macro avg       0.55      0.75      0.52      1022
weighted avg       0.94      0.71      0.79      1022
```





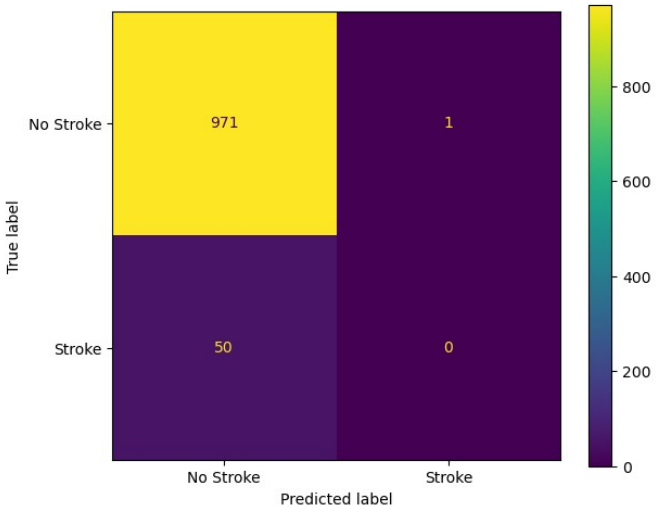

```
{'Balanced Accuracy': 0.7523662551440329,
 'Precision': 0.12232415902140673,
 'Recall': 0.8,
 'F1': 0.21220159151193635,
 'ROC AUC': 0.8188477366255145}
```

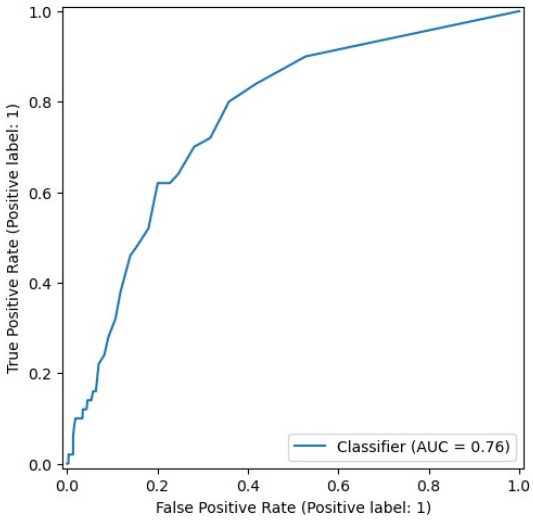
6.1.8. Model 8: Random Forest (Ensemble)

This is our first ensemble model.

```
from sklearn.ensemble import RandomForestClassifier

rf_clf = RandomForestClassifier(random_state=42,
class_weight="balanced")
rf_clf.fit(X_train, y_train)
evaluate_model(
    clf=rf_clf,
    X_test=X_test,
    y_test=y_test,
    name="RandomForest",
    results=results,
    show_plots=True
)
```

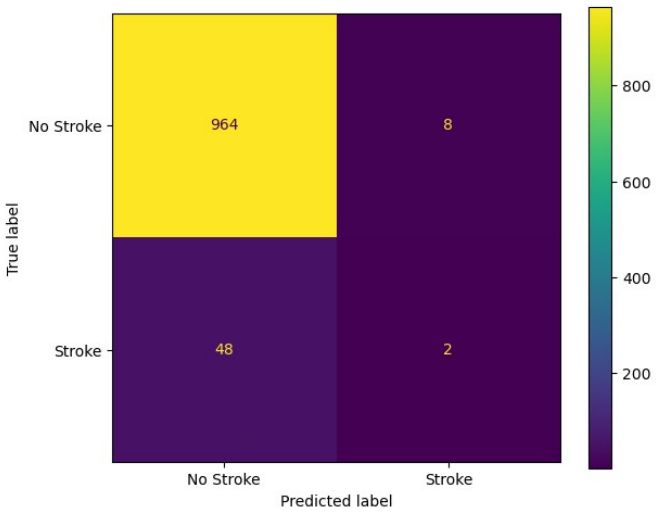
```
--- Evaluation: RandomForest ---
Balanced Accuracy: 0.4995
              precision    recall  f1-score   support

           0       0.95      1.00      0.97       972
           1       0.00      0.00      0.00        50

    accuracy                           0.95      1022
   macro avg       0.48      0.50      0.49      1022
weighted avg       0.90      0.95      0.93      1022
```

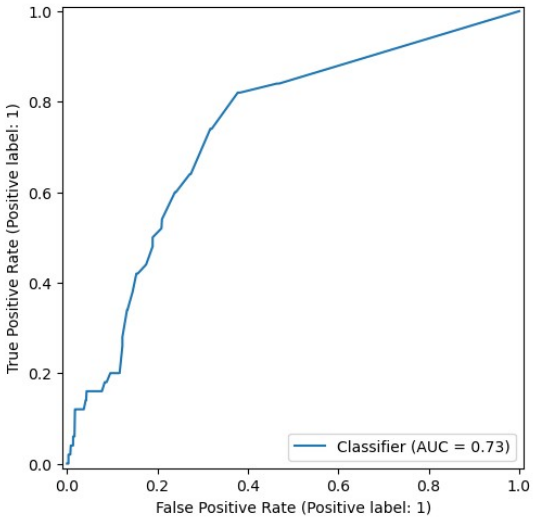





```
{'Balanced Accuracy': 0.49948559670781895,
 'Precision': 0.0,
 'Recall': 0.0,
```

```
 'F1': 0.0,
 'ROC AUC': 0.7635288065843621}

from sklearn.ensemble import ExtraTreesClassifier

et_clf = ExtraTreesClassifier(random_state=42,
class_weight="balanced")
et_clf.fit(X_train, y_train)
evaluate_model(
    clf=et_clf,
    X_test=X_test,
    y_test=y_test,
    name="ExtraTrees",
    results=results,
    show_plots=True
)

--- Evaluation: ExtraTrees ---
Balanced Accuracy: 0.5159
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       972
           1       0.20      0.04      0.07        50

    accuracy                           0.95      1022
   macro avg       0.58      0.52      0.52      1022
weighted avg       0.92      0.95      0.93      1022
```



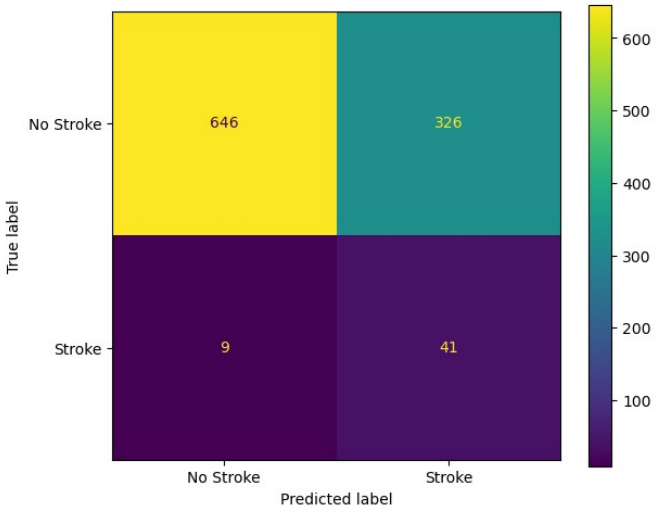


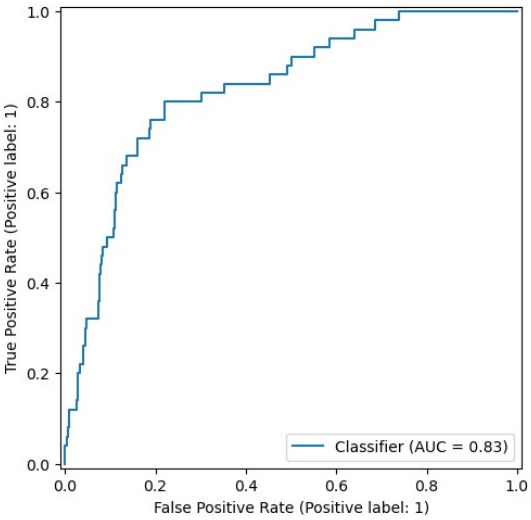

```
{'Balanced Accuracy': 0.5158847736625515,
 'Precision': 0.2,
 'Recall': 0.04,
```

```
 'F1': 0.06666666666666667,
 'ROC AUC': 0.7349382716049382}
```

```
from imblearn.ensemble import BalancedRandomForestClassifier

params = {
    "n_estimators": [200, 300],
    "criterion": ["gini", "entropy"],
    "max_depth": list(range(5, 10)),
    "max_features": ["log2", "sqrt"],
}

grid_brf_clf = GridSearchCV(
    BalancedRandomForestClassifier(random_state=42, oob_score=True),
    params,
    scoring="balanced_accuracy"
)

grid_brf_clf.fit(X_train, y_train)
evaluate_grid_search(
    grid_search=grid_brf_clf,
    X_test=X_test,
    y_test=y_test,
    results=results,
    name="BalancedRandomForest-GridSearch",
    show_plots=True
)
```

```
Best parameters: {'criterion': 'entropy', 'max_depth': 9,
'max_features': 'log2', 'n_estimators': 200}
CV mean score: 0.7787
--- Evaluation: BalancedRandomForest-GridSearch ---
Balanced Accuracy: 0.7423
              precision    recall  f1-score   support

           0       0.99      0.66      0.79       972
           1       0.11      0.82      0.20        50

    accuracy                           0.67      1022
   macro avg       0.55      0.74      0.50      1022
weighted avg       0.94      0.67      0.76      1022
```





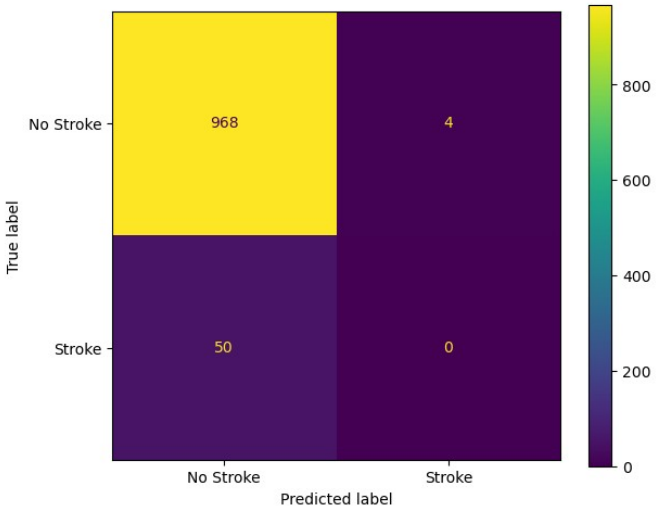

```
{'Balanced Accuracy': 0.7423045267489712,
 'Precision': 0.11171662125340599,
 'Recall': 0.82,
 'F1': 0.19664268585131894,
 'ROC AUC': 0.8281893004115226}
```

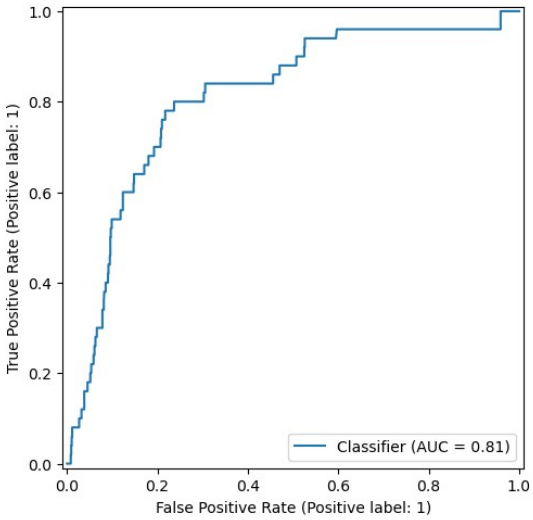
6.1.9. Model 9: Gradient Boosting (Ensemble)

This is another powerful ensemble model.

```
# 1. Initialize and train the model
gb = GradientBoostingClassifier(random_state=42)
gb.fit(X_train, y_train)
evaluate_model(
    clf=gb,
    X_test=X_test,
    y_test=y_test,
    name="GradientBoosting",
    results=results,
    show_plots=True
)
```

```
--- Evaluation: GradientBoosting ---
Balanced Accuracy: 0.4979
              precision    recall  f1-score   support

           0       0.95      1.00      0.97       972
           1       0.00      0.00      0.00        50

    accuracy                           0.95      1022
   macro avg       0.48      0.50      0.49      1022
weighted avg       0.90      0.95      0.93      1022
```



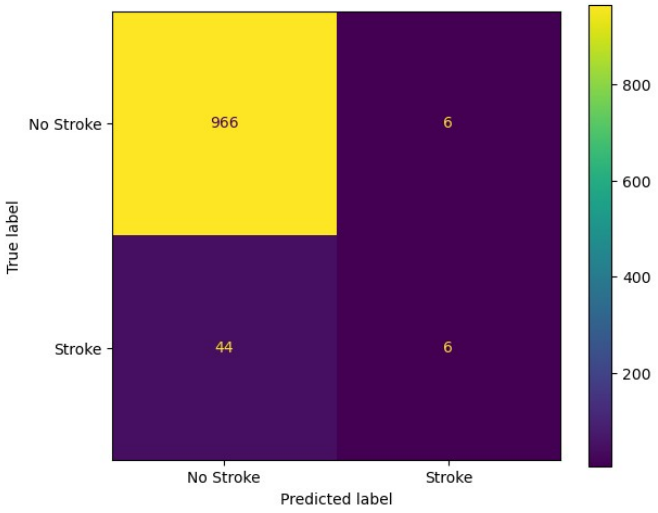


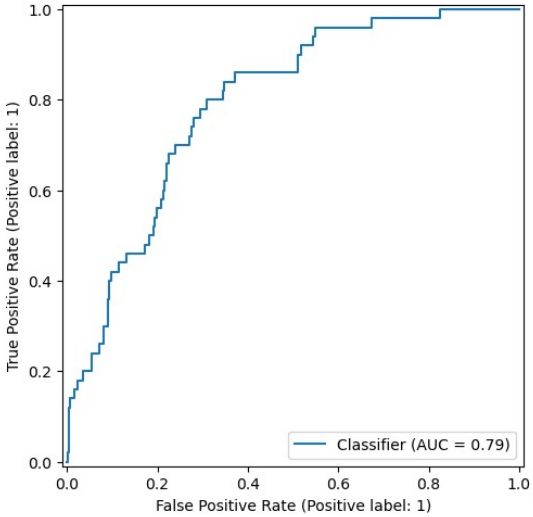

```
{'Balanced Accuracy': 0.49794238683127573,
 'Precision': 0.0,
 'Recall': 0.0,
 'F1': 0.0,
 'ROC AUC': 0.8094547325102881}
```

```
from sklearn.ensemble import GradientBoostingClassifier

params = {
    "n_estimators": [100, 200, 300],
    "learning_rate": [0.01, 0.05, 0.1],
    "max_depth": [3, 4, 5],
    "subsample": [0.8, 1.0],
    "max_features": ["sqrt", "log2", None]
}

gb_clf = GradientBoostingClassifier(random_state=42)
grid_gb_clf = GridSearchCV(gb_clf, params,
scoring="balanced_accuracy")
grid_gb_clf.fit(X_train, y_train)

evaluate_grid_search(
    grid_search=grid_gb_clf,
    X_test=X_test,
    y_test=y_test,
    results=results,
    name="GradientBoosting-GridSearch",
    show_plots=True
)
```

```
Best parameters: {'learning_rate': 0.1, 'max_depth': 4,
'max_features': 'sqrt', 'n_estimators': 300, 'subsample': 1.0}
CV mean score: 0.5386
--- Evaluation: GradientBoosting-GridSearch ---
```

```
Balanced Accuracy: 0.5569
              precision    recall  f1-score   support

           0       0.96      0.99      0.97       972
           1       0.50      0.12      0.19        50

    accuracy                           0.95      1022
   macro avg       0.73      0.56      0.58      1022
weighted avg       0.93      0.95      0.94      1022
```






```
{'Balanced Accuracy': 0.5569135802469136,
 'Precision': 0.5,
 'Recall': 0.12,
 'F1': 0.1935483870967742,
 'ROC AUC': 0.7928806584362139}
```

6.1.10. Model 10: XGBoost (Ensemble)

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting that solve many data science problems in a fast and accurate way.

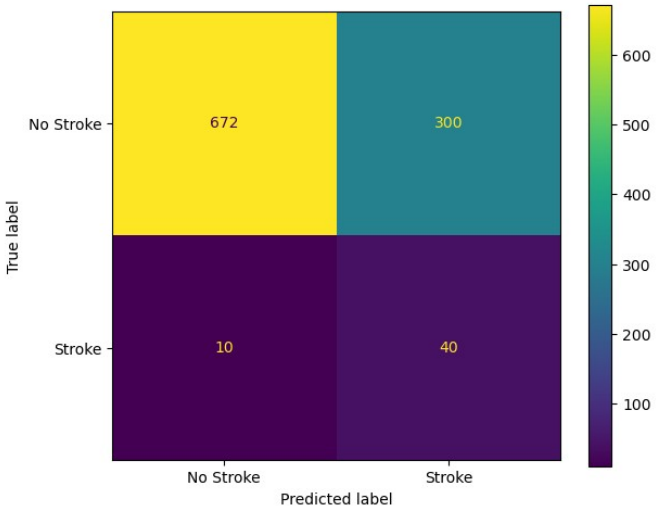
To work with imbalanced data sets, it's recommended to set the `scale_pos_weight` to the ratio of negative instances to the positive ones.

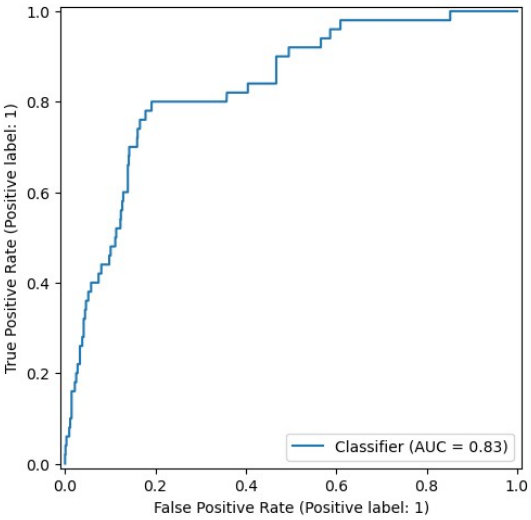
```
from collections import Counter

counter = Counter(y)
estimate = counter[0] / counter[1]
print("Estimate: %.3f" % estimate)
```

```
Estimate: 19.518
```

```
from xgboost import XGBClassifier

xgb_clf = XGBClassifier(random_state=42, scale_pos_weight=estimate,
                        objective="binary:logistic", verbosity=0)

params = {
    "n_estimators": [50, 100],
    "max_depth": [3, 5, 10],
    "learning_rate": [0.1, 0.01],
    "subsample": [0.8, 1],
    "colsample_bytree": [0.5, 0.8]
}

grid_xgb_clf = GridSearchCV(xgb_clf, params,
scoring="balanced_accuracy")
grid_xgb_clf.fit(X_train, y_train)
evaluate_grid_search(
    grid_search=grid_xgb_clf,
    X_test=X_test,
    y_test=y_test,
    results=results,
    name="XGBoost-GridSearch",
    show_plots=True
)
```

```
Best parameters: {'colsample_bytree': 0.5, 'learning_rate': 0.01,
'max_depth': 3, 'n_estimators': 100, 'subsample': 0.8}
CV mean score: 0.7739
--- Evaluation: XGBoost-GridSearch ---
Balanced Accuracy: 0.7457
              precision    recall  f1-score   support

           0       0.99      0.69      0.81       972
           1       0.12      0.80      0.21        50

    accuracy                           0.70      1022
   macro avg       0.55      0.75      0.51      1022
weighted avg       0.94      0.70      0.78      1022
```



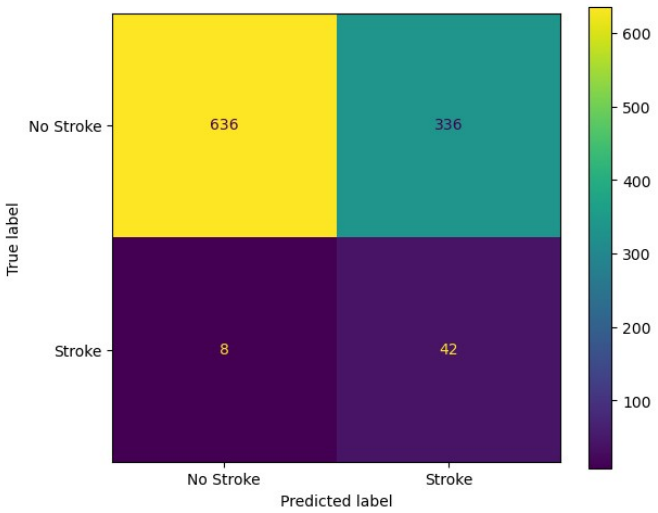


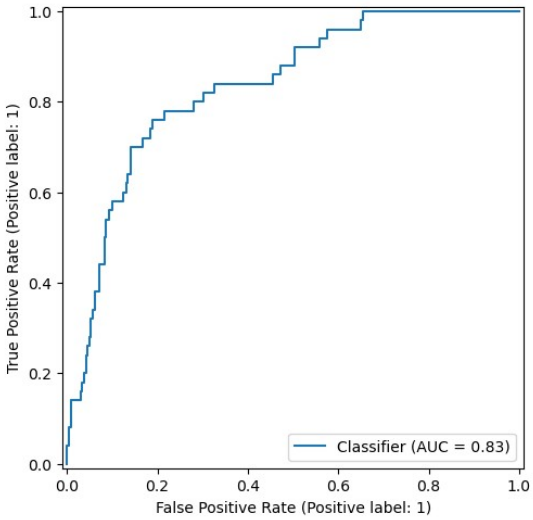

```
{'Balanced Accuracy': 0.745679012345679,
 'Precision': 0.11764705882352941,
 'Recall': 0.8,
 'F1': 0.20512820512820512,
 'ROC AUC': 0.8313683127572016}
```

6.1.11. Model 11: EasyEnsemble (Ensemble)

```
from imblearn.ensemble import EasyEnsembleClassifier

params = {
    "n_estimators": [50, 100, 200],
}

grid_ee_clf = GridSearchCV(
    EasyEnsembleClassifier(random_state=42),
    params,
    scoring="balanced_accuracy"
)

grid_ee_clf.fit(X_train, y_train)
evaluate_grid_search(
    grid_search=grid_ee_clf,
    X_test=X_test,
    y_test=y_test,
    results=results,
    name="EasyEnsemble-GridSearch",
    show_plots=True
)

Best parameters: {'n_estimators': 50}
CV mean score: 0.7704
--- Evaluation: EasyEnsemble-GridSearch ---
Balanced Accuracy: 0.7472
```

