National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Sudhir Clinton Manjunath
Student ID: x20247818

School of Computing
National College of Ireland

Supervisor: Vladimir Milosavljevic

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Sudhir Clinton Manjunath		
Student ID:	x20247818		
Programme:	M.Sc. Data Analytics	**Year:**	2022
Module:	Research Project		
Supervisor:	Vladimir Milosavljevic		
Submission Due Date:	1st February 2023		
Project Title:	Predicting Stroke at Adulthood Using Machine Learning Techniques		
Word Count:	1555	**Page Count:**	22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Sudhir Clinton Manjunath

Date: 31st January 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Sudhir Clinton Manjunath
x20247818

1 Introduction

In order to set up the ICT solution on any suitable system and achieve the desired results, the user can utilize this configuration manual, which includes a complete walkthrough of the codes for data analysis, preprocessing, and implementing the machine learning models for stroke prediction. The necessary hardware and software specifications are also included in this document.

2 System Configuration

The overall requirements of hardware and software used for this project is discussed in this section.

2.1 Hardware

Device specifications
Nitro AN515-57
Device name LAPTOP-VKUAC4LI
Processor 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz
Installed RAM 8.00 GB (7.77 GB usable)

Windows specifications
Edition Windows 10 Home Single Language
Version 21H2
Installed on 13-12-2021
OS build 19044.2251
Experience Windows Feature Experience Pack 120.2212.4180.0

Figure 1: Hardware Specification

A Windows 10 laptop with a 11th generation intel core i5-processor along with 8 GB RAM is used as shown in figure 1.

2.2 Software

2.2.1 Anaconda

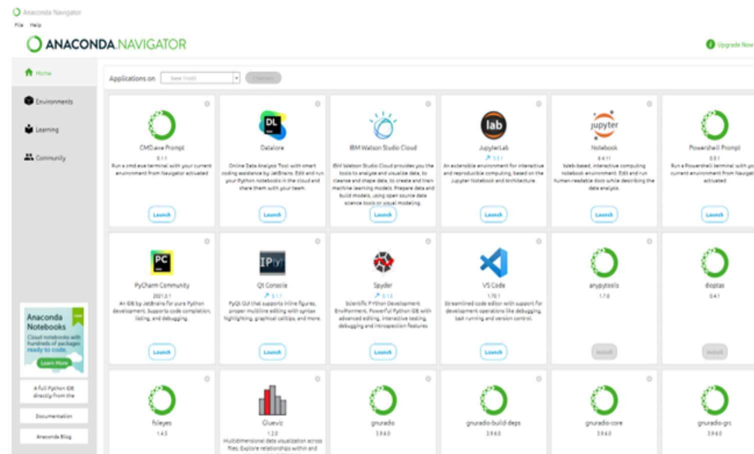


Figure 2: Anaconda Navigator Window

It is a distribution of the Python and R scientific computing programming languages with the goal of streamlining package management and deployment and figure 2 displays the anaconda navigator window. Python is the selected programming language due to its large variety of libraries used for analyzing the data.

2.2.2 Jupyter Notebook

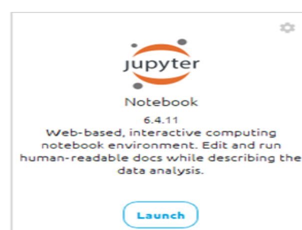


Figure 3: Jupyter Notebook launch icon

It is used as an integrated development editor (IDE) for the deployment of machine learning models since it enables interactive browser-based code execution. The launch icon for the Jupyter notebook which is visible in the anaconda navigator is depicted in figure 3.

3 Python Libraries

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
import category_encoders as ce
from matplotlib import colors as c
from matplotlib import gridspec as grid
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV
from sklearn.metrics import confusion_matrix, classification_report, roc_auc_score, roc_curve
from sklearn import linear_model, metrics
from sklearn.preprocessing import LabelEncoder, StandardScaler
from imblearn.over_sampling import SMOTE
import warnings
warnings.filterwarnings('ignore')
import missingno as msno
```

Figure 4: Necessary Libraries

Figure 4 shows the necessary libraries imported using the keyword import, the other libraries are imported later when required.

4 Data Selection/Reading the file

```
# reading the data
DF = pd.read_csv('stroke_dataset.csv')
```

Figure 5: Reading the data file

The read_csv() is a function in pandas used for importing the dataset [1] which is in the comma separated values (CSV) file format into python and DF is the name of the data frame used to store the data as shown in figure 5.

5 Age Filtering

```
# filtering the age based on target groups
df = DF[DF['AgeCategory'].between('25', '64', inclusive='both')]
```

Figure 6: Filtering the target age groups

The .between() function is used to select a specific age range from the data, inclusive='both' means that both the numbers specified must be included in that age range and is stored in a data frame named df as shown in figure 6.

[1] https://www.kaggle.com/datasets/kamilpytlak/personal-key-indicators-of-heart-disease

5.1 Checking for Duplicate Records

```
1  print(f'Nubmer of duplicated recrods = {df.duplicated().sum()}')
```

```
Nubmer of duplicated recrods = 11169
```

```
1  df = df.drop_duplicates()
```

```
1  print(f'Nubmer of duplicated recrods = {df.duplicated().sum()}')
```

```
Nubmer of duplicated recrods = 0
```

Figure 7: Checking