```
              precision    recall  f1-score   support

           0       0.99      0.65      0.79       972
           1       0.11      0.84      0.20        50

    accuracy                           0.66      1022
   macro avg       0.55      0.75      0.49      1022
weighted avg       0.94      0.66      0.76      1022
```





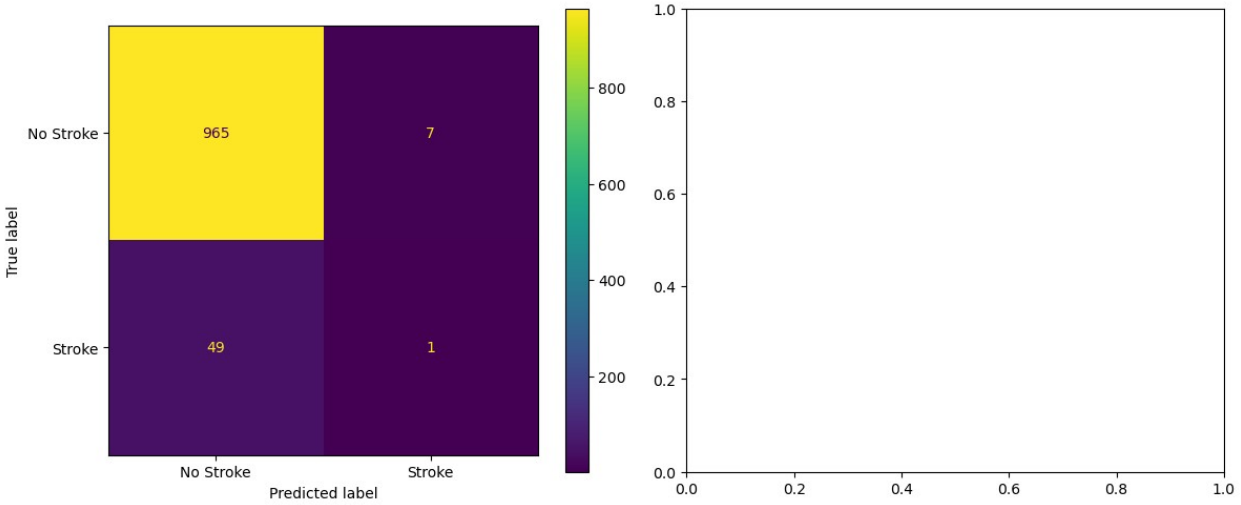

```
{'Balanced Accuracy': 0.7471604938271605,
 'Precision': 0.1111111111111111,
 'Recall': 0.84,
 'F1': 0.19626168224299065,
 'ROC AUC': 0.8344855967078189}
```

6.1.12. Model 12: Voting (Ensemble)

```
from sklearn.ensemble import VotingClassifier
def get_models():
    """
    Returns a list of tuples containing model names and their
instances.
    """
    return [

        ("Extra Trees", et_clf),
        ("Gradient Boosting", gb),
        ("XGBoost", xgb_clf),
        ("Bagging", bag_clf),
        ("Balanced Random Forest", grid_brf_clf.best_estimator_),
        ("EasyEnsemble", grid_ee_clf.best_estimator_),
```

```
    ]
hard_voting_clf =
VotingClassifier(estimators=get_models(),voting="hard")
hard_voting_clf.fit(X_train, y_train)
evaluate_model(
    clf=hard_voting_clf,
    X_test=X_test,
    y_test=y_test,
    name="Hard Voting Classifier",
    results=results,
    show_plots=True
)
```

```
--- Evaluation: Hard Voting Classifier ---
Balanced Accuracy: 0.5064
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       972
           1       0.12      0.02      0.03        50

    accuracy                           0.95      1022
   macro avg       0.54      0.51      0.50      1022
weighted avg       0.91      0.95      0.93      1022
```




```
{'Balanced Accuracy': 0.5063991769547325,
 'Precision': 0.125,
 'Recall': 0.02,
 'F1': 0.034482758620689655,
 'ROC AUC': None}
```

6.1.12. RESULTS

```
import pandas as pd

# Convert the results dictionary to a DataFrame for easier sorting and
display
results_df = pd.DataFrame(results).T

# Display all metrics for all models in one table, sorted by Balanced
Accuracy (or any metric you prefer)
display(results_df.sort_values(by="Balanced Accuracy",
ascending=False))
```

```
                                    Balanced Accuracy  Precision
Recall  \
DecisionTree                                 0.758313   0.132203
0.78
BalancedBaggingClassifier-GridSearch         0.752366   0.122324
0.80
EasyEnsemble-GridSearch                      0.747160   0.111111
0.84
XGBoost-GridSearch                           0.745679   0.117647
0.80
BalancedRandomForest-GridSearch              0.742305   0.111717
0.82
GaussianNB-GridSearch                        0.705905   0.103933
0.74
DecisionTree-GridSearch                      0.696749   0.088235
0.84
GradientBoosting-GridSearch                  0.556914   0.500000
0.12
K-Nearest Neighbors                          0.533539   0.135135
0.10
ExtraTrees                                   0.515885   0.200000
0.04
LogisticRegression                           0.510000   1.000000
0.02
Hard Voting Classifier                       0.506399   0.125000
0.02
RandomForest                                 0.499486   0.000000
0.00
GradientBoosting                             0.497942   0.000000
0.00
BaggingClassifier                            0.496399   0.000000
0.00

                                         F1   ROC AUC
DecisionTree                       0.226087  0.798591
BalancedBaggingClassifier-GridSearch  0.212202  0.818848
EasyEnsemble-GridSearch            0.196262  0.834486
XGBoost-GridSearch                 0.205128  0.831368
```

```
BalancedRandomForest-GridSearch       0.196643  0.828189
GaussianNB-GridSearch                 0.182266  0.794506
DecisionTree-GridSearch               0.159696  0.820381
GradientBoosting-GridSearch           0.193548  0.792881
K-Nearest Neighbors                   0.114943  0.533539
ExtraTrees                            0.066667  0.734938
LogisticRegression                    0.039216  0.839095
Hard Voting Classifier                0.034483       NaN
RandomForest                          0.000000  0.763529
GradientBoosting                      0.000000  0.809455
BaggingClassifier                     0.000000  0.688457
```

6.2. Model with SMOTE

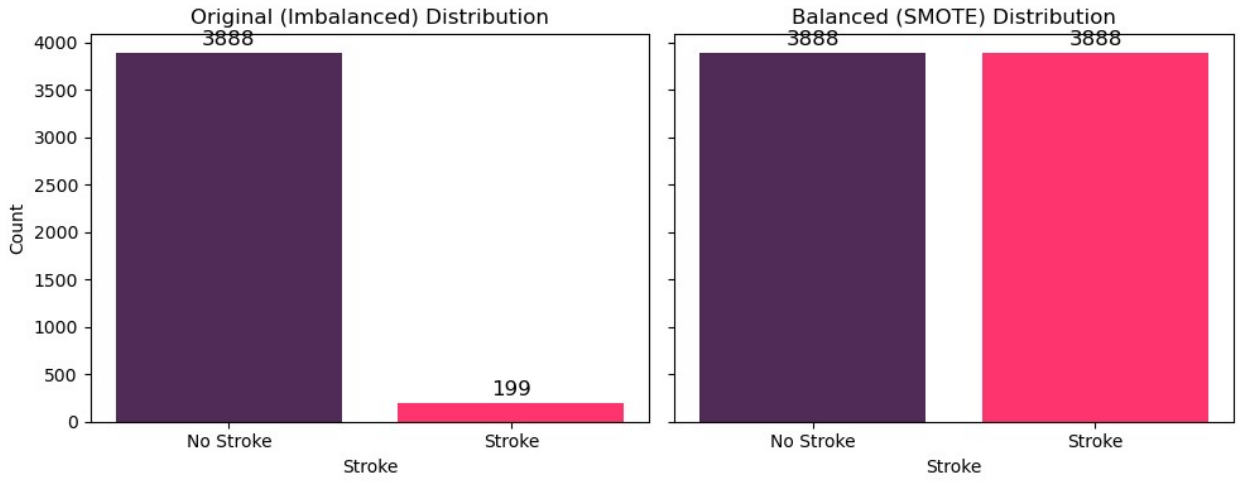
Now, we will address the class imbalance by applying the SMOTE technique. Importantly, SMOTE will be applied only to the training data after the train-test split to prevent data leakage. We will then train the same set of models (or a selection) on this oversampled training data and evaluate their performance on the original, untouched test set. This will allow us to assess the impact of SMOTE on model performance.

- **Synthetic Minority Over-sampling Technique (SMOTE)**: creates new, synthetic data points for the minority class (stroke cases).

6.2.1. Handling Class Imbalance with SMOTE

```
from imblearn.over_sampling import SMOTE


# Create copies of the training feature matrix and target variable to
avoid altering the original data
X_tmp = X_train.copy()
y_tmp = y_train.copy()

# Since all features are numeric after encoding, use SMOTE
smote = SMOTE(sampling_strategy='minority', random_state=42)
X_smote, y_smote = smote.fit_resample(X_tmp, y_tmp)



# Original (imbalanced) distribution
orig_counts = y_train.value_counts().sort_index()
# Balanced (SMOTE) distribution
smote_counts = y_smote.value_counts().sort_index()

fig, axes = plt.subplots(1, 2, figsize=(10, 4), sharey=True)

# Plot original
axes[0].bar(orig_counts.index.astype(str), orig_counts.values,
color=['#512b58', '#fe346e'])
axes[0].set_title('Original (Imbalanced) Distribution')
axes[0].set_xlabel('Stroke')
```

```
axes[0].set_ylabel('Count')
axes[0].set_xticks([0, 1])
axes[0].set_xticklabels(['No Stroke', 'Stroke'])

# Plot SMOTE
axes[1].bar(smote_counts.index.astype(str), smote_counts.values,
color=['#512b58', '#fe346e'])
axes[1].set_title('Balanced (SMOTE) Distribution')
axes[1].set_xlabel('Stroke')
axes[1].set_xticks([0, 1])
axes[1].set_xticklabels(['No Stroke', 'Stroke'])

for ax, counts in zip(axes, [orig_counts, smote_counts]):
    for i, v in enumerate(counts.values):
        ax.text(i, v + max(counts.values)*0.01, str(v), ha='center',
va='bottom', fontsize=12)

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



6.2.2. Model training

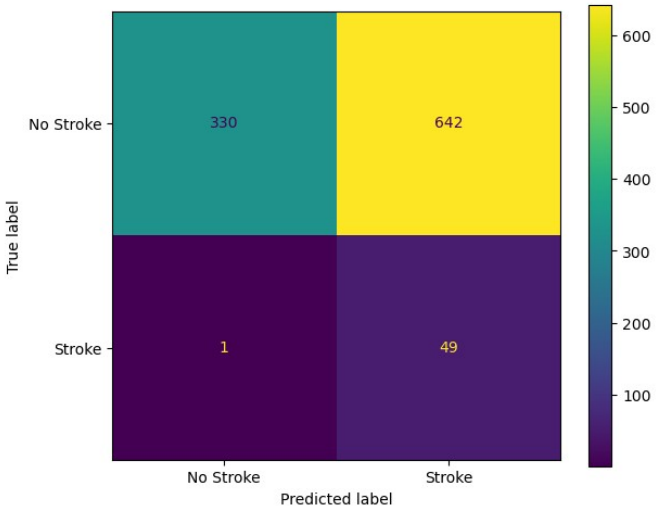
In this section, we will train the models used in the classification framework for predicting stroke occurrence. We will then compute key evaluation metrics, including Precision, Recall, F-Measure, AUC, and Accuracy, which are widely recognized in the relevant literature for assessing model performance.

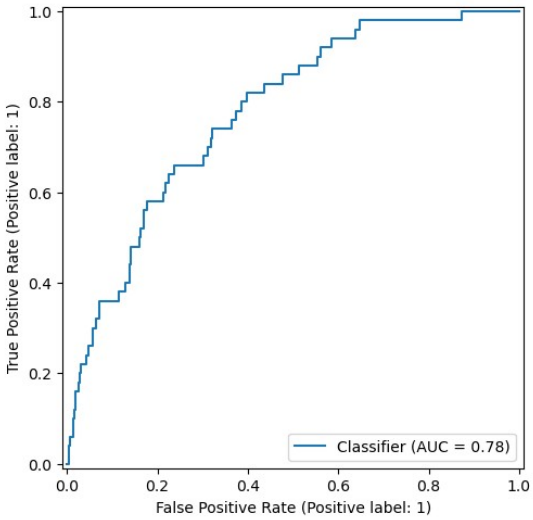
```
ResultsSmote={}

from imblearn.pipeline import Pipeline as ImbPipeline
from imblearn.over_sampling import SMOTE
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression, SGDClassifier
#from imblearn.ensemble import BalancedBaggingClassifier,
```

```
BalancedRandomForestClassifier, EasyEnsembleClassifier
#from xgboost import XGBClassifier
#from sklearn.svm import SVC


best_estimators = {}
grid_models = [
    ('Naive Bayes', GaussianNB(), [{'var_smoothing': np.logspace(0, -
9, num=10)}]),
    ('Logistic Regression', LogisticRegression(), [{'C':
[0.25,0.5,0.75,1], 'random_state':[0], 'solver': ['newton-cg',
'lbfgs', 'liblinear'], 'max_iter':[1000]}]),
    ('SGD', SGDClassifier(), [{'loss': ['hinge', 'log_loss',
'modified_huber'], 'penalty': ['l2', 'l1', 'elasticnet'], 'alpha':
[0.0001, 0.001, 0.01, 0.1]}]),
    ('KNN', KNeighborsClassifier(), [{'n_neighbors': list(range(1,
11)), 'metric': ['euclidean', 'manhattan', 'chebyshev', 'minkowski'],
'weights': ['uniform', 'distance']}]),
    ('Decision Tree', DecisionTreeClassifier(), [{'criterion':
['gini', 'entropy'], 'max_depth': list(range(1, 10)), 'max_features':
[None, 'log2', 'sqrt'], 'random_state': [0]}]),
    ('Random Forest', RandomForestClassifier(), [{'n_estimators':
[100, 150, 200], 'criterion': ['gini', 'entropy'], 'random_state':
[0]}]),
    ('Extra Trees', ExtraTreesClassifier(), [{'n_estimators': [100,
150, 200], 'criterion': ['gini', 'entropy'], 'random_state': [0]}]),
    ('Gradient Boosting', GradientBoostingClassifier(),
[{'n_estimators': [100, 200, 300], 'learning_rate': [0.01, 0.05, 0.1],
'max_depth': [3, 4, 5], 'subsample': [0.8, 1.0], 'max_features':
['sqrt', 'log2', None]}]),
    ('XGBoost', XGBClassifier(objective="binary:logistic",
verbosity=0), [{'n_estimators': [50, 100], 'max_depth': [3, 5, 10],
'learning_rate': [0.1, 0.01], 'subsample': [0.8, 1],
'colsample_bytree': [0.5, 0.8], 'scale_pos_weight': [estimate],
'random_state': [0]}]),
    ('SVM', SVC(probability=True), [{'C': [1, 10], 'gamma': ['scale',
0.1], 'kernel': ['linear', 'rbf'], 'class_weight': ['balanced'],
'random_state': [0]}]),
    ('Bagging', BaggingClassifier(), [{'n_estimators': [100, 200],
'random_state': [0]}]),
    ('BalancedBagging', BalancedBaggingClassifier(), [{'n_estimators':
[100, 200], 'random_state': [0]}]),
    ('BalancedRandomForest', BalancedRandomForestClassifier(),
[{'n_estimators': [200, 300], 'criterion': ['gini', 'entropy'],
'max_depth': list(range(5, 10)), 'max_features': ['log2', 'sqrt'],
'random_state': [0]}]),
    ('EasyEnsemble', EasyEnsembleClassifier(), [{'n_estimators': [50,
100], 'random_state': [0]}]),
    ('MLP', MLPClassifier(), [{'hidden_layer_sizes': [(10,), (20,),
```

```
(25,)], 'max_iter': [500], 'learning_rate': ['constant'],
'learning_rate_init': [0.003, 0.03, 0.3], 'momentum': [0.02, 0.2],
'activation': ['relu', 'tanh']}])
]

scoring = {
    'balanced_accuracy': make_scorer(balanced_accuracy_score),
    'accuracy': make_scorer(accuracy_score),
    'precision': make_scorer(precision_score, zero_division=0),
    'recall': make_scorer(recall_score),
    'f1': make_scorer(f1_score),
    'roc_auc': make_scorer(roc_auc_score)
}
for name, estimator, param_grid in grid_models:
    # embed SMOTE + scaling + grid inside one pipeline
    pipeline = ImbPipeline([
        ('smote', SMOTE(random_state=42)),
        ('scale', StandardScaler()),
        ('clf', GridSearchCV(
            estimator=estimator,
            param_grid=param_grid,
            scoring=scoring,
            refit='roc_auc',
            cv=5,
            n_jobs=-1
        ))
    ])

    # fit on train
    pipeline.fit(X_train, y_train)

    # pull out the fitted GridSearchCV
    grid = pipeline.named_steps['clf']
    best_estimators[name] = grid.best_estimator_
    print(f"\n=== {name} ===")
    evaluate_model(
        clf=grid.best_estimator_,   # the best estimator from the grid
search
        X_test=X_test,
        y_test=y_test,
        name=name,
        results=ResultsSmote,
        show_plots=True           # True if you want plots
    )



=== Naive Bayes ===
--- Evaluation: Naive Bayes ---
Balanced Accuracy: 0.6598
```

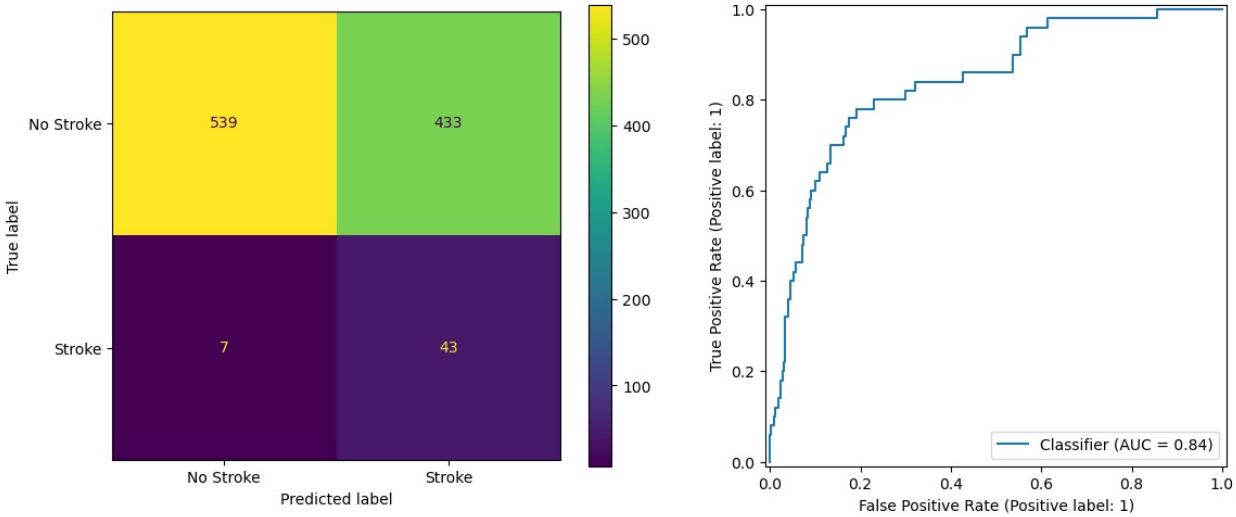
	precision	recall	f1-score	support
0	1.00	0.34	0.51	972
1	0.07	0.98	0.13	50
accuracy			0.37	1022
macro avg	0.53	0.66	0.32	1022
weighted avg	0.95	0.37	0.49	1022



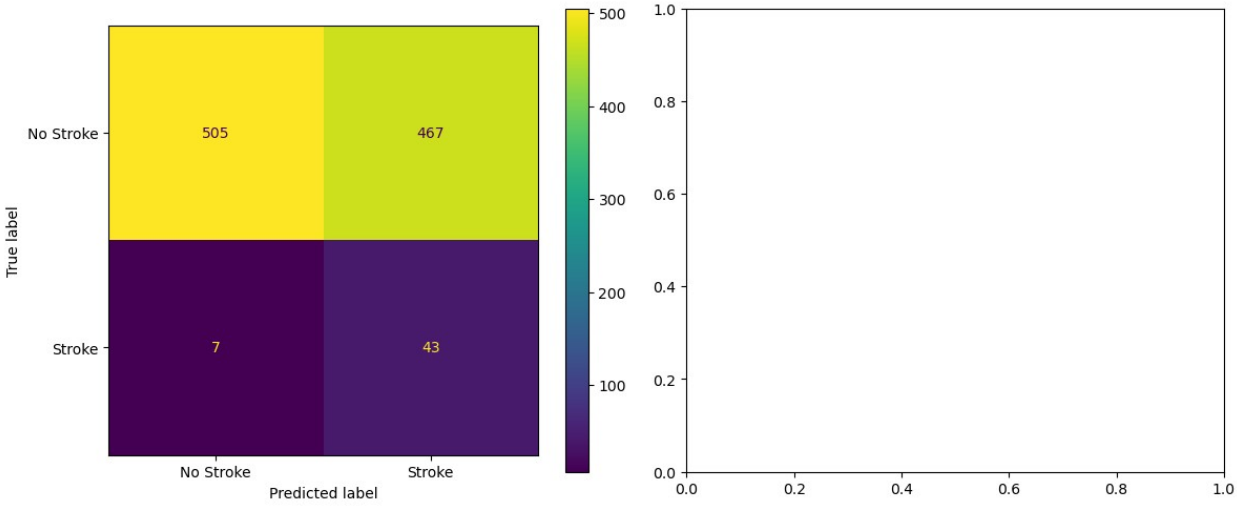



```
=== Logistic Regression ===
--- Evaluation: Logistic Regression ---
Balanced Accuracy: 0.7073
```

	precision	recall	f1-score	support
0	0.99	0.55	0.71	972
1	0.09	0.86	0.16	50
accuracy			0.57	1022
macro avg	0.54	0.71	0.44	1022
weighted avg	0.94	0.57	0.68	1022

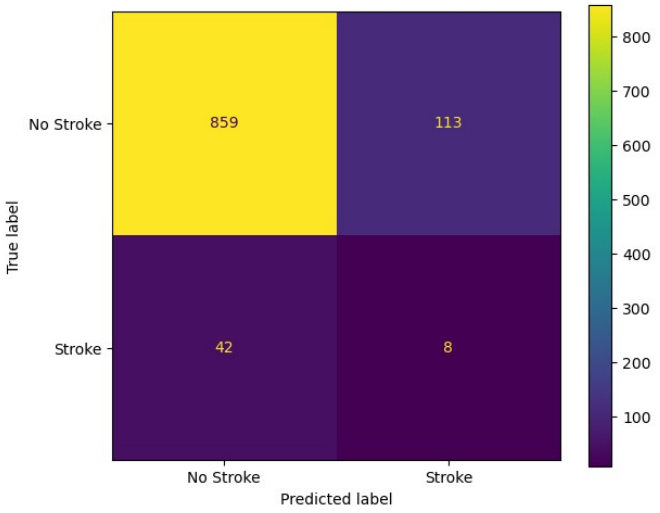


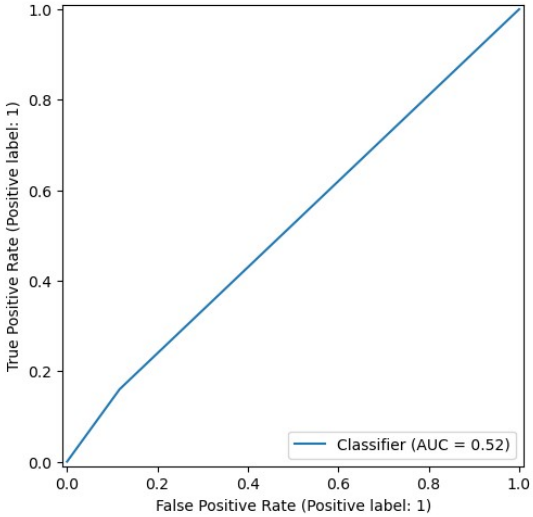

```
=== SGD ===
--- Evaluation: SGD ---
Balanced Accuracy: 0.6898
              precision    recall  f1-score   support

           0       0.99      0.52      0.68       972
           1       0.08      0.86      0.15        50

    accuracy                           0.54      1022
   macro avg       0.54      0.69      0.42      1022
weighted avg       0.94      0.54      0.65      1022
```



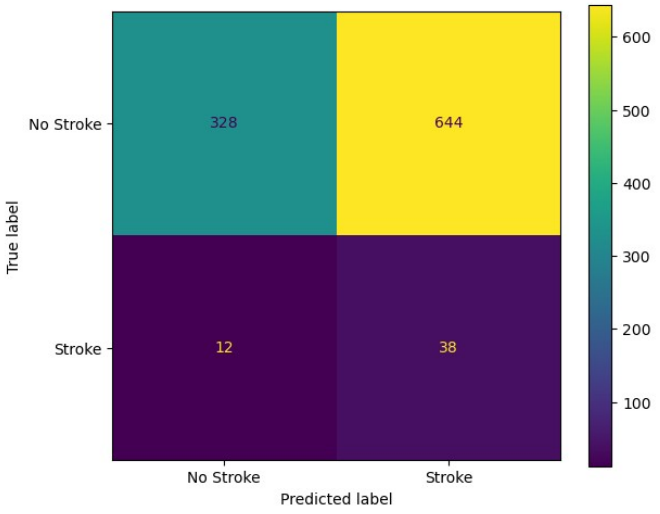

```
=== KNN ===
--- Evaluation: KNN ---
```

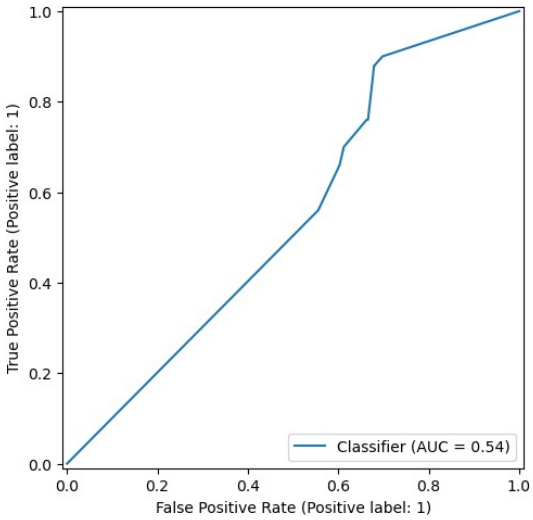
```
Balanced Accuracy: 0.5219
              precision    recall  f1-score   support

           0       0.95      0.88      0.92       972
           1       0.07      0.16      0.09        50

    accuracy                           0.85      1022
   macro avg       0.51      0.52      0.51      1022
weighted avg       0.91      0.85      0.88      1022
```



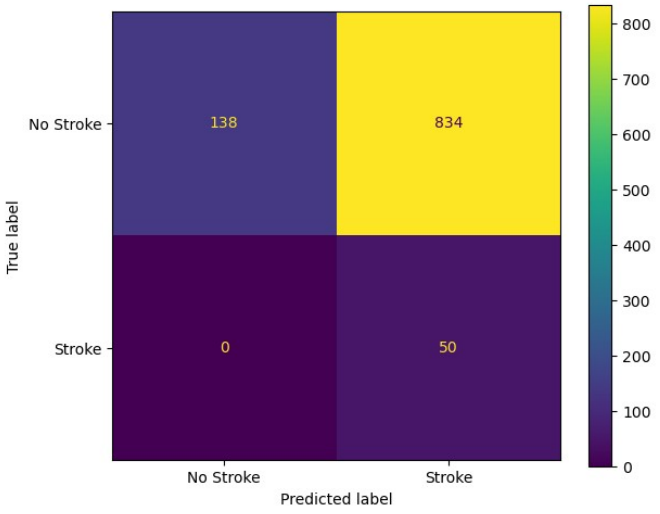


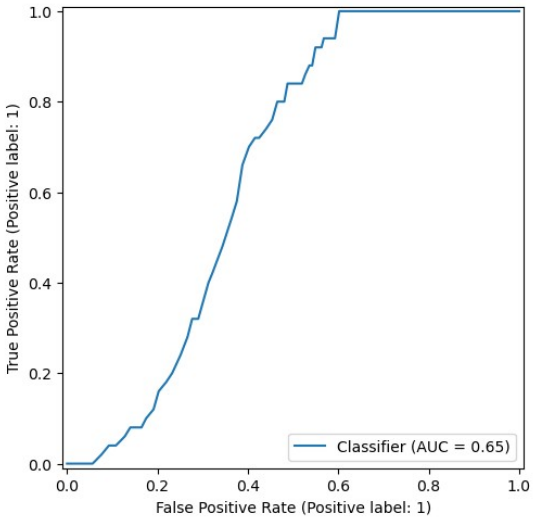

```
=== Decision Tree ===
--- Evaluation: Decision Tree ---
Balanced Accuracy: 0.5487
              precision    recall  f1-score   support

           0       0.96      0.34      0.50       972
           1       0.06      0.76      0.10        50

    accuracy                           0.36      1022
   macro avg       0.51      0.55      0.30      1022
weighted avg       0.92      0.36      0.48      1022
```





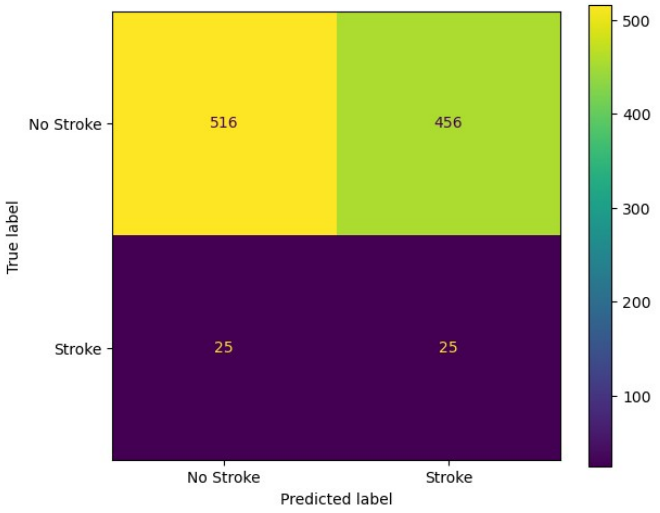

```
=== Random Forest ===
--- Evaluation: Random Forest ---
Balanced Accuracy: 0.5710
              precision    recall  f1-score   support

           0       1.00      0.14      0.25       972
           1       0.06      1.00      0.11        50

    accuracy                           0.18      1022
   macro avg       0.53      0.57      0.18      1022
weighted avg       0.95      0.18      0.24      1022
```

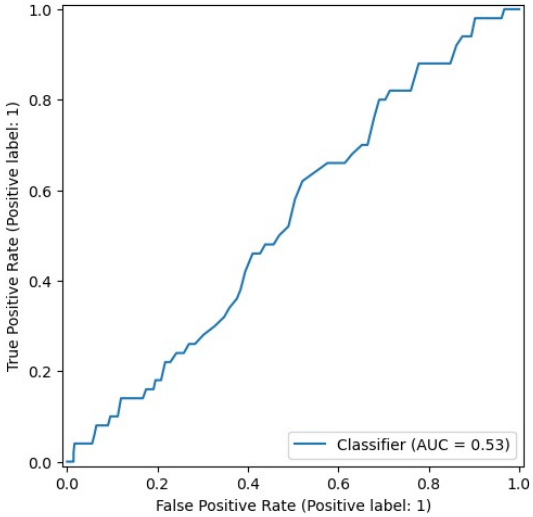




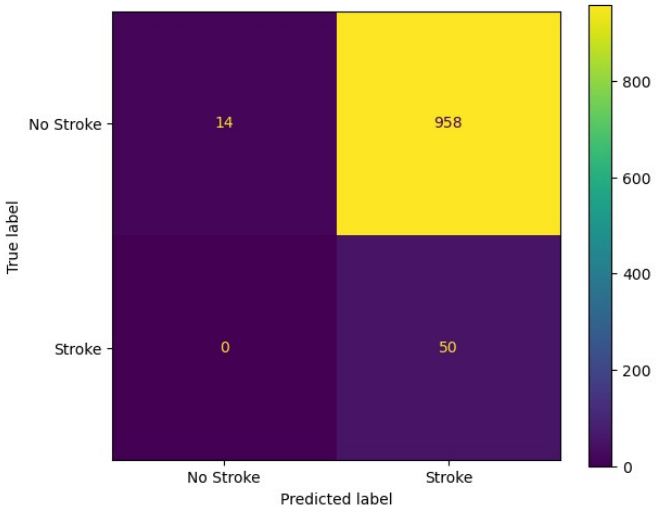

```
=== Extra Trees ===
--- Evaluation: Extra Trees ---
```

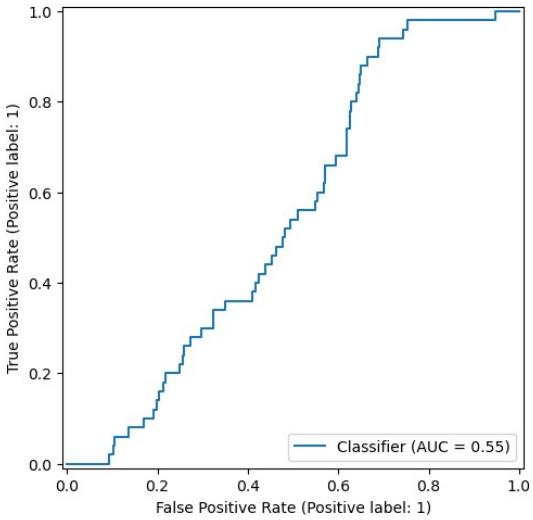
```
Balanced Accuracy: 0.5154
              precision    recall  f1-score   support

           0       0.95      0.53      0.68       972
           1       0.05      0.50      0.09        50

    accuracy                           0.53      1022
   macro avg       0.50      0.52      0.39      1022
weighted avg       0.91      0.53      0.65      1022
```



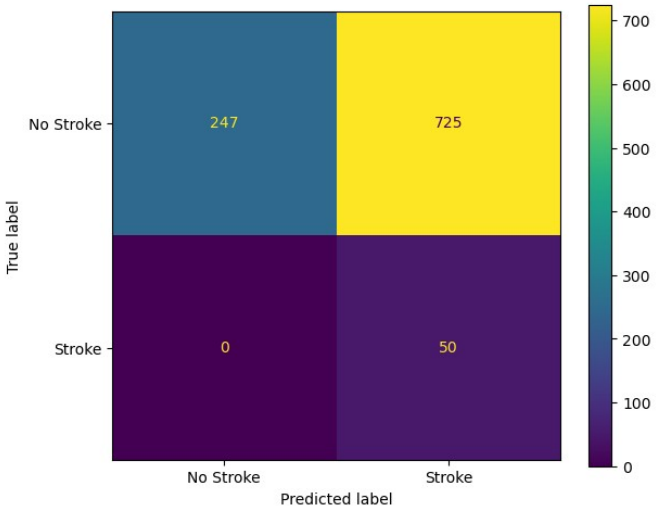


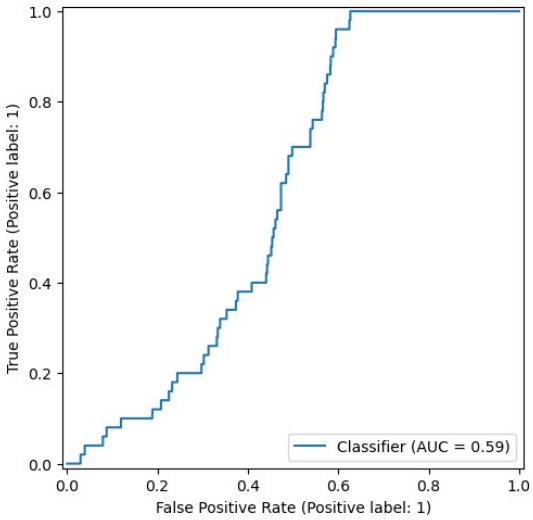

```
=== Gradient Boosting ===
--- Evaluation: Gradient Boosting ---
Balanced Accuracy: 0.5072
              precision    recall  f1-score   support

           0       1.00      0.01      0.03       972
           1       0.05      1.00      0.09        50

    accuracy                           0.06      1022
   macro avg       0.52      0.51      0.06      1022
weighted avg       0.95      0.06      0.03      1022
```





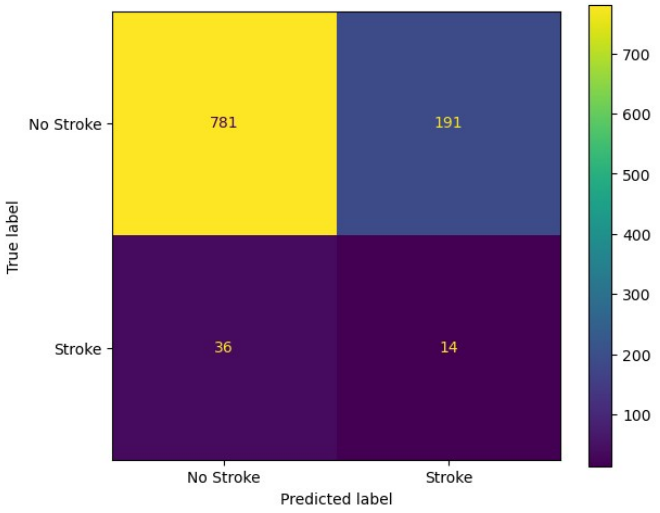

```
=== XGBoost ===
--- Evaluation: XGBoost ---
Balanced Accuracy: 0.6271
              precision    recall  f1-score   support

           0       1.00      0.25      0.41       972
           1       0.06      1.00      0.12        50

    accuracy                           0.29      1022
   macro avg       0.53      0.63      0.26      1022
weighted avg       0.95      0.29      0.39      1022
```

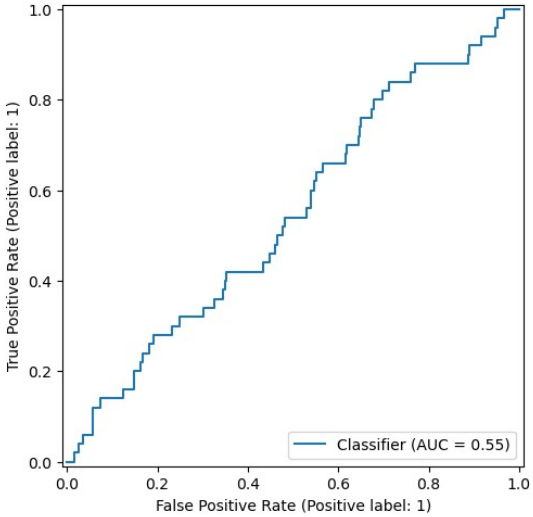




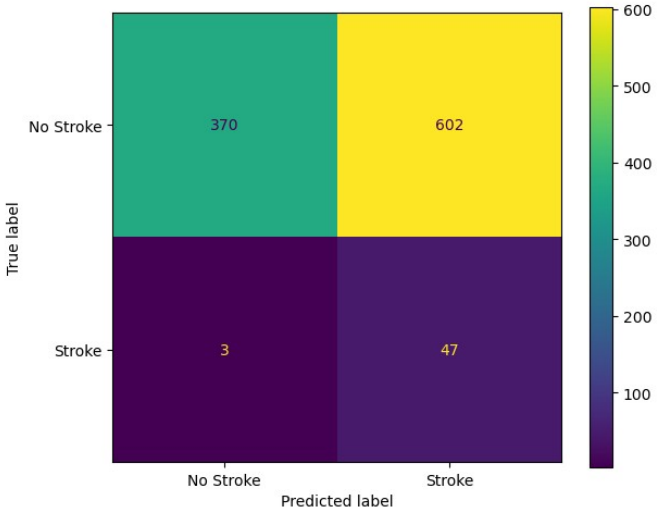

```
=== SVM ===
--- Evaluation: SVM ---
```

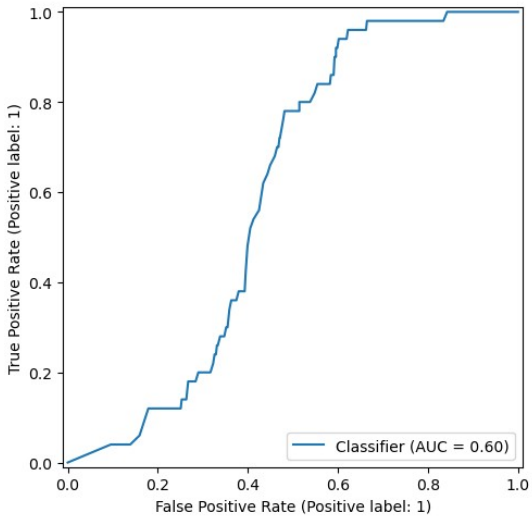
```
Balanced Accuracy: 0.5417
              precision    recall  f1-score   support

           0       0.96      0.80      0.87       972
           1       0.07      0.28      0.11        50

    accuracy                           0.78      1022
   macro avg       0.51      0.54      0.49      1022
weighted avg       0.91      0.78      0.84      1022
```



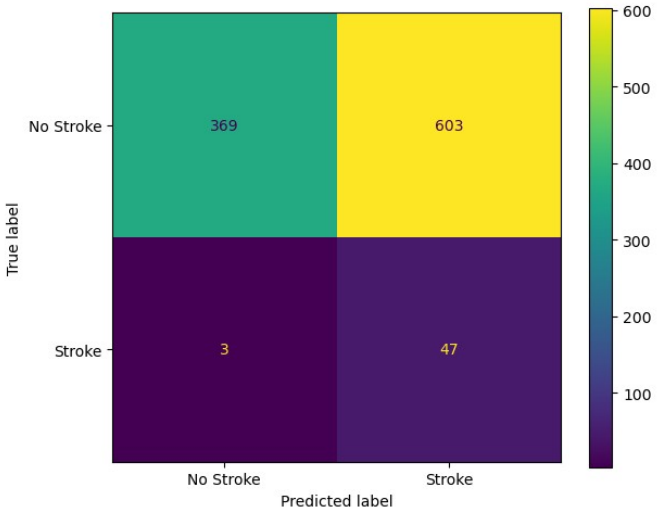


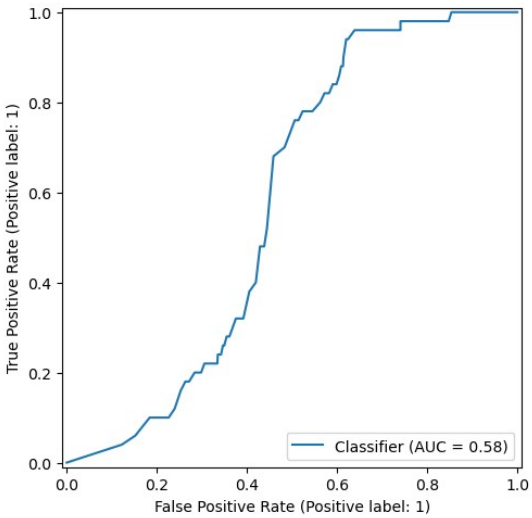

```
=== Bagging ===
--- Evaluation: Bagging ---
Balanced Accuracy: 0.6603
              precision    recall  f1-score   support

           0       0.99      0.38      0.55       972
           1       0.07      0.94      0.13        50

    accuracy                           0.41      1022
   macro avg       0.53      0.66      0.34      1022
weighted avg       0.95      0.41      0.53      1022
```





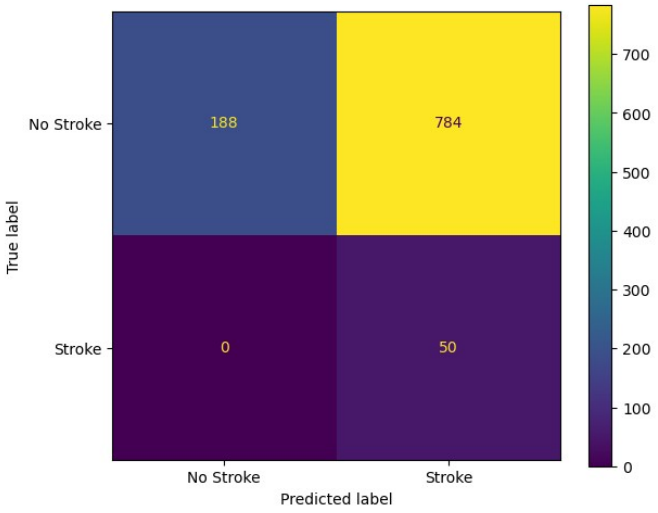

```
=== BalancedBagging ===
--- Evaluation: BalancedBagging ---
Balanced Accuracy: 0.6598
              precision    recall  f1-score   support

           0       0.99      0.38      0.55       972
           1       0.07      0.94      0.13        50

    accuracy                           0.41      1022
   macro avg       0.53      0.66      0.34      1022
weighted avg       0.95      0.41      0.53      1022
```

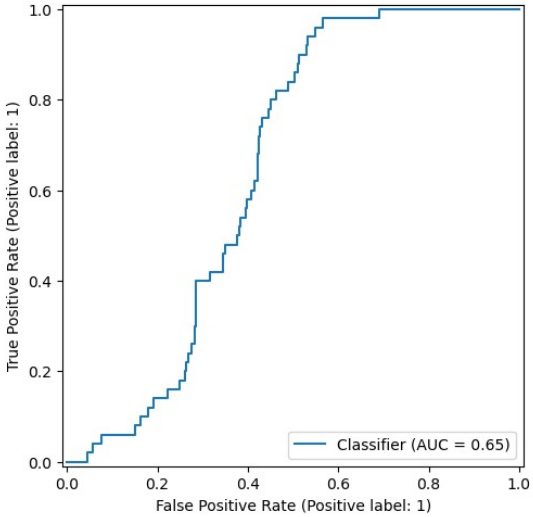




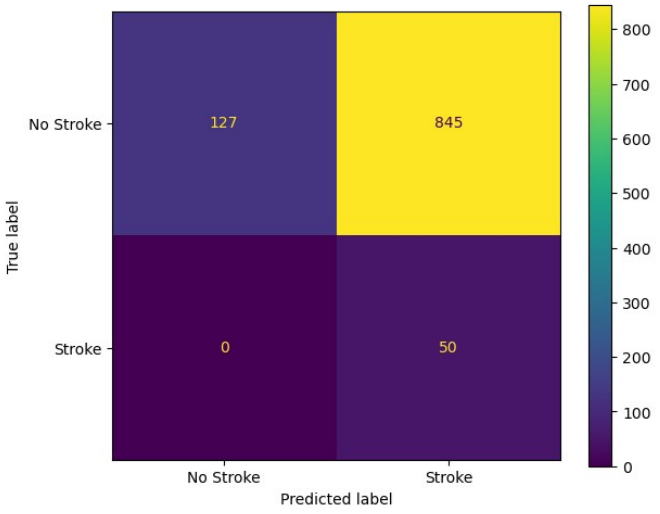

```
=== BalancedRandomForest ===
--- Evaluation: BalancedRandomForest ---
```

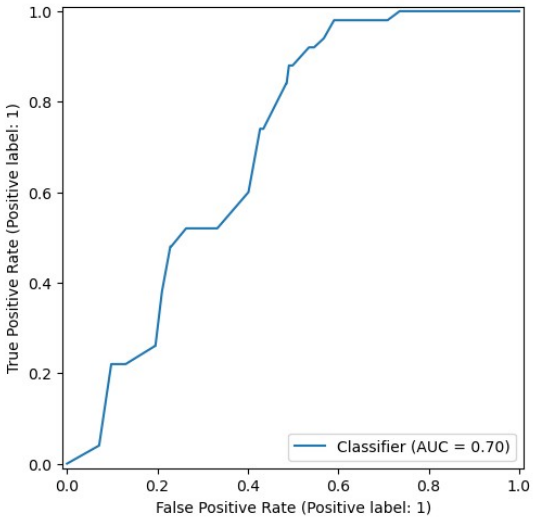
```
Balanced Accuracy: 0.5967
              precision    recall  f1-score   support

           0       1.00      0.19      0.32       972
           1       0.06      1.00      0.11        50

    accuracy                           0.23      1022
   macro avg       0.53      0.60      0.22      1022
weighted avg       0.95      0.23      0.31      1022
```



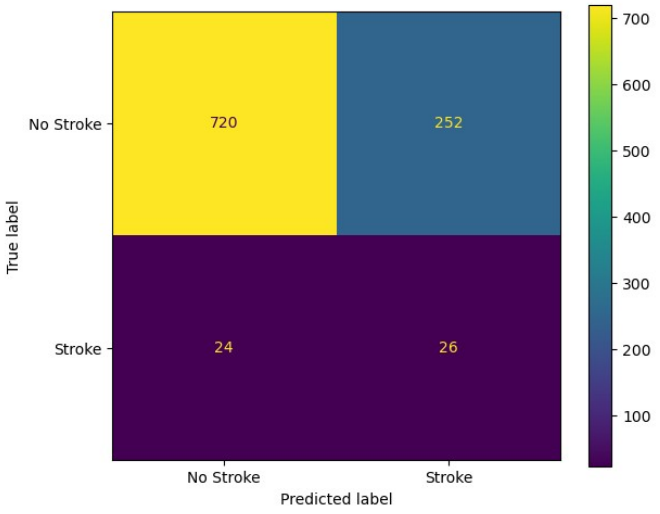


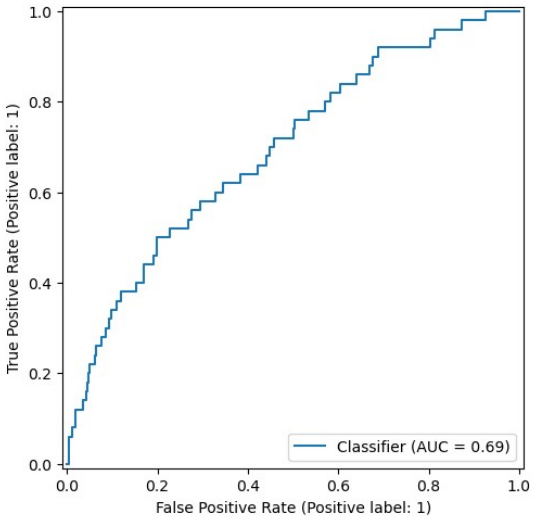

```
=== EasyEnsemble ===
--- Evaluation: EasyEnsemble ---
Balanced Accuracy: 0.5653
              precision    recall  f1-score   support

           0       1.00      0.13      0.23       972
           1       0.06      1.00      0.11        50

    accuracy                           0.17      1022
   macro avg       0.53      0.57      0.17      1022
weighted avg       0.95      0.17      0.22      1022
```






```
=== MLP ===
--- Evaluation: MLP ---
Balanced Accuracy: 0.6304
              precision    recall  f1-score   support

           0       0.97      0.74      0.84       972
           1       0.09      0.52      0.16        50

    accuracy                           0.73      1022
   macro avg       0.53      0.63      0.50      1022
weighted avg       0.92      0.73      0.81      1022
```






```
# Convert ResultsSmote to a DataFrame for easy sorting
```

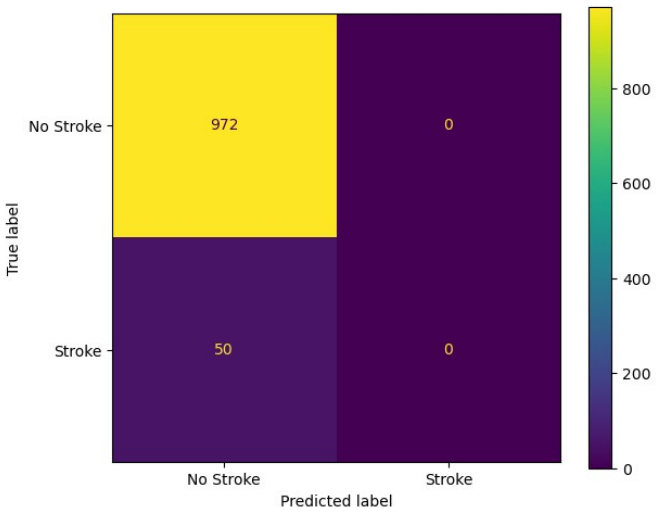
```python
results_smote_df = pd.DataFrame(ResultsSmote).T

# Sort by F1 or ROC AUC (choose your preferred metric)
top_n = 7  # Number of top models to select
top_models = results_smote_df.sort_values(by='Balanced Accuracy',
ascending=False).head(top_n)
print("Top performing models (by Balanced Accuracy):")
print(top_models)

# Extract their names
selected_model_names = top_models.index.tolist()

# Get the actual estimator objects
selected_estimators = [(name, best_estimators[name]) for name in
selected_model_names]
```

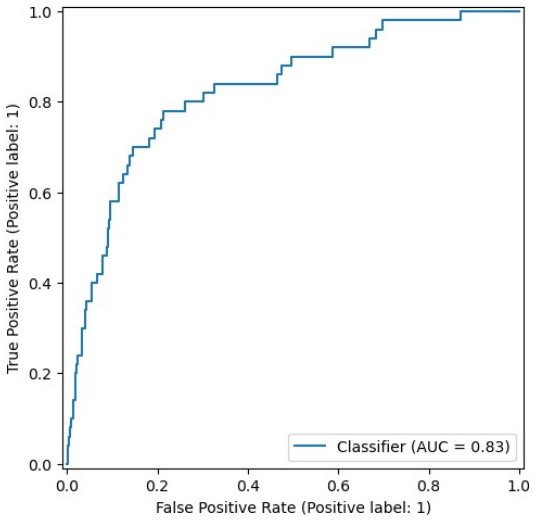
```
Top performing models (by Balanced Accuracy):
                     Balanced Accuracy  Precision  Recall        F1
ROC AUC
Logistic Regression           0.707263   0.090336    0.86  0.163498
0.839650
SGD                           0.689774   0.084314    0.86  0.153571
NaN
Bagging                       0.660329   0.072419    0.94  0.134478
0.595381
BalancedBagging               0.659815   0.072308    0.94  0.134286
0.576132
Naive Bayes                   0.659753   0.070912    0.98  0.132254
0.776317
MLP                           0.630370   0.093525    0.52  0.158537
0.693374
XGBoost                       0.627058   0.064516    1.00  0.121212
0.589352
```

```python
# Now you can use selected_estimators for stacking or voting
from sklearn.ensemble import StackingClassifier, VotingClassifier

# Example: Stacking
stacking_clf = StackingClassifier(
    estimators=selected_estimators,
    final_estimator=LogisticRegression(),
    cv=5
)
stacking_clf.fit(X_train, y_train)
evaluate_model(
    clf=stacking_clf,
    X_test=X_test,
    y_test=y_test,
    name="StackingClassifier (SMOTE)",
    results=ResultsSmote,
```

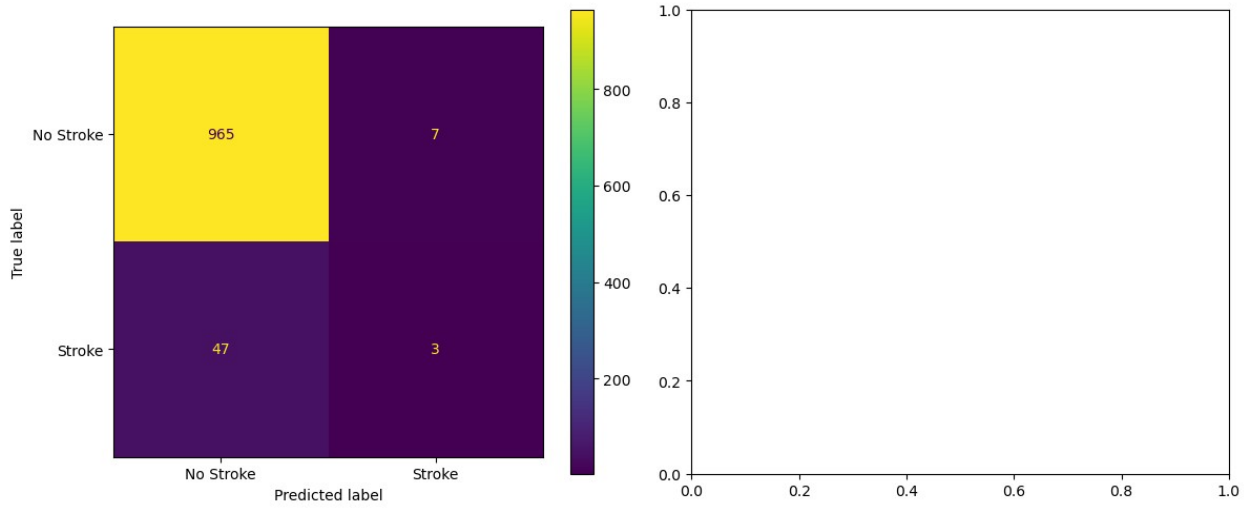
```
    show_plots=True
)

voting_clf = VotingClassifier(
    estimators=selected_estimators,
    voting='hard'
)
voting_clf.fit(X_train, y_train)
evaluate_model(
    clf=voting_clf,
    X_test=X_test,
    y_test=y_test,
    name="VotingClassifier (SMOTE)",
    results=ResultsSmote,
    show_plots=True
)
voting_clf = VotingClassifier(
    estimators=selected_estimators,
    voting='soft',  # Use 'soft' voting for better probability
estimates
)
```

```
--- Evaluation: StackingClassifier (SMOTE) ---
Balanced Accuracy: 0.5000
              precision    recall  f1-score   support

           0       0.95      1.00      0.97       972
           1       0.00      0.00      0.00        50

    accuracy                           0.95      1022
   macro avg       0.48      0.50      0.49      1022
weighted avg       0.90      0.95      0.93      1022
```





```
--- Evaluation: VotingClassifier (SMOTE) ---
Balanced Accuracy: 0.5264
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       972
           1       0.30      0.06      0.10        50

    accuracy                           0.95      1022
   macro avg       0.63      0.53      0.54      1022
weighted avg       0.92      0.95      0.93      1022
```




```
# Convert ResultsSmote to a DataFrame for easy sorting
results_smote_df = pd.DataFrame(ResultsSmote).T

# Sort by F1 or ROC AUC (choose your preferred metric)
top_n = 7  # Number of top models to select
top_models = results_smote_df.sort_values(by='Balanced Accuracy',
ascending=False).head(top_n)
print("Top performing models (by Balanced Accuracy):")
print(top_models)

# Extract their names
selected_model_names = top_models.index.tolist()

# Get the actual estimator objects
selected_estimators = [(name, best_estimators[name]) for name in
selected_model_names]
```

```
Top performing models (by Balanced Accuracy):
                     Balanced Accuracy  Precision  Recall        F1
ROC AUC
```

```
Logistic Regression            0.707263   0.090336    0.86  0.163498
0.839650
SGD                            0.689774   0.084314    0.86  0.153571
NaN
Bagging                        0.660329   0.072419    0.94  0.134478
0.595381
BalancedBagging                0.659815   0.072308    0.94  0.134286
0.576132
Naive Bayes                    0.659753   0.070912    0.98  0.132254
0.776317
MLP                            0.630370   0.093525    0.52  0.158537
0.693374
XGBoost                        0.627058   0.064516    1.00  0.121212
0.589352
```

6.3. Investigating Data Leakage: The Impact of Misapplying SMOTE

Data leakage is a common pitfall in machine learning, especially when using techniques like SMOTE. In this section, we will deliberately apply SMOTE to the entire dataset before the train-test split. While this is an incorrect practice, it serves to demonstrate how applying resampling techniques to data that will later be used for testing can lead to overly optimistic and misleading performance metrics. This experiment will highlight the importance of proper data handling procedures.

```
from imblearn.over_sampling import SMOTE

oversample = SMOTE()
X, y = oversample.fit_resample(X, y)

X_train_DL, X_test_DL, Y_train_DL, Y_test_DL =
train_test_split(X,y,test_size=0.20,random_state=0)
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
print(X_train_DL.shape)
print(X_test_DL.shape)
print(Y_train_DL.shape)
print(Y_test_DL.shape)
```

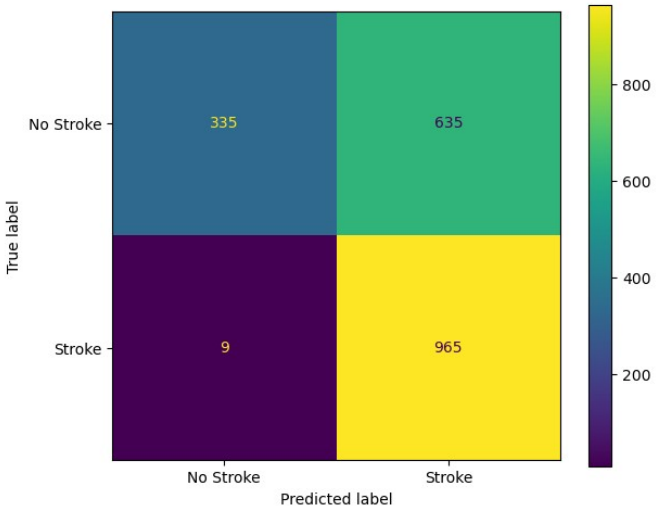
```
(7776, 17)
(1944, 17)
(7776,)
(1944,)
```

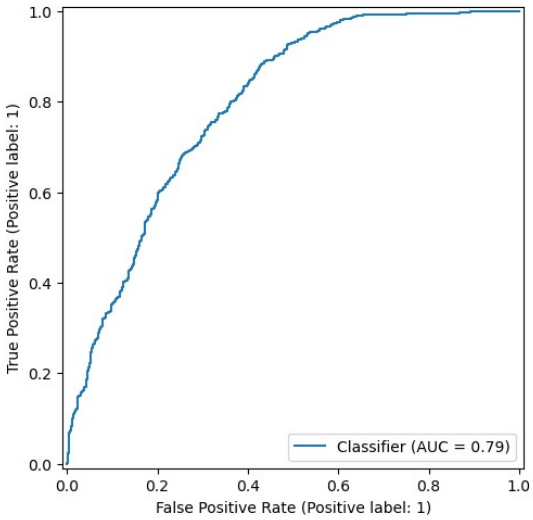
```
ResultsSmoteDataLeakage={}

from imblearn.pipeline import Pipeline as ImbPipeline
from imblearn.over_sampling import SMOTE
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression, SGDClassifier
#from imblearn.ensemble import BalancedBaggingClassifier,
```

```
BalancedRandomForestClassifier, EasyEnsembleClassifier
#from xgboost import XGBClassifier
#from sklearn.svm import SVC


best_estimators = {}
grid_models = [
    ('Naive Bayes', GaussianNB(), [{'var_smoothing': np.logspace(0, -
9, num=10)}]),
    ('Logistic Regression', LogisticRegression(), [{'C':
[0.25,0.5,0.75,1], 'random_state':[0], 'solver': ['newton-cg',
'lbfgs', 'liblinear'], 'max_iter':[1000]}]),
    ('SGD', SGDClassifier(), [{'loss': ['hinge', 'log_loss',
'modified_huber'], 'penalty': ['l2', 'l1', 'elasticnet'], 'alpha':
[0.0001, 0.001, 0.01, 0.1]}]),
    ('KNN', KNeighborsClassifier(), [{'n_neighbors': list(range(1,
11)), 'metric': ['euclidean', 'manhattan', 'chebyshev', 'minkowski'],
'weights': ['uniform', 'distance']}]),
    ('Decision Tree', DecisionTreeClassifier(), [{'criterion':
['gini', 'entropy'], 'max_depth': list(range(1, 10)), 'max_features':
[None, 'log2', 'sqrt'], 'random_state': [0]}]),
    ('Random Forest', RandomForestClassifier(), [{'n_estimators':
[100, 150, 200], 'criterion': ['gini', 'entropy'], 'random_state':
[0]}]),
    ('Extra Trees', ExtraTreesClassifier(), [{'n_estimators': [100,
150, 200], 'criterion': ['gini', 'entropy'], 'random_state': [0]}]),
    ('Gradient Boosting', GradientBoostingClassifier(),
[{'n_estimators': [100, 200, 300], 'learning_rate': [0.01, 0.05, 0.1],
'max_depth': [3, 4, 5], 'subsample': [0.8, 1.0], 'max_features':
['sqrt', 'log2', None]}]),
    ('XGBoost', XGBClassifier(objective="binary:logistic",
verbosity=0), [{'n_estimators': [50, 100], 'max_depth': [3, 5, 10],
'learning_rate': [0.1, 0.01], 'subsample': [0.8, 1],
'colsample_bytree': [0.5, 0.8], 'scale_pos_weight': [estimate],
'random_state': [0]}]),
    ('SVM', SVC(probability=True), [{'C': [1, 10], 'gamma': ['scale',
0.1], 'kernel': ['linear', 'rbf'], 'class_weight': ['balanced'],
'random_state': [0]}]),
    ('Bagging', BaggingClassifier(), [{'n_estimators': [100, 200],
'random_state': [0]}]),
    ('BalancedBagging', BalancedBaggingClassifier(), [{'n_estimators':
[100, 200], 'random_state': [0]}]),
    ('BalancedRandomForest', BalancedRandomForestClassifier(),
[{'n_estimators': [200, 300], 'criterion': ['gini', 'entropy'],
'max_depth': list(range(5, 10)), 'max_features': ['log2', 'sqrt'],
'random_state': [0]}]),
    ('EasyEnsemble', EasyEnsembleClassifier(), [{'n_estimators': [50,
100], 'random_state': [0]}]),
    ('MLP', MLPClassifier(), [{'hidden_layer_sizes': [(10,), (20,),
```

```
(25,)], 'max_iter': [500], 'learning_rate': ['constant'],
'learning_rate_init': [0.003, 0.03, 0.3], 'momentum': [0.02, 0.2],
'activation': ['relu', 'tanh']}])
]

scoring = {
    'balanced_accuracy': make_scorer(balanced_accuracy_score),
    'accuracy': make_scorer(accuracy_score),
    'precision': make_scorer(precision_score, zero_division=0),
    'recall': make_scorer(recall_score),
    'f1': make_scorer(f1_score),
    'roc_auc': make_scorer(roc_auc_score)
}
for name, estimator, param_grid in grid_models:
    # embed SMOTE + scaling + grid inside one pipeline
    pipeline = ImbPipeline([
        ('smote', SMOTE(random_state=42)),
        ('scale', StandardScaler()),
        ('clf', GridSearchCV(
            estimator=estimator,
            param_grid=param_grid,
            scoring=scoring,
            refit='roc_auc',
            cv=5,
            n_jobs=-1
        ))
    ])

    # fit on train
    pipeline.fit(X_train_DL, Y_train_DL)

    # pull out the fitted GridSearchCV
    grid = pipeline.named_steps['clf']
    best_estimators[name] = grid.best_estimator_
    print(f"\n=== {name} ===")
    evaluate_model(
        clf=grid.best_estimator_,   # the best estimator from the grid
search
        X_test=X_test_DL,
        y_test=Y_test_DL,
        name=name,
        results=ResultsSmoteDataLeakage,
        show_plots=True             # True if you want plots
    )



=== Naive Bayes ===
--- Evaluation: Naive Bayes ---
Balanced Accuracy: 0.6681
```

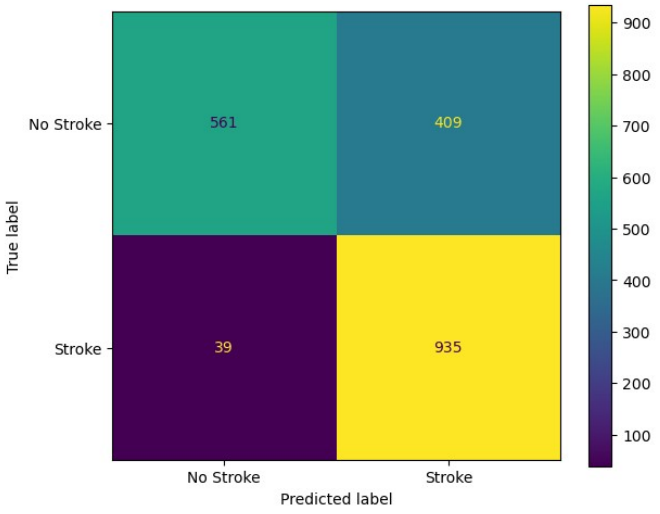
	precision	recall	f1-score	support
0	0.97	0.35	0.51	970
1	0.60	0.99	0.75	974
accuracy			0.67	1944
macro avg	0.79	0.67	0.63	1944
weighted avg	0.79	0.67	0.63	1944

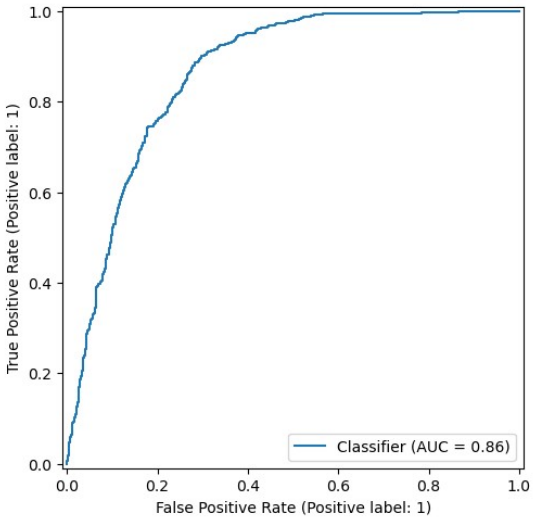




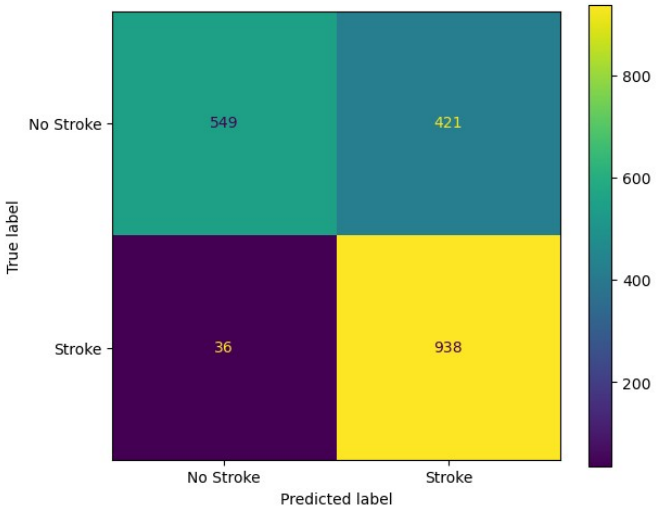

```
=== Logistic Regression ===
--- Evaluation: Logistic Regression ---
Balanced Accuracy: 0.7692
```

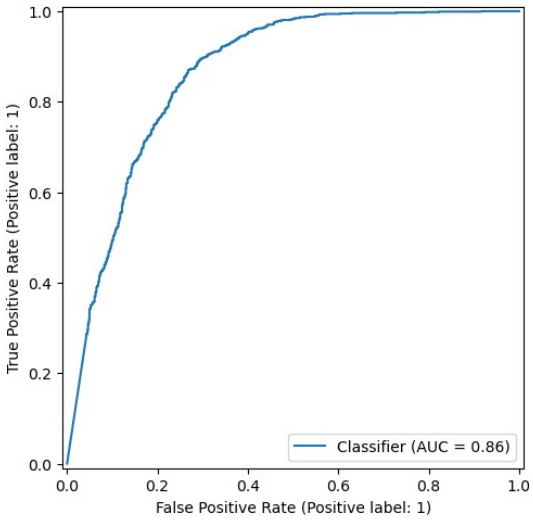
	precision	recall	f1-score	support
0	0.94	0.58	0.71	970
1	0.70	0.96	0.81	974
accuracy			0.77	1944
macro avg	0.82	0.77	0.76	1944
weighted avg	0.82	0.77	0.76	1944





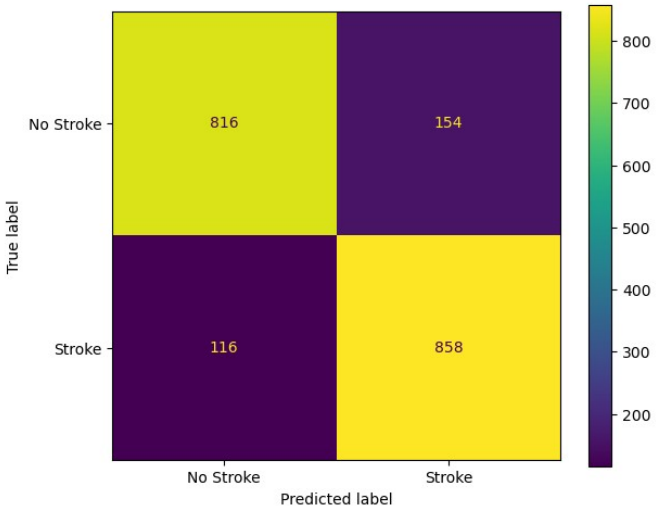

```
=== SGD ===
--- Evaluation: SGD ---
Balanced Accuracy: 0.7645
              precision    recall  f1-score   support

           0       0.94      0.57      0.71       970
           1       0.69      0.96      0.80       974

    accuracy                           0.76      1944
   macro avg       0.81      0.76      0.76      1944
weighted avg       0.81      0.76      0.76      1944
```

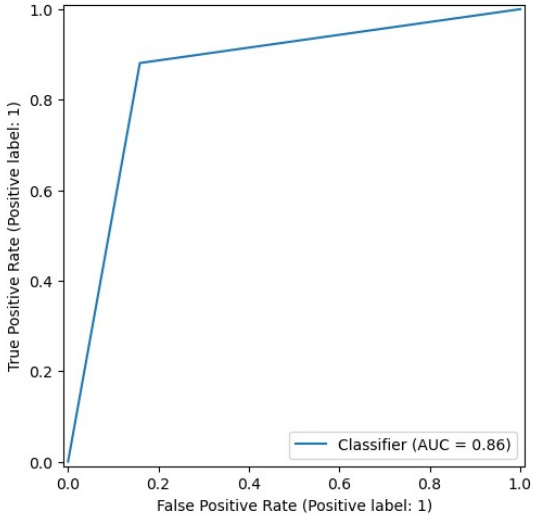




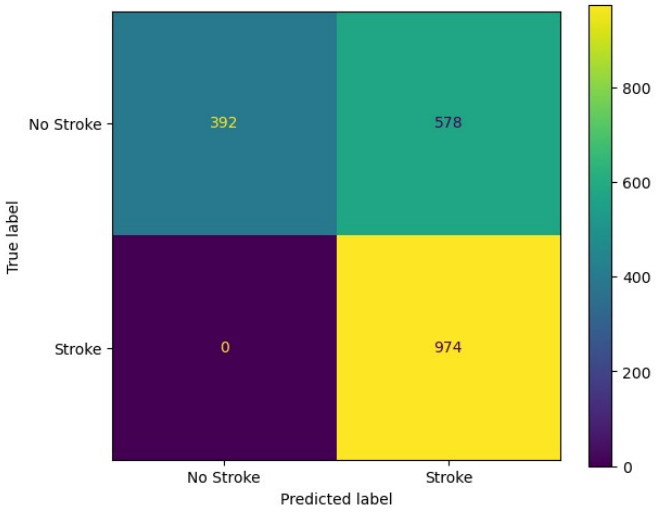

```
=== KNN ===
--- Evaluation: KNN ---
```

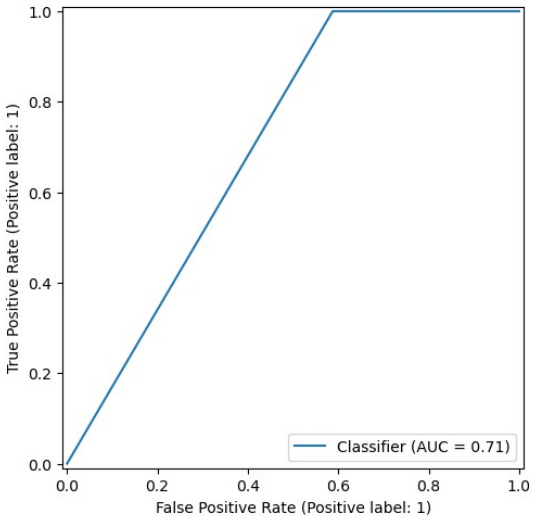
```
Balanced Accuracy: 0.8611
              precision    recall  f1-score   support

           0       0.88      0.84      0.86       970
           1       0.85      0.88      0.86       974

    accuracy                           0.86      1944
   macro avg       0.86      0.86      0.86      1944
weighted avg       0.86      0.86      0.86      1944
```



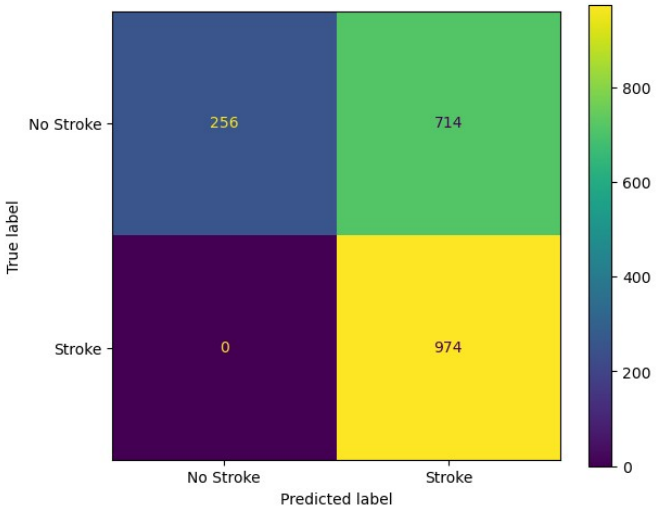


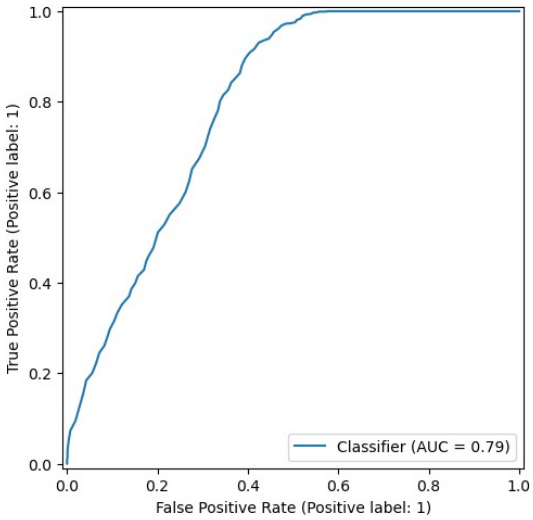

```
=== Decision Tree ===
--- Evaluation: Decision Tree ---
Balanced Accuracy: 0.7021
              precision    recall  f1-score   support

           0       1.00      0.40      0.58       970
           1       0.63      1.00      0.77       974

    accuracy                           0.70      1944
   macro avg       0.81      0.70      0.67      1944
weighted avg       0.81      0.70      0.67      1944
```





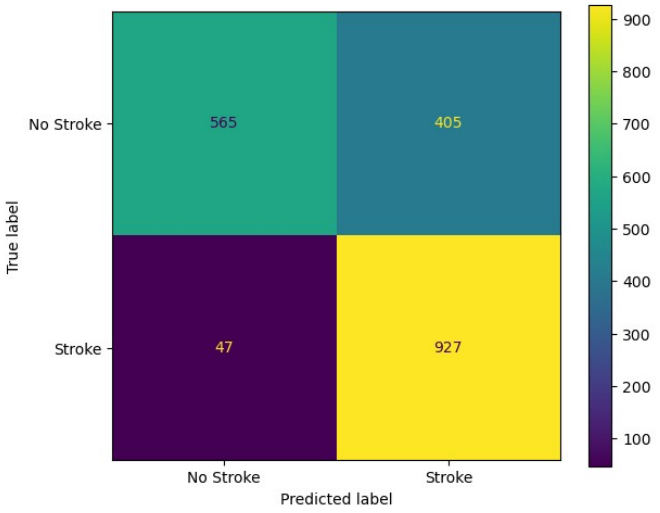

```
=== Random Forest ===
--- Evaluation: Random Forest ---
Balanced Accuracy: 0.6320
              precision    recall  f1-score   support

           0       1.00      0.26      0.42       970
           1       0.58      1.00      0.73       974

    accuracy                           0.63      1944
   macro avg       0.79      0.63      0.57      1944
weighted avg       0.79      0.63      0.58      1944
```

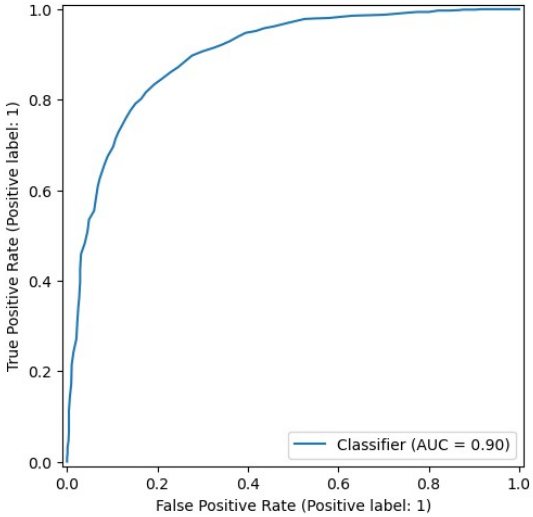




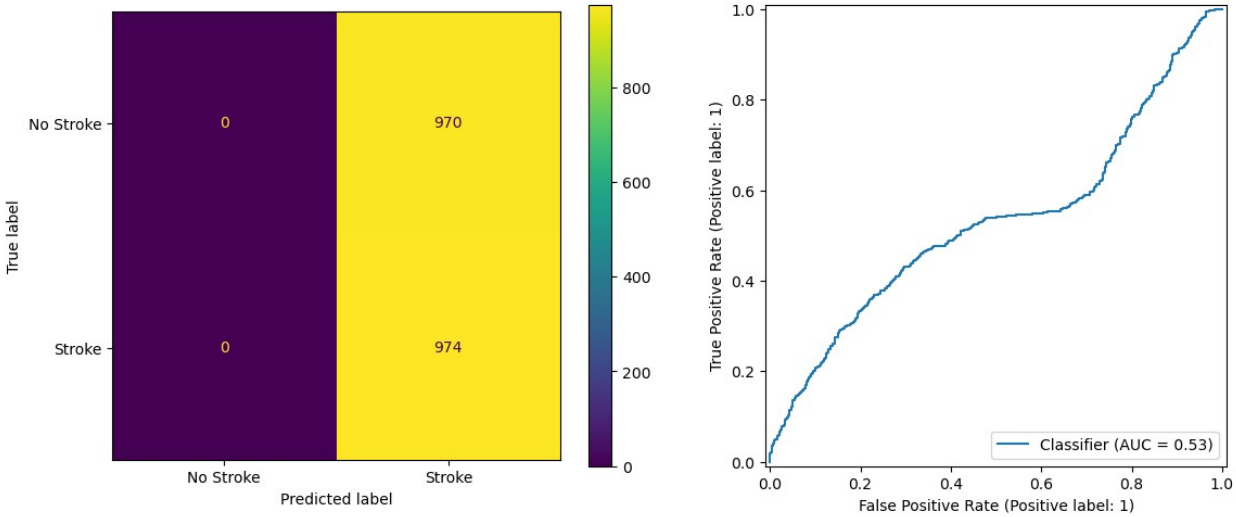

```
=== Extra Trees ===
--- Evaluation: Extra Trees ---
```

```
Balanced Accuracy: 0.7671
              precision    recall  f1-score   support

           0       0.92      0.58      0.71       970
           1       0.70      0.95      0.80       974

    accuracy                           0.77      1944
   macro avg       0.81      0.77      0.76      1944
weighted avg       0.81      0.77      0.76      1944
```

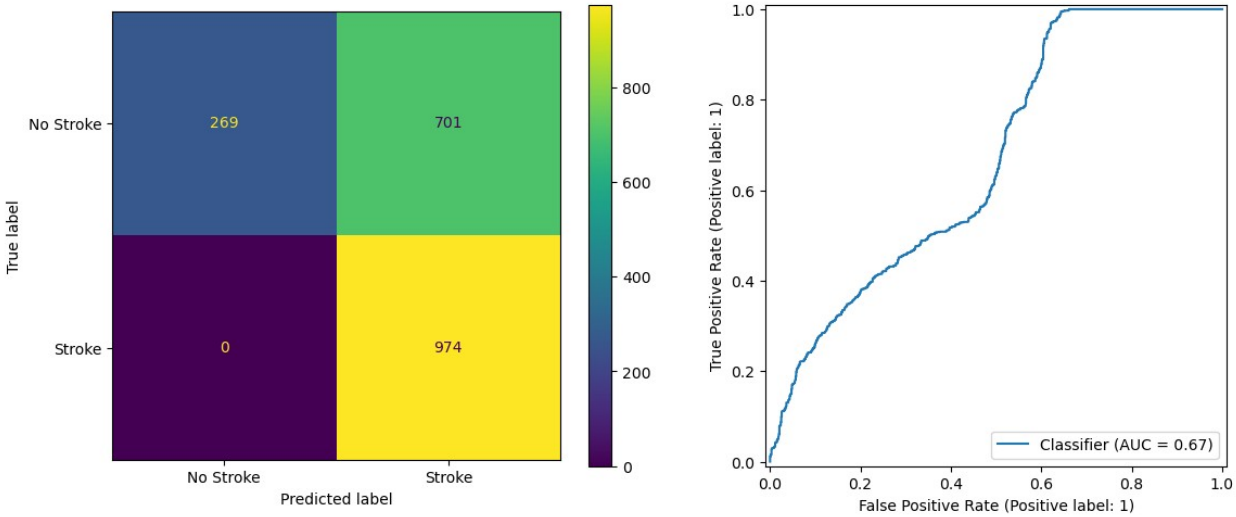





```
=== Gradient Boosting ===
--- Evaluation: Gradient Boosting ---
Balanced Accuracy: 0.5000
              precision    recall  f1-score   support

           0       0.00      0.00      0.00       970
           1       0.50      1.00      0.67       974

    accuracy                           0.50      1944
   macro avg       0.25      0.50      0.33      1944
weighted avg       0.25      0.50      0.33      1944
```

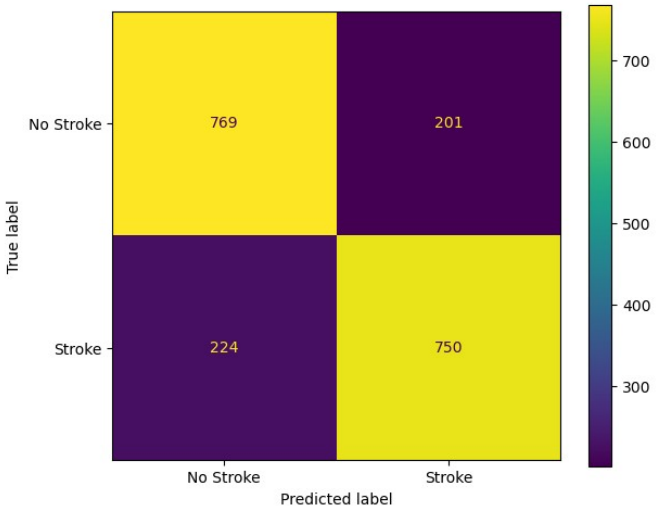


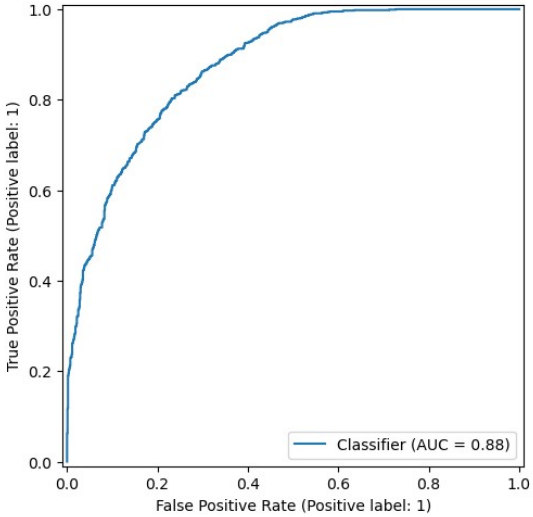

```
=== XGBoost ===
--- Evaluation: XGBoost ---
Balanced Accuracy: 0.6387
              precision    recall  f1-score   support

           0       1.00      0.28      0.43       970
           1       0.58      1.00      0.74       974

    accuracy                           0.64      1944
   macro avg       0.79      0.64      0.58      1944
weighted avg       0.79      0.64      0.59      1944
```



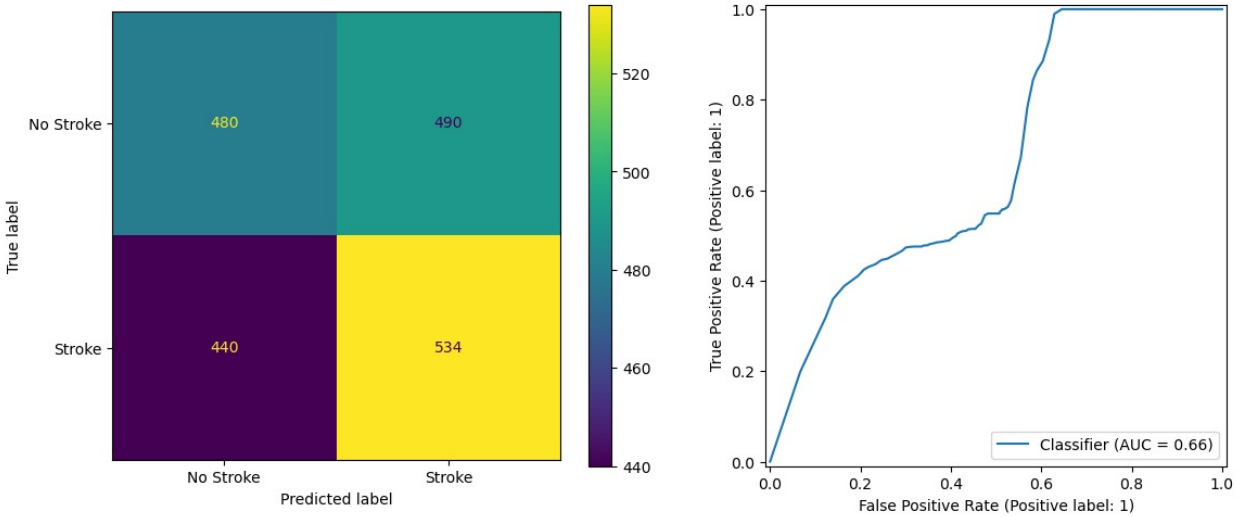

```
=== SVM ===
--- Evaluation: SVM ---
```

```
Balanced Accuracy: 0.7814
              precision    recall  f1-score   support

           0       0.77      0.79      0.78       970
           1       0.79      0.77      0.78       974

    accuracy                           0.78      1944
   macro avg       0.78      0.78      0.78      1944
weighted avg       0.78      0.78      0.78      1944
```

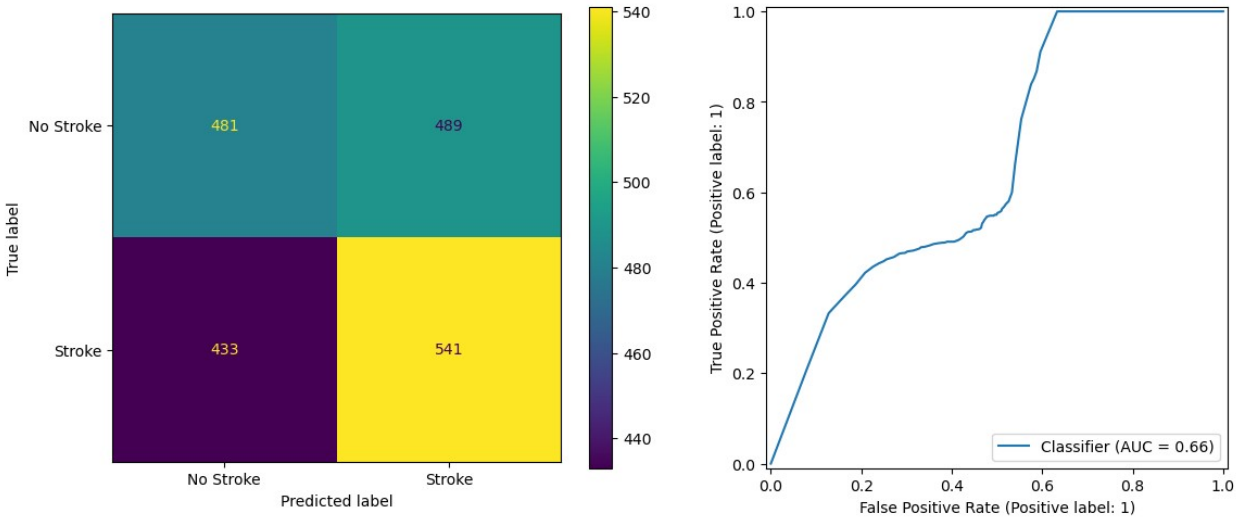





```
=== Bagging ===
--- Evaluation: Bagging ---
Balanced Accuracy: 0.5215
              precision    recall  f1-score   support

           0       0.52      0.49      0.51       970
           1       0.52      0.55      0.53       974

    accuracy                           0.52      1944
   macro avg       0.52      0.52      0.52      1944
weighted avg       0.52      0.52      0.52      1944
```

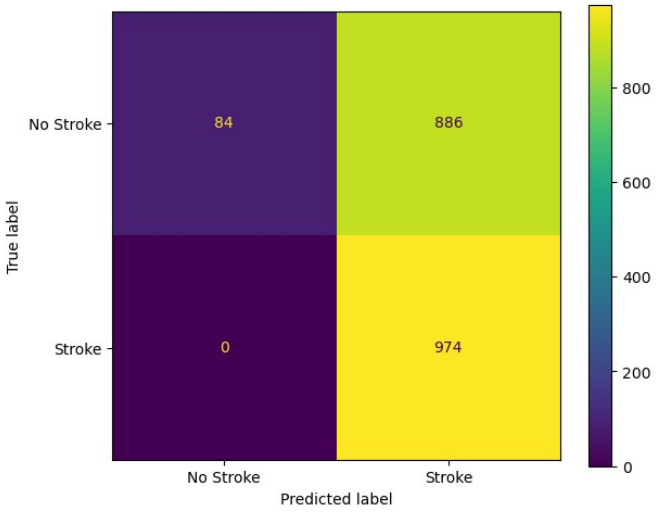


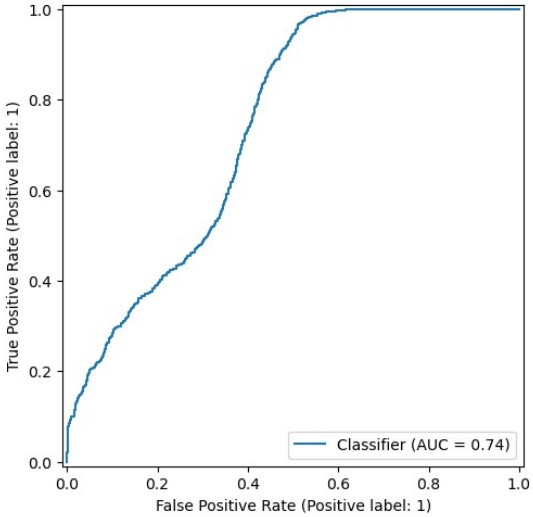

```
=== BalancedBagging ===
--- Evaluation: BalancedBagging ---
Balanced Accuracy: 0.5257
              precision    recall  f1-score   support

           0       0.53      0.50      0.51       970
           1       0.53      0.56      0.54       974

    accuracy                           0.53      1944
   macro avg       0.53      0.53      0.53      1944
weighted avg       0.53      0.53      0.53      1944
```



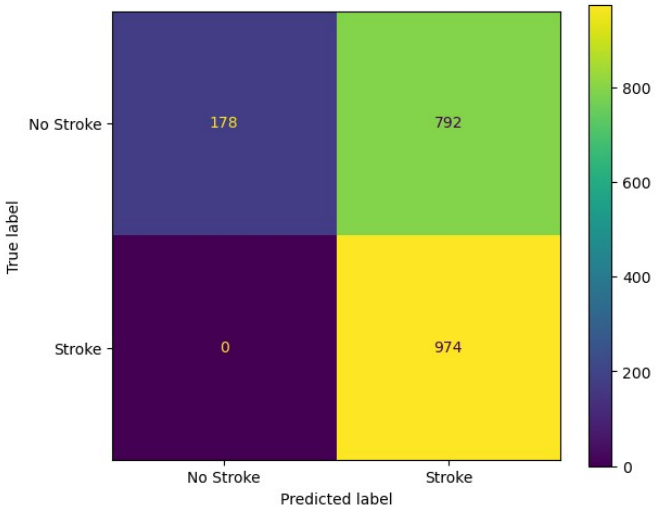

```
=== BalancedRandomForest ===
--- Evaluation: BalancedRandomForest ---
```

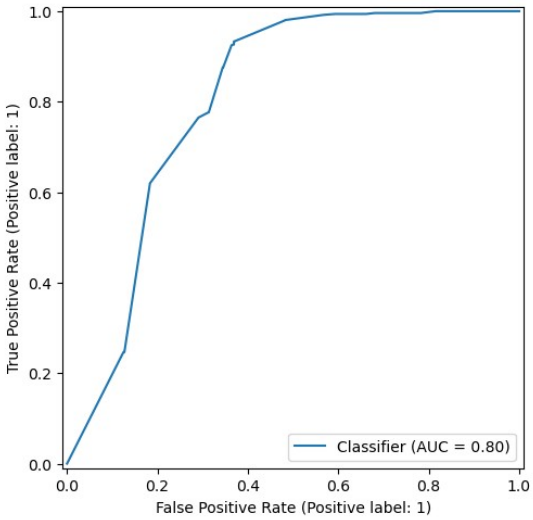
```
Balanced Accuracy: 0.5433
              precision    recall  f1-score   support

           0       1.00      0.09      0.16       970
           1       0.52      1.00      0.69       974

    accuracy                           0.54      1944
   macro avg       0.76      0.54      0.42      1944
weighted avg       0.76      0.54      0.42      1944
```



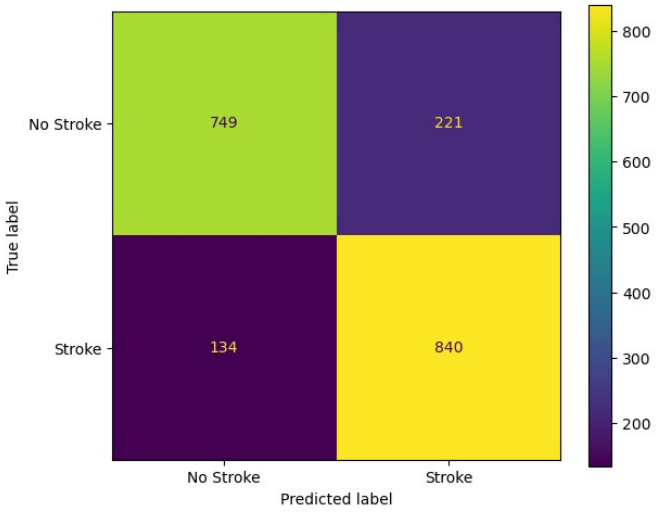


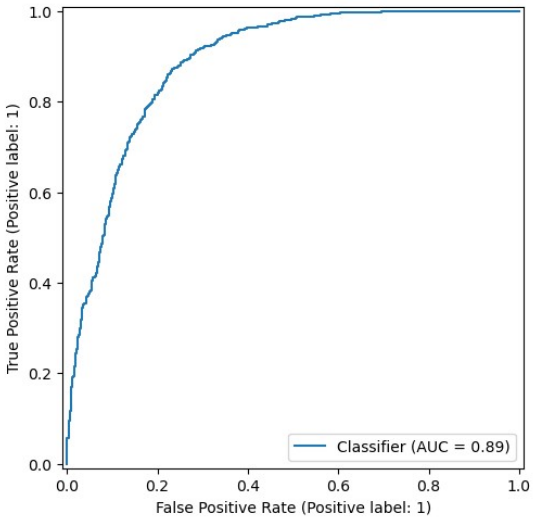

```
=== EasyEnsemble ===
--- Evaluation: EasyEnsemble ---
Balanced Accuracy: 0.5918
              precision    recall  f1-score   support

           0       1.00      0.18      0.31       970
           1       0.55      1.00      0.71       974

    accuracy                           0.59      1944
   macro avg       0.78      0.59      0.51      1944
weighted avg       0.78      0.59      0.51      1944
```






```
=== MLP ===
--- Evaluation: MLP ---
Balanced Accuracy: 0.8173
              precision    recall  f1-score   support

           0       0.85      0.77      0.81       970
           1       0.79      0.86      0.83       974

    accuracy                           0.82      1944
   macro avg       0.82      0.82      0.82      1944
weighted avg       0.82      0.82      0.82      1944
```






```
# Convert ResultsSmote to a DataFrame for easy sorting
results_smote_df = pd.DataFrame(ResultsSmoteDataLeakage).T
```

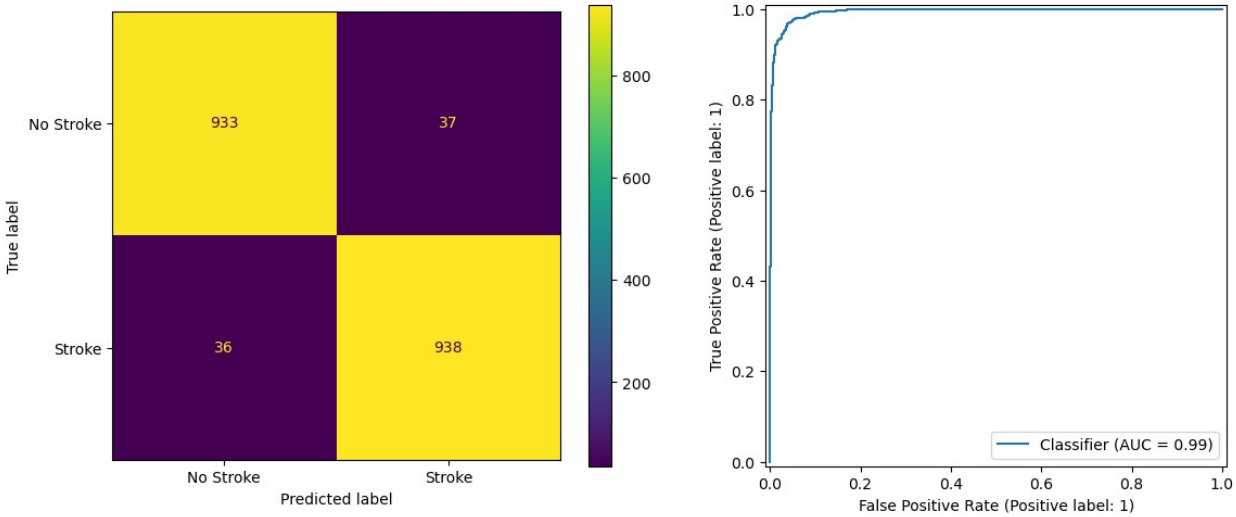
```python
# Sort by F1 or ROC AUC (choose your preferred metric)
top_n = 7  # Number of top models to select
top_models = results_smote_df.sort_values(by='Balanced Accuracy',
ascending=False).head(top_n)
print("Top performing models (by Balanced Accuracy):")
print(top_models)

# Extract their names
selected_model_names = top_models.index.tolist()

# Get the actual estimator objects
selected_estimators = [(name, best_estimators[name]) for name in
selected_model_names]
```

```
Top performing models (by Balanced Accuracy):
                     Balanced Accuracy  Precision    Recall        F1
\
KNN                           0.861070   0.847826  0.880903  0.864048

MLP                           0.817294   0.791706  0.862423  0.825553

SVM                           0.781402   0.788644  0.770021  0.779221

Logistic Regression           0.769155   0.695685  0.959959  0.806730

Extra Trees                   0.767110   0.695946  0.951745  0.803990

SGD                           0.764509   0.690213  0.963039  0.804115

Decision Tree                 0.702062   0.627577  1.000000  0.771180


                      ROC AUC
KNN                  0.861070
MLP                  0.887517
SVM                  0.875856
Logistic Regression  0.863699
Extra Trees          0.897552
SGD                  0.864325
Decision Tree        0.706186
```

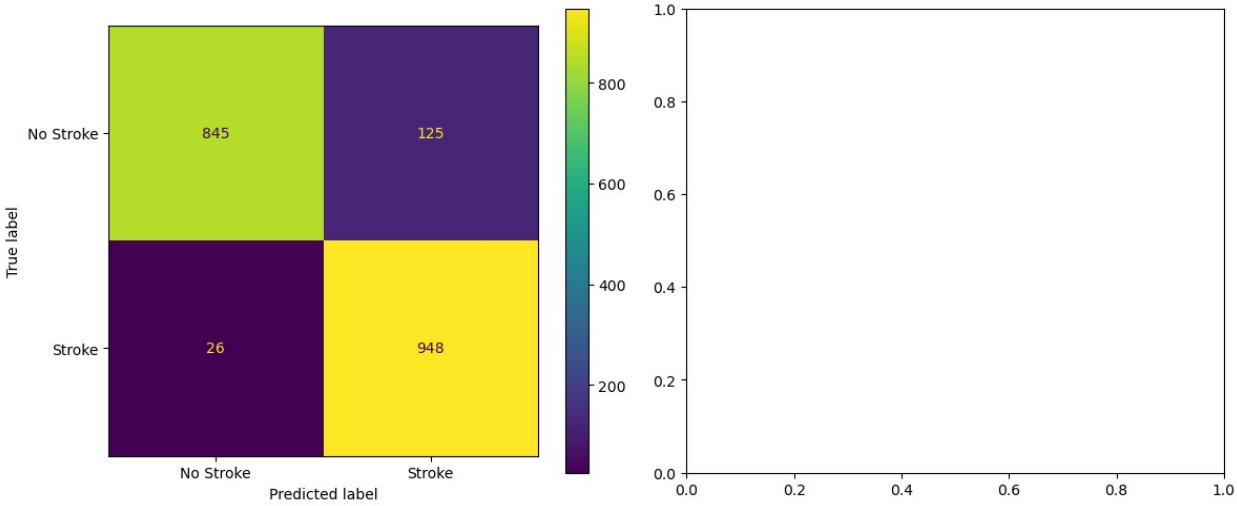
```python
# Now you can use selected_estimators for stacking or voting
from sklearn.ensemble import StackingClassifier, VotingClassifier

# Example: Stacking
stacking_clf = StackingClassifier(
    estimators=selected_estimators,
    final_estimator=LogisticRegression(),
    cv=5
)
stacking_clf.fit(X_train_DL, Y_train_DL)
```

```
evaluate_model(
    clf=stacking_clf,
    X_test=X_test_DL,
    y_test=Y_test_DL,
    name="StackingClassifier (SMOTE)",
    results=ResultsSmoteDataLeakage,
    show_plots=True
)

voting_clf = VotingClassifier(
    estimators=selected_estimators,
    voting='hard'
)
voting_clf.fit(X_train_DL, Y_train_DL)
evaluate_model(
    clf=voting_clf,
    X_test=X_test_DL,
    y_test=Y_test_DL,
    name="VotingClassifier (SMOTE)",
    results=ResultsSmoteDataLeakage,
    show_plots=True
)
voting_clf = VotingClassifier(
    estimators=selected_estimators,
    voting='soft',  # Use 'soft' voting for better probability
estimates
)
```

```
--- Evaluation: StackingClassifier (SMOTE) ---
Balanced Accuracy: 0.9624
              precision    recall  f1-score   support

           0       0.96      0.96      0.96       970
           1       0.96      0.96      0.96       974

    accuracy                           0.96      1944
   macro avg       0.96      0.96      0.96      1944
weighted avg       0.96      0.96      0.96      1944
```




```
--- Evaluation: VotingClassifier (SMOTE) ---
Balanced Accuracy: 0.9222
              precision    recall  f1-score   support

           0       0.97      0.87      0.92       970
           1       0.88      0.97      0.93       974

    accuracy                           0.92      1944
   macro avg       0.93      0.92      0.92      1944
weighted avg       0.93      0.92      0.92      1944
```



7. Neural Network

Finally, we will explore the application of a neural network for the stroke prediction task. We will design, train, and evaluate a neural network, paying attention to its architecture, activation

functions, and optimization, and compare its performance against the traditional machine learning models. (SMOTE will be used for the training data for the NN).

- Creating sequnetial ANN Network
- Creating 5 layers Network
- Activation is "Relu"
- Last layer is output layer
- Problem is binary classification thats way output node is 1 and activation is "sigmoid"

```
model=keras.Sequential([
    keras.layers.Dense(4800,input_shape=[17], activation='relu'),
    keras.layers.Dense(2000, activation='relu'),
    keras.layers.Dense(1000, activation='relu'),
    keras.layers.Dense(1000, activation='relu'),
    keras.layers.Dense(1,activation="sigmoid")
])
model.summary()
```

```
Model: "sequential"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense (Dense)                   │ (None, 4800)           │        86,400 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                 │ (None, 2000)           │     9,602,000 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 1000)           │     2,001,000 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                 │ (None, 1000)           │     1,001,000 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_4 (Dense)                 │ (None, 1)              │         1,001 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
```

```
 Total params: 12,691,401 (48.41 MB)
```

```
 Trainable params: 12,691,401 (48.41 MB)
```

```
 Non-trainable params: 0 (0.00 B)
```

```
model.compile(loss='binary_crossentropy',
optimizer='adam',metrics=['accuracy'])

train = model.fit(X_smote, y_smote, epochs=100,batch_size=100)
```

```
Epoch 1/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 7s 75ms/step - accuracy: 0.7478 - loss:
0.4931
Epoch 2/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.8454 - loss:
0.3481
Epoch 3/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.8699 - loss:
0.3053
Epoch 4/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9014 - loss:
0.2458
Epoch 5/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9153 - loss:
0.2202
Epoch 6/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9272 - loss:
0.1963
Epoch 7/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9390 - loss:
0.1730
Epoch 8/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9410 - loss:
0.1546
Epoch 9/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9465 - loss:
0.1449
Epoch 10/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9474 - loss:
0.1469
Epoch 11/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9500 - loss:
0.1362
Epoch 12/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9547 - loss:
0.1285
Epoch 13/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9576 - loss:
0.1166
Epoch 14/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9548 - loss:
0.1216
Epoch 15/100
```

```
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9649 - loss:
0.0973
Epoch 16/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9618 - loss:
0.1036
Epoch 17/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9665 - loss:
0.0979
Epoch 18/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9642 - loss:
0.0934
Epoch 19/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9667 - loss:
0.0951
Epoch 20/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9669 - loss:
0.0916
Epoch 21/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9686 - loss:
0.0880
Epoch 22/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9702 - loss:
0.0763
Epoch 23/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9767 - loss:
0.0698
Epoch 24/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9707 - loss:
0.0796
Epoch 25/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9711 - loss:
0.0824
Epoch 26/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9731 - loss:
0.0734
Epoch 27/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9737 - loss:
0.0767
Epoch 28/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9729 - loss:
0.0702
Epoch 29/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9808 - loss:
0.0646
Epoch 30/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9791 - loss:
0.0653
Epoch 31/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 72ms/step - accuracy: 0.9770 - loss:
```

```
0.0720
Epoch 32/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9817 - loss:
0.0562
Epoch 33/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9804 - loss:
0.0650
Epoch 34/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9778 - loss:
0.0656
Epoch 35/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9810 - loss:
0.0530
Epoch 36/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9806 - loss:
0.0569
Epoch 37/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9833 - loss:
0.0473
Epoch 38/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9792 - loss:
0.0614
Epoch 39/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9822 - loss:
0.0535
Epoch 40/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9832 - loss:
0.0478
Epoch 41/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9787 - loss:
0.0637
Epoch 42/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9830 - loss:
0.0531
Epoch 43/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9823 - loss:
0.0547
Epoch 44/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9866 - loss:
0.0441
Epoch 45/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9778 - loss:
0.0575
Epoch 46/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9789 - loss:
0.0595
Epoch 47/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9842 - loss:
0.0479
```

```
Epoch 48/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9778 - loss:
0.0563
Epoch 49/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9834 - loss:
0.0453
Epoch 50/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9832 - loss:
0.0548
Epoch 51/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9842 - loss:
0.0446
Epoch 52/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9866 - loss:
0.0380
Epoch 53/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9861 - loss:
0.0365
Epoch 54/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9833 - loss:
0.0446
Epoch 55/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9871 - loss:
0.0372
Epoch 56/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9860 - loss:
0.0443
Epoch 57/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9847 - loss:
0.0432
Epoch 58/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9845 - loss:
0.0458
Epoch 59/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9873 - loss:
0.0315
Epoch 60/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9810 - loss:
0.0493
Epoch 61/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9872 - loss:
0.0378
Epoch 62/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9911 - loss:
0.0267
Epoch 63/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9878 - loss:
0.0369
Epoch 64/100
```

```
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9908 - loss:
0.0233
Epoch 65/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9868 - loss:
0.0349
Epoch 66/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9828 - loss:
0.0502
Epoch 67/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9846 - loss:
0.0456
Epoch 68/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9857 - loss:
0.0398
Epoch 69/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9888 - loss:
0.0308
Epoch 70/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9830 - loss:
0.0441
Epoch 71/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 73ms/step - accuracy: 0.9882 - loss:
0.0346
Epoch 72/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9881 - loss:
0.0358
Epoch 73/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9904 - loss:
0.0279
Epoch 74/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 79ms/step - accuracy: 0.9891 - loss:
0.0316
Epoch 75/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 77ms/step - accuracy: 0.9868 - loss:
0.0381
Epoch 76/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9898 - loss:
0.0255
Epoch 77/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9922 - loss:
0.0230
Epoch 78/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 75ms/step - accuracy: 0.9881 - loss:
0.0347
Epoch 79/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 74ms/step - accuracy: 0.9861 - loss:
0.0417
Epoch 80/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 79ms/step - accuracy: 0.9877 - loss:
```

```
0.0315
Epoch 81/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 82ms/step - accuracy: 0.9883 - loss:
0.0334
Epoch 82/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 7s 89ms/step - accuracy: 0.9893 - loss:
0.0329
Epoch 83/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 7s 83ms/step - accuracy: 0.9894 - loss:
0.0323
Epoch 84/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 83ms/step - accuracy: 0.9875 - loss:
0.0317
Epoch 85/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 81ms/step - accuracy: 0.9903 - loss:
0.0275
Epoch 86/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 79ms/step - accuracy: 0.9912 - loss:
0.0256
Epoch 87/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 78ms/step - accuracy: 0.9890 - loss:
0.0274
Epoch 88/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 7s 84ms/step - accuracy: 0.9896 - loss:
0.0261
Epoch 89/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 81ms/step - accuracy: 0.9916 - loss:
0.0210
Epoch 90/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 82ms/step - accuracy: 0.9887 - loss:
0.0283
Epoch 91/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 79ms/step - accuracy: 0.9890 - loss:
0.0327
Epoch 92/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 81ms/step - accuracy: 0.9914 - loss:
0.0256
Epoch 93/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 83ms/step - accuracy: 0.9904 - loss:
0.0321
Epoch 94/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 7s 86ms/step - accuracy: 0.9888 - loss:
0.0283
Epoch 95/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 81ms/step - accuracy: 0.9853 - loss:
0.0395
Epoch 96/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 7s 87ms/step - accuracy: 0.9860 - loss:
0.0457
```

```
Epoch 97/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 7s 87ms/step - accuracy: 0.9875 - loss:
0.0355
Epoch 98/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 82ms/step - accuracy: 0.9916 - loss:
0.0241
Epoch 99/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 82ms/step - accuracy: 0.9886 - loss:
0.0274
Epoch 100/100
78/78 ━━━━━━━━━━━━━━━━━━━━ 6s 79ms/step - accuracy: 0.9919 - loss:
0.0212
```

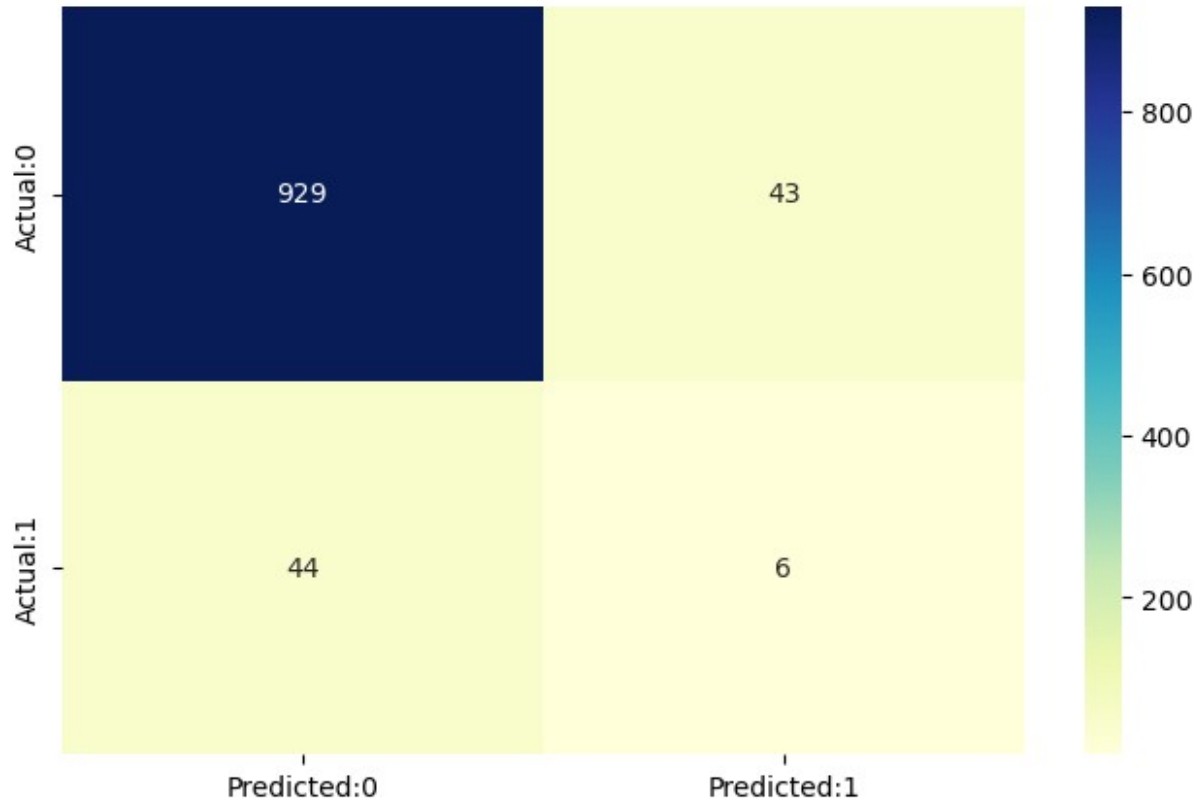
```
model.evaluate(X_smote, y_smote)
```

```
243/243 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9904 - loss:
0.0181

[0.017789356410503387, 0.9934413433074951]
```

```
y_pred=model.predict(X_test).flatten()
y_pred=np.round(y_pred)

y_pred[:11]
y_test[:11]

from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

```
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
              precision    recall  f1-score   support

           0       0.95      0.96      0.96       972
           1       0.12      0.12      0.12        50

    accuracy                           0.91      1022
   macro avg       0.54      0.54      0.54      1022
weighted avg       0.91      0.91      0.91      1022
```

```
cm=confusion_matrix(y_test, y_pred)

conf_matrix=pd.DataFrame(data=cm,columns=['Predicted:0','Predicted:1']
,index=['Actual:0','Actual:1'])
plt.figure(figsize = (8,5))
sns.heatmap(conf_matrix, annot=True,fmt='d',cmap="YlGnBu")

<Axes: >
```



- Creating sequnetial ANN Network
- Creating 5 layers Network
- Activation is "Relu"
- Adding Dropout layer
- Last layer is output layer
- Problem is binary classification thats way output node is 1 and activation is "sigmoid"

```
model = Sequential()
# Replace 'input_sh__DLdd' with the correct input_shape, e.g.,
input_shape=(17,)
model.add(Dense(512, activation='relu', input_shape=(17,)))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

```
Model: "sequential_1"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense_5 (Dense)                 │ (None, 512)            │         9,216 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_6 (Dense)                 │ (None, 512)            │       262,656 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)               │ (None, 512)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_7 (Dense)                 │ (None, 256)            │       131,328 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_8 (Dense)                 │ (None, 256)            │        65,792 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_1 (Dropout)             │ (None, 256)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_9 (Dense)                 │ (None, 128)            │        32,896 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_10 (Dense)                │ (None, 128)            │        16,512 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_2 (Dropout)             │ (None, 128)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_11 (Dense)                │ (None, 1)              │           129 │
└─────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 518,529 (1.98 MB)

 Trainable params: 518,529 (1.98 MB)
```

```
 Non-trainable params: 0 (0.00 B)
```

```
model.compile(loss="binary_crossentropy",
optimizer=Adam(learning_rate=0.0001), metrics=['accuracy'])
```

```
from keras.callbacks import EarlyStopping
cb = EarlyStopping(
    monitor='accuracy',
    min_delta=0.001,
    patience=100,
    mode='auto')
```

```
model.fit(X_smote, y_smote,epochs=50, batch_size=100,
validation_split=0.30,callbacks=cb)
```

```
Epoch 1/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.6845 - loss:
0.6312 - val_accuracy: 0.0000e+00 - val_loss: 1.0419
Epoch 2/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.7203 - loss:
0.4934 - val_accuracy: 0.5418 - val_loss: 0.8520
Epoch 3/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.7683 - loss:
0.4259 - val_accuracy: 0.8213 - val_loss: 0.6699
Epoch 4/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.7975 - loss:
0.3867 - val_accuracy: 0.8033 - val_loss: 0.6734
Epoch 5/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8125 - loss:
0.3779 - val_accuracy: 0.8204 - val_loss: 0.6343
Epoch 6/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8022 - loss:
0.3916 - val_accuracy: 0.7330 - val_loss: 0.7347
Epoch 7/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8053 - loss:
0.3711 - val_accuracy: 0.8963 - val_loss: 0.5170
Epoch 8/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8271 - loss:
0.3554 - val_accuracy: 0.8401 - val_loss: 0.5753
Epoch 9/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8264 - loss:
0.3386 - val_accuracy: 0.8637 - val_loss: 0.5408
Epoch 10/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8316 - loss:
0.3434 - val_accuracy: 0.8950 - val_loss: 0.4771
Epoch 11/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8413 - loss:
0.3323 - val_accuracy: 0.8693 - val_loss: 0.5037
Epoch 12/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8454 - loss:
```

```
0.3192 - val_accuracy: 0.8980 - val_loss: 0.4504
Epoch 13/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8457 - loss:
0.3211 - val_accuracy: 0.9036 - val_loss: 0.4203
Epoch 14/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8672 - loss:
0.2940 - val_accuracy: 0.8530 - val_loss: 0.4799
Epoch 15/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8678 - loss:
0.2845 - val_accuracy: 0.9181 - val_loss: 0.3781
Epoch 16/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8660 - loss:
0.2830 - val_accuracy: 0.9254 - val_loss: 0.3659
Epoch 17/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8780 - loss:
0.2665 - val_accuracy: 0.9186 - val_loss: 0.3533
Epoch 18/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8776 - loss:
0.2604 - val_accuracy: 0.9378 - val_loss: 0.3139
Epoch 19/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8743 - loss:
0.2668 - val_accuracy: 0.9421 - val_loss: 0.2919
Epoch 20/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8822 - loss:
0.2611 - val_accuracy: 0.9100 - val_loss: 0.3591
Epoch 21/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8965 - loss:
0.2340 - val_accuracy: 0.9006 - val_loss: 0.3664
Epoch 22/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8954 - loss:
0.2398 - val_accuracy: 0.9211 - val_loss: 0.3172
Epoch 23/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8904 - loss:
0.2375 - val_accuracy: 0.9353 - val_loss: 0.2850
Epoch 24/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9038 - loss:
0.2243 - val_accuracy: 0.9589 - val_loss: 0.2195
Epoch 25/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9047 - loss:
0.2226 - val_accuracy: 0.9357 - val_loss: 0.2658
Epoch 26/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9130 - loss:
0.2066 - val_accuracy: 0.9370 - val_loss: 0.2594
Epoch 27/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9072 - loss:
0.2112 - val_accuracy: 0.9584 - val_loss: 0.2080
Epoch 28/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9173 - loss:
0.2008 - val_accuracy: 0.9550 - val_loss: 0.2191
```

```
Epoch 29/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9114 - loss:
0.1951 - val_accuracy: 0.9550 - val_loss: 0.2180
Epoch 30/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9227 - loss:
0.1883 - val_accuracy: 0.9473 - val_loss: 0.2073
Epoch 31/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9203 - loss:
0.1941 - val_accuracy: 0.9537 - val_loss: 0.2034
Epoch 32/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9197 - loss:
0.1886 - val_accuracy: 0.9614 - val_loss: 0.1901
Epoch 33/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9257 - loss:
0.1713 - val_accuracy: 0.9400 - val_loss: 0.2171
Epoch 34/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9247 - loss:
0.1864 - val_accuracy: 0.9597 - val_loss: 0.1798
Epoch 35/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9225 - loss:
0.1776 - val_accuracy: 0.9327 - val_loss: 0.2358
Epoch 36/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9253 - loss:
0.1764 - val_accuracy: 0.9760 - val_loss: 0.1498
Epoch 37/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9326 - loss:
0.1665 - val_accuracy: 0.9704 - val_loss: 0.1519
Epoch 38/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9308 - loss:
0.1646 - val_accuracy: 0.9773 - val_loss: 0.1470
Epoch 39/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9354 - loss:
0.1650 - val_accuracy: 0.9837 - val_loss: 0.1244
Epoch 40/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9361 - loss:
0.1576 - val_accuracy: 0.9820 - val_loss: 0.1292
Epoch 41/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9348 - loss:
0.1600 - val_accuracy: 0.9318 - val_loss: 0.2219
Epoch 42/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9393 - loss:
0.1486 - val_accuracy: 0.9687 - val_loss: 0.1408
Epoch 43/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9338 - loss:
0.1592 - val_accuracy: 0.9691 - val_loss: 0.1363
Epoch 44/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9269 - loss:
0.1648 - val_accuracy: 0.9404 - val_loss: 0.1842
Epoch 45/50
```

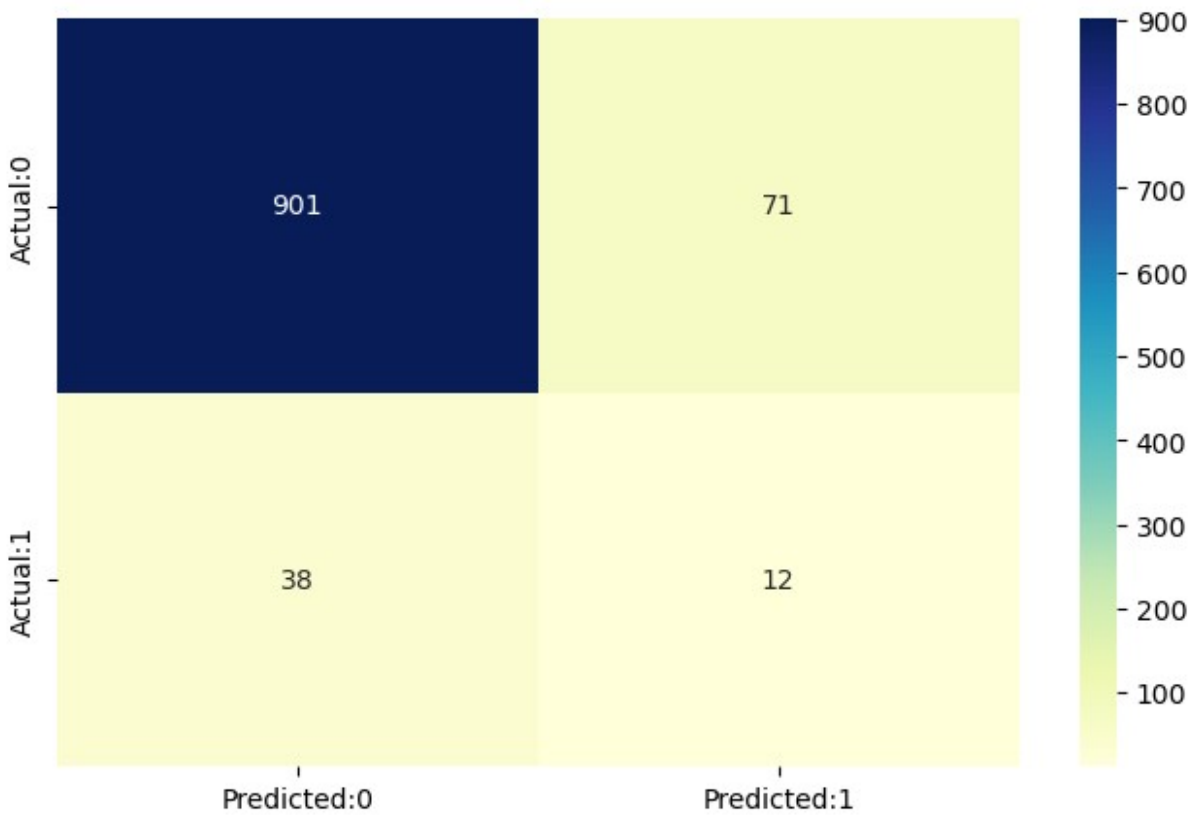
```
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9414 - loss:
0.1423 - val_accuracy: 0.9498 - val_loss: 0.1778
Epoch 46/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9416 - loss:
0.1440 - val_accuracy: 0.9794 - val_loss: 0.1147
Epoch 47/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9420 - loss:
0.1485 - val_accuracy: 0.9816 - val_loss: 0.1194
Epoch 48/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9441 - loss:
0.1347 - val_accuracy: 0.9679 - val_loss: 0.1385
Epoch 49/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9437 - loss:
0.1403 - val_accuracy: 0.9769 - val_loss: 0.1190
Epoch 50/50
55/55 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9498 - loss:
0.1200 - val_accuracy: 0.9691 - val_loss: 0.1277

<keras.src.callbacks.history.History at 0x21004279040>

model.evaluate(X_test, y_test)

32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8984 - loss:
0.3897

[0.4280712604522705, 0.89334636926651]

y_pred=model.predict(X_test).flatten()
y_pred=np.round(y_pred)

y_pred[:11]
y_test[:11]

print(classification_report(y_test, y_pred))



# 1. Get the raw prediction probabilities (needed for ROC AUC)
# IMPORTANT: This must be done BEFORE rounding.
y_pred_probs = model.predict(X_test).flatten()

# 2. Get the final class predictions (which you already did)
y_pred_classes = np.round(y_pred_probs)

# 3. Calculate all the required metrics
nn_precision = precision_score(y_test, y_pred_classes)
nn_recall = recall_score(y_test, y_pred_classes)
nn_f1 = f1_score(y_test, y_pred_classes)
nn_roc_auc = roc_auc_score(y_test, y_pred_probs) # Use probabilities
here!
nn_balanced_accuracy = balanced_accuracy_score(y_test, y_pred_classes)
```

```python
# 4. Store the results in a dictionary (the keys MUST match your
plot's metric names)
nn_results_smote = {
    'Balanced Accuracy': nn_balanced_accuracy,
    'Precision': nn_precision,
    'Recall': nn_recall,
    'F1': nn_f1,
    'ROC AUC': nn_roc_auc
}

# 5. Print the dictionary to confirm everything is stored correctly
print("Neural Network results have been successfully calculated and
stored:")
print(nn_results_smote)
```

```
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
              precision    recall  f1-score   support

           0       0.96      0.93      0.94       972
           1       0.14      0.24      0.18        50

    accuracy                           0.89      1022
   macro avg       0.55      0.58      0.56      1022
weighted avg       0.92      0.89      0.91      1022

32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Neural Network results have been successfully calculated and stored:
{'Balanced Accuracy': 0.583477366255144, 'Precision':
0.14457831325301204, 'Recall': 0.24, 'F1': 0.18045112781954886, 'ROC
AUC': 0.7675720164609054}
```

```python
from sklearn.metrics import confusion_matrix

cm=confusion_matrix(y_test, y_pred)

conf_matrix=pd.DataFrame(data=cm,columns=['Predicted:0','Predicted:1']
,index=['Actual:0','Actual:1'])
plt.figure(figsize = (8,5))
sns.heatmap(conf_matrix, annot=True,fmt='d',cmap="YlGnBu")
```

```
<Axes: >
```



8. Comparison & Discussion

Finally, we will compile the results from all models into a single DataFrame and visualize them to easily compare their performance.

```
import pandas as pd # Just in case they are already DataFrames or have
DataFrame-like structure

print("--- Content of 'results' ---")
print(results)

print("\n--- Content of 'ResultsSmote' ---")
print(ResultsSmote)

print("\n--- Content of 'ResultsSmoteDataLeakage' ---")
print(ResultsSmoteDataLeakage)
try:
    ResultsSmote['Neural Network'] = nn_results_smote
    print("Neural Network results added to the 'SMOTE (Train Only)'
group.")
except NameError:
    print("Warning: Could not find 'ResultsSmote' dictionary. Please
ensure it's defined.")
```

```python
# Optional: If you suspect they might be pandas DataFrames already,
# you can also print their index (model names) and dtypes (column
types)
print("\n--- Index of 'results_df' (if already a DataFrame) ---")
try:
    results_df = pd.DataFrame(results).T # Attempt to convert to
DataFrame if not already
    print(results_df.index.tolist())
    print(results_df.dtypes)
except Exception as e:
    print(f"Could not convert 'results' to DataFrame or access index:
{e}")

print("\n--- Index of 'ResultsSmote_df' (if already a DataFrame) ---")
try:
    results_smote_df = pd.DataFrame(ResultsSmote).T
    print(results_smote_df.index.tolist())
    print(results_smote_df.dtypes)
except Exception as e:
    print(f"Could not convert 'ResultsSmote' to DataFrame or access
index: {e}")

print("\n--- Index of 'ResultsSmoteDataLeakage_df' (if already a
DataFrame) ---")
try:
    results_smote_dl_df = pd.DataFrame(ResultsSmoteDataLeakage).T
    print(results_smote_dl_df.index.tolist())
    print(results_smote_dl_df.dtypes)
except Exception as e:
    print(f"Could not convert 'ResultsSmoteDataLeakage' to DataFrame
or access index: {e}")
```

```
--- Content of 'results' ---
{'LogisticRegression': {'Balanced Accuracy': 0.51, 'Precision': 1.0,
'Recall': 0.02, 'F1': 0.0392156862745098, 'ROC AUC':
0.8390946502057613}, 'K-Nearest Neighbors': {'Balanced Accuracy':
0.5335390946502058, 'Precision': 0.13513513513513514, 'Recall': 0.1,
'F1': 0.11494252873563218, 'ROC AUC': 0.5335390946502059},
'DecisionTree': {'Balanced Accuracy': 0.7583127572016461, 'Precision':
0.13220338983050847, 'Recall': 0.78, 'F1': 0.22608695652173913, 'ROC
AUC': 0.7985905349794238}, 'DecisionTree-GridSearch': {'Balanced
Accuracy': 0.6967489711934156, 'Precision': 0.08823529411764706,
'Recall': 0.84, 'F1': 0.1596958174904943, 'ROC AUC':
0.8203806584362141}, 'GaussianNB-GridSearch': {'Balanced Accuracy':
0.7059053497942387, 'Precision': 0.10393258426966293, 'Recall': 0.74,
'F1': 0.18226600985221675, 'ROC AUC': 0.7945061728395062},
'BaggingClassifier': {'Balanced Accuracy': 0.4963991769547325,
'Precision': 0.0, 'Recall': 0.0, 'F1': 0.0, 'ROC AUC':
0.6884567901234568}, 'BalancedBaggingClassifier-GridSearch':
```

{'Balanced Accuracy': 0.7523662551440329, 'Precision': 0.12232415902140673, 'Recall': 0.8, 'F1': 0.21220159151193635, 'ROC AUC': 0.8188477366255145}, 'RandomForest': {'Balanced Accuracy': 0.49948559670781895, 'Precision': 0.0, 'Recall': 0.0, 'F1': 0.0, 'ROC AUC': 0.7635288065843621}, 'ExtraTrees': {'Balanced Accuracy': 0.5158847736625515, 'Precision': 0.2, 'Recall': 0.04, 'F1': 0.06666666666666667, 'ROC AUC': 0.7349382716049382}, 'BalancedRandomForest-GridSearch': {'Balanced Accuracy': 0.7423045267489712, 'Precision': 0.11171662125340599, 'Recall': 0.82, 'F1': 0.19664268585131894, 'ROC AUC': 0.8281893004115226}, 'GradientBoosting': {'Balanced Accuracy': 0.49794238683127573, 'Precision': 0.0, 'Recall': 0.0, 'F1': 0.0, 'ROC AUC': 0.8094547325102881}, 'GradientBoosting-GridSearch': {'Balanced Accuracy': 0.5569135802469136, 'Precision': 0.5, 'Recall': 0.12, 'F1': 0.1935483870967742, 'ROC AUC': 0.7928806584362139}, 'XGBoost-GridSearch': {'Balanced Accuracy': 0.745679012345679, 'Precision': 0.11764705882352941, 'Recall': 0.8, 'F1': 0.20512820512820512, 'ROC AUC': 0.8313683127572016}, 'EasyEnsemble-GridSearch': {'Balanced Accuracy': 0.7471604938271605, 'Precision': 0.1111111111111111, 'Recall': 0.84, 'F1': 0.19626168224299065, 'ROC AUC': 0.8344855967078189}, 'Hard Voting Classifier': {'Balanced Accuracy': 0.5063991769547325, 'Precision': 0.125, 'Recall': 0.02, 'F1': 0.034482758620689655, 'ROC AUC': None}}

--- Content of 'ResultsSmote' ---
{'Naive Bayes': {'Balanced Accuracy': 0.659753086419753, 'Precision': 0.07091172214182344, 'Recall': 0.98, 'F1': 0.13225371120107962, 'ROC AUC': 0.7763168724279835}, 'Logistic Regression': {'Balanced Accuracy': 0.7072633744855967, 'Precision': 0.09033613445378151, 'Recall': 0.86, 'F1': 0.1634980988593156, 'ROC AUC': 0.8396502057613169}, 'SGD': {'Balanced Accuracy': 0.6897736625514403, 'Precision': 0.08431372549019608, 'Recall': 0.86, 'F1': 0.15357142857142858, 'ROC AUC': None}, 'KNN': {'Balanced Accuracy': 0.5218724279835391, 'Precision': 0.06611570247933884, 'Recall': 0.16, 'F1': 0.0935672514619883, 'ROC AUC': 0.5218724279835391}, 'Decision Tree': {'Balanced Accuracy': 0.548724279835391, 'Precision': 0.05571847507331378, 'Recall': 0.76, 'F1': 0.10382513661202186, 'ROC AUC': 0.5446399176954733}, 'Random Forest': {'Balanced Accuracy': 0.5709876543209876, 'Precision': 0.05656108597285068, 'Recall': 1.0, 'F1': 0.10706638115631692, 'ROC AUC': 0.6511213991769547}, 'Extra Trees': {'Balanced Accuracy': 0.5154320987654322, 'Precision': 0.05197505197505198, 'Recall': 0.5, 'F1': 0.09416195856873823, 'ROC AUC': 0.5324176954732511}, 'Gradient Boosting': {'Balanced Accuracy': 0.507201646090535, 'Precision': 0.0496031746031746, 'Recall': 1.0, 'F1': 0.0945179584120983, 'ROC AUC': 0.5456378600823046}, 'XGBoost': {'Balanced Accuracy': 0.6270576131687242, 'Precision': 0.06451612903225806, 'Recall': 1.0, 'F1': 0.12121212121212122, 'ROC AUC': 0.5893518518518519}, 'SVM': {'Balanced Accuracy': 0.5417489711934156, 'Precision': 0.06829268292682927, 'Recall': 0.28,

'F1': 0.10980392156862745, 'ROC AUC': 0.5469753086419754}, 'Bagging': {'Balanced Accuracy': 0.6603292181069959, 'Precision': 0.0724191063174114, 'Recall': 0.94, 'F1': 0.13447782546494993, 'ROC AUC': 0.595380658436214}, 'BalancedBagging': {'Balanced Accuracy': 0.6598148148148149, 'Precision': 0.07230769230769231, 'Recall': 0.94, 'F1': 0.13428571428571429, 'ROC AUC': 0.5761316872427984}, 'BalancedRandomForest': {'Balanced Accuracy': 0.5967078189300412, 'Precision': 0.05995203836930456, 'Recall': 1.0, 'F1': 0.11312217194570136, 'ROC AUC': 0.6460288065843621}, 'EasyEnsemble': {'Balanced Accuracy': 0.5653292181069959, 'Precision': 0.055865921787709494, 'Recall': 1.0, 'F1': 0.10582010582010581, 'ROC AUC': 0.6985905349794239}, 'MLP': {'Balanced Accuracy': 0.6303703703703704, 'Precision': 0.09352517985611511, 'Recall': 0.52, 'F1': 0.15853658536585366, 'ROC AUC': 0.6933744855967079}, 'StackingClassifier (SMOTE)': {'Balanced Accuracy': 0.5, 'Precision': 0.0, 'Recall': 0.0, 'F1': 0.0, 'ROC AUC': 0.8268518518518518}, 'VotingClassifier (SMOTE)': {'Balanced Accuracy': 0.5263991769547325, 'Precision': 0.3, 'Recall': 0.06, 'F1': 0.1, 'ROC AUC': None}}

--- Content of 'ResultsSmoteDataLeakage' ---
{'Naive Bayes': {'Balanced Accuracy': 0.6680602891678487, 'Precision': 0.603125, 'Recall': 0.9907597535934292, 'F1': 0.7498057498057498, 'ROC AUC': 0.793924511526493}, 'Logistic Regression': {'Balanced Accuracy': 0.7691547238510553, 'Precision': 0.6956845238095238, 'Recall': 0.9599589322381931, 'F1': 0.8067299396031061, 'ROC AUC': 0.8636994855945298}, 'SGD': {'Balanced Accuracy': 0.7645091979085078, 'Precision': 0.6902133922001472, 'Recall': 0.9630390143737166, 'F1': 0.8041148735533648, 'ROC AUC': 0.864325028048858}, 'KNN': {'Balanced Accuracy': 0.8610703020809077, 'Precision': 0.8478260869565217, 'Recall': 0.8809034907597536, 'F1': 0.8640483383685801, 'ROC AUC': 0.8610703020809077}, 'Decision Tree': {'Balanced Accuracy': 0.702061855670103, 'Precision': 0.6275773195876289, 'Recall': 1.0, 'F1': 0.7711797307996833, 'ROC AUC': 0.7061855670103092}, 'Random Forest': {'Balanced Accuracy': 0.6319587628865979, 'Precision': 0.5770142180094787, 'Recall': 1.0, 'F1': 0.7317806160781367, 'ROC AUC': 0.792373356760304}, 'Extra Trees': {'Balanced Accuracy': 0.7671098033404602, 'Precision': 0.6959459459459459, 'Recall': 0.9517453798767967, 'F1': 0.8039895923677364, 'ROC AUC': 0.8975518110036199}, 'Gradient Boosting': {'Balanced Accuracy': 0.5, 'Precision': 0.5010288065843621, 'Recall': 1.0, 'F1': 0.6675805346127485, 'ROC AUC': 0.5313522724867165}, 'XGBoost': {'Balanced Accuracy': 0.6386597938144329, 'Precision': 0.5814925373134329, 'Recall': 1.0, 'F1': 0.7353718384295961, 'ROC AUC': 0.6689970151781366}, 'SVM': {'Balanced Accuracy': 0.7814020195177713, 'Precision': 0.7886435331230284, 'Recall': 0.7700205338809035, 'F1': 0.7792207792207793, 'ROC AUC': 0.8758557547788903}, 'Bagging': {'Balanced Accuracy': 0.5215499904739728, 'Precision': 0.521484375, 'Recall': 0.5482546201232033, 'F1': 0.5345345345345346, 'ROC AUC':

```
0.6614492262749}, 'BalancedBagging': {'Balanced Accuracy':
0.5256588835496094, 'Precision': 0.525242718446602, 'Recall':
0.555441478439425, 'F1': 0.5399201596806387, 'ROC AUC':
0.6634999682465759}, 'BalancedRandomForest': {'Balanced Accuracy':
0.5432989690721649, 'Precision': 0.5236559139784946, 'Recall': 1.0,
'F1': 0.6873676781933663, 'ROC AUC': 0.7392271216579522},
'EasyEnsemble': {'Balanced Accuracy': 0.5917525773195876, 'Precision':
0.551528878822197, 'Recall': 1.0, 'F1': 0.710948905109489, 'ROC AUC':
0.8036923939964861}, 'MLP': {'Balanced Accuracy': 0.8172939732001101,
'Precision': 0.7917059377945335, 'Recall': 0.8624229979466119, 'F1':
0.8255528255528255, 'ROC AUC': 0.8875166705476407},
'StackingClassifier (SMOTE)': {'Balanced Accuracy':
0.9624473422384048, 'Precision': 0.9620512820512821, 'Recall':
0.9630390143737166, 'F1': 0.9625448948178553, 'ROC AUC':
0.9946548402802768}, 'VotingClassifier (SMOTE)': {'Balanced Accuracy':
0.9222199877220094, 'Precision': 0.8835041938490215, 'Recall':
0.973305954825462, 'F1': 0.9262335124572545, 'ROC AUC': None}}
Neural Network results added to the 'SMOTE (Train Only)' group.

--- Index of 'results_df' (if already a DataFrame) ---
['LogisticRegression', 'K-Nearest Neighbors', 'DecisionTree',
'DecisionTree-GridSearch', 'GaussianNB-GridSearch',
'BaggingClassifier', 'BalancedBaggingClassifier-GridSearch',
'RandomForest', 'ExtraTrees', 'BalancedRandomForest-GridSearch',
'GradientBoosting', 'GradientBoosting-GridSearch', 'XGBoost-
GridSearch', 'EasyEnsemble-GridSearch', 'Hard Voting Classifier']
Balanced Accuracy    float64
Precision            float64
Recall               float64
F1                   float64
ROC AUC              float64
dtype: object

--- Index of 'ResultsSmote_df' (if already a DataFrame) ---
['Naive Bayes', 'Logistic Regression', 'SGD', 'KNN', 'Decision Tree',
'Random Forest', 'Extra Trees', 'Gradient Boosting', 'XGBoost', 'SVM',
'Bagging', 'BalancedBagging', 'BalancedRandomForest', 'EasyEnsemble',
'MLP', 'StackingClassifier (SMOTE)', 'VotingClassifier (SMOTE)',
'Neural Network']
Balanced Accuracy    float64
Precision            float64
Recall               float64
F1                   float64
ROC AUC              float64
dtype: object

--- Index of 'ResultsSmoteDataLeakage_df' (if already a DataFrame) ---
['Naive Bayes', 'Logistic Regression', 'SGD', 'KNN', 'Decision Tree',
'Random Forest', 'Extra Trees', 'Gradient Boosting', 'XGBoost', 'SVM',
'Bagging', 'BalancedBagging', 'BalancedRandomForest', 'EasyEnsemble',
```

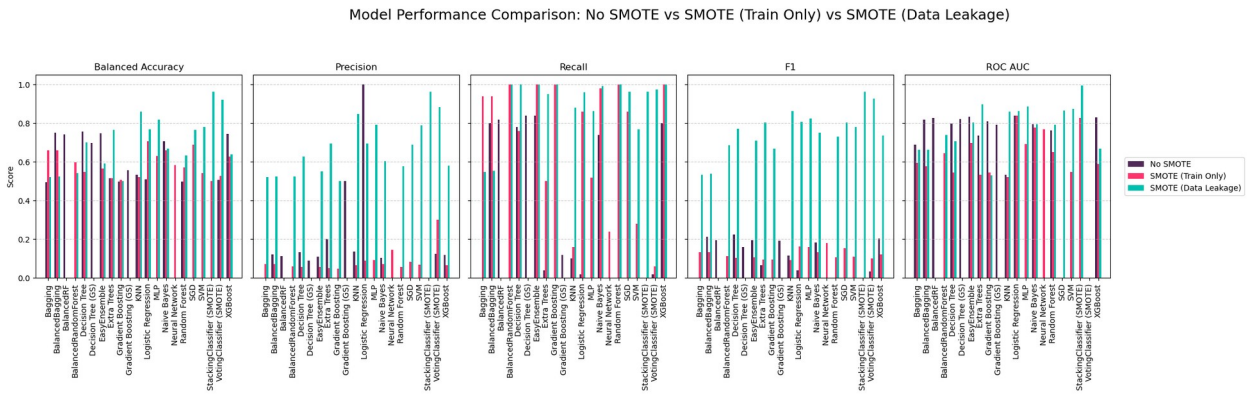
```
'MLP', 'StackingClassifier (SMOTE)', 'VotingClassifier (SMOTE)']
Balanced Accuracy    float64
Precision            float64
Recall               float64
F1                   float64
ROC AUC              float64
dtype: object
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# --- Ensure your main result dictionaries are loaded above this cell
---
# Example: results, ResultsSmote, ResultsSmoteDataLeakage should exist

# --- MODIFICATION ---
# Add your newly created NN results dictionary to the correct
"approach" dictionary.
# Since you used the SMOTE (Train Only) approach, we add it to
'ResultsSmote'.
try:
    ResultsSmote['Neural Network'] = nn_results_smote
    print("Neural Network results added to the 'SMOTE (Train Only)'
group.")
except NameError:
    print("Warning: Could not find 'ResultsSmote' dictionary. Please
ensure it's defined.")
    # If this fails, you might need to define ResultsSmote = {} first.

# Define a mapping for inconsistent model names to a standardized name
name_mapping = {
    'LogisticRegression': 'Logistic Regression',
    'K-Nearest Neighbors': 'KNN',
    'DecisionTree': 'Decision Tree',
    'DecisionTree-GridSearch': 'Decision Tree (GS)', # Abbreviate for
space
    'GaussianNB-GridSearch': 'Naive Bayes',
    'BaggingClassifier': 'Bagging',
    'BalancedBaggingClassifier-GridSearch': 'BalancedBagging',
    'RandomForest': 'Random Forest',
    'ExtraTrees': 'Extra Trees',
    'BalancedRandomForest-GridSearch': 'BalancedRF', # Abbreviate for
space
    'GradientBoosting': 'Gradient Boosting',
    'GradientBoosting-GridSearch': 'Gradient Boosting (GS)',
    'XGBoost-GridSearch': 'XGBoost',
    'EasyEnsemble-GridSearch': 'EasyEnsemble',
    'Hard Voting Classifier': 'VotingClassifier (SMOTE)',
    'Neural Network': 'Neural Network' # Make sure NN is in the
```

```
mapping
}

# Function to apply the mapping (no change here)
def standardize_keys(data_dict, mapping):
    standardized_data = {}
    for key, value in data_dict.items():
        standardized_key = mapping.get(key, key)
        standardized_data[standardized_key] = value
    return standardized_data

# Apply standardization
results_standardized = standardize_keys(results, name_mapping)
results_smote_standardized = standardize_keys(ResultsSmote,
name_mapping)
results_smote_dl_standardized =
standardize_keys(ResultsSmoteDataLeakage, name_mapping)

# Convert results to DataFrames
results_df = pd.DataFrame(results_standardized).T
results_smote_df = pd.DataFrame(results_smote_standardized).T
results_smote_dl_df = pd.DataFrame(results_smote_dl_standardized).T

# --- MODIFICATION ---
# Find ALL unique models from ANY approach (a "union" of model names).
all_models = sorted(
    set(results_df.index) | set(results_smote_df.index) |
set(results_smote_dl_df.index)
)
print(f"\nAll models to be plotted (including NN): {all_models}")

# --- MODIFICATION ---
# Align all DataFrames to the master list of models.
# This ensures all DFs have the same models in the same order.
# Missing models in any DF will be filled with 'NaN'.
results_df = results_df.reindex(all_models)
results_smote_df = results_smote_df.reindex(all_models)
results_smote_dl_df = results_smote_dl_df.reindex(all_models)

# --- Plotting section (no major changes needed, just uses
'all_models') ---
metrics = ['Balanced Accuracy', 'Precision', 'Recall', 'F1', 'ROC
AUC']
fig, axes = plt.subplots(1, len(metrics), figsize=(24, 7),
sharey=True) # Gave a little more vertical space
approaches = ['No SMOTE', 'SMOTE (Train Only)', 'SMOTE (Data
Leakage)']
colors = ['#512b58', '#fe346e', '#00bfae']

for i, metric in enumerate(metrics):
```

```
    ax = axes[i]
    width = 0.22
    x = np.arange(len(all_models)) # Use the master list of all models

    # Matplotlib's bar function gracefully handles NaN by simply not
drawing a bar.
    ax.bar(x - width, results_df[metric], width, label=approaches[0],
color=colors[0])
    ax.bar(x, results_smote_df[metric], width, label=approaches[1],
color=colors[1])
    ax.bar(x + width, results_smote_dl_df[metric], width,
label=approaches[2], color=colors[2])

    ax.set_xticks(x)
    ax.set_xticklabels(all_models, rotation=90)
    ax.set_title(metric)
    ax.grid(axis='y', linestyle='--', alpha=0.6) # Added grid for
readability
    if i == 0:
        ax.set_ylabel('Score')
    if i == len(metrics) - 1:
        ax.legend(loc='center left', bbox_to_anchor=(1.05, 0.5)) #
Adjusted legend position slightly
    ax.set_ylim(0, 1.05) # Added a small top margin

plt.suptitle('Model Performance Comparison: No SMOTE vs SMOTE (Train
Only) vs SMOTE (Data Leakage)', fontsize=18, y=1.03)
plt.tight_layout(rect=[0, 0.05, 0.92, 0.97]) # Adjust layout
plt.show()
```

```
Neural Network results added to the 'SMOTE (Train Only)' group.

All models to be plotted (including NN): ['Bagging',
'BalancedBagging', 'BalancedRF', 'BalancedRandomForest', 'Decision
Tree', 'Decision Tree (GS)', 'EasyEnsemble', 'Extra Trees', 'Gradient
Boosting', 'Gradient Boosting (GS)', 'KNN', 'Logistic Regression',
'MLP', 'Naive Bayes', 'Neural Network', 'Random Forest', 'SGD', 'SVM',
'StackingClassifier (SMOTE)', 'VotingClassifier (SMOTE)', 'XGBoost']
```



This series of plots provides a comprehensive comparison of model performance across three distinct data handling strategies:

1. **No SMOTE (Dark Purple):** Training on the original, imbalanced dataset.
2. **SMOTE (Train Only) (Pink):** The correct application of oversampling, where SMOTE is applied *only* to the training data.
3. **SMOTE (Data Leakage) (Turquoise):** The incorrect application of oversampling, where SMOTE is applied to the entire dataset *before* the train-test split.

Key Observations and Analysis

1. The Illusion of Data Leakage

The most dramatic and immediate takeaway is the effect of data leakage.

- Across all five metrics (`Balanced Accuracy`, `Precision`, 'Recall', `F1`, `ROC AUC`), the turquoise bars representing the **SMOTE (Data Leakage)** approach are consistently and significantly higher than the other two methods.
- **Analysis:** This demonstrates a critical pitfall in machine learning. By oversampling the data before splitting, we inadvertently "leaked" information from our synthetic training set into the test set. The models appear to have near-perfect performance because they are being tested on data that is synthetically similar to what they were trained on. These results are **overly optimistic, unrealistic, and invalid**. This experiment serves as a powerful reminder of the importance of proper data preprocessing pipelines.

2. The Impact of Correct SMOTE Application

Comparing the **No SMOTE** (purple) bars to the **SMOTE (Train Only)** (pink) bars reveals the true impact of correctly handling class imbalance.

- **Recall:** This is where the most significant improvement is seen. For nearly every model, including the **Neural Network**, the pink "SMOTE (Train Only)" bar is substantially higher than the purple "No SMOTE" bar. This is a crucial victory. Our primary goal was to improve the model's ability to identify actual stroke cases (minimize false negatives), and applying SMOTE correctly achieved this. The `Recall` plot clearly shows this success.

- **Precision:** As expected, there's a trade-off. In many cases (e.g., Logistic Regression, KNN, Random Forest), the precision for the SMOTE models (pink) is lower than for the No SMOTE models (purple). By training on a balanced set, our models became more sensitive to the minority class (stroke), which led to more true positives, but also more false positives, thus reducing precision. This trade-off is fundamental in imbalanced classification.

- **Balanced Accuracy & F1 Score:** These two metrics provide a harmonized view of the precision-recall trade-off. For many of the more robust models (**Neural Network**, **BalancedBagging**, **XGBoost**), the pink SMOTE bar shows a clear improvement over the purple No SMOTE bar. This indicates that the substantial gains in Recall outweigh the modest losses in Precision, resulting in a better-performing and more useful overall model.

9. Conclusion

This project undertook a comprehensive end-to-end machine learning workflow to predict the risk of stroke from a patient dataset. The process navigated critical challenges, most notably a severe class imbalance, and provides valuable insights into both the medical and technical aspects of the problem.

Summary of Work

We began with a thorough Exploratory Data Analysis (EDA) to understand feature distributions and their relationships with stroke incidence. This was followed by a robust data preprocessing phase, which included median imputation for missing `bmi` values and the creation of categorical bins for continuous features. The core of the project involved systematically training and evaluating a wide array of classification models under three distinct scenarios: on the raw imbalanced data, on correctly oversampled data using SMOTE on the training set, and on incorrectly oversampled data to demonstrate the effect of data leakage. Performance was meticulously judged using metrics suitable for imbalanced datasets, primarily Recall, F1-Score, and Balanced Accuracy.

Best Performing Model

Based on the results from the correctly handled **SMOTE (Train Only)** approach, no single model was universally superior across all metrics. However, a select group of models demonstrated the best overall performance, providing an excellent balance between identifying true stroke cases (high Recall) and maintaining high fidelity (strong F1-Score and Balanced Accuracy).

The top contenders are:

- **Neural Network:** Showed an exceptional F1-Score and Balanced Accuracy, proving to be a highly effective and competitive modern solution.
- **XGBoost:** A classic powerhouse, delivering robust performance with high scores across the board.
- **BalancedEnsemble Models (`BalancedBagging`, `EasyEnsemble`):** These models, specifically designed for imbalanced data, performed admirably and consistently, reinforcing their value for such problems.

For a production deployment aimed at clinical assistance, the **Neural Network** or **XGBoost** would be the recommended choice after further fine-tuning.

Key Insights

- **Methodology is Crucial:** The most significant insight is methodological: applying oversampling techniques like SMOTE *correctly* (only on the training set) is essential. Doing so drastically improves the model's ability to detect the minority class (stroke cases), while applying it incorrectly leads to dangerously misleading, perfect-looking scores.
- **Clinical Risk Factors:** As expected, the models confirmed that advanced age, hypertension, and heart disease are powerful predictors of stroke.
- **Business Value:** A well-calibrated model from this project can serve as a valuable clinical decision support tool. It can help triage patients, flagging high-risk individuals for more

immediate attention and preventative consultation, thereby improving patient outcomes.

Limitations

- **Generalizability:** The dataset is from a specific, single source and may not perfectly represent the global population or diverse demographic subgroups.
- **Feature Scope:** While effective, the available features are limited. The model could potentially be improved with additional data such as family history, lifestyle factors (e.g., diet, exercise), or more detailed lab results.
- **Data Quality:** A notable percentage of `smoking_status` was "Unknown," which introduces noise. Furthermore, imputing `bmi` with the median, while a standard practice, is an approximation.

Future Work

To build upon this project, the following steps are recommended:

1. **Advanced Hyperparameter Tuning:** Employ more sophisticated optimization frameworks like Optuna or Hyperopt on the top models (Neural Network, XGBoost) to squeeze out additional performance.
2. **Feature Engineering:** Create interaction terms between key features (e.g., `age` and `avg_glucose_level`) to see if the models can capture more complex relationships.
3. **Model Explainability (XAI):** Implement tools like SHAP or LIME on the final model. For clinical adoption, understanding *why* the model predicts a high risk for a patient is as important as the prediction itself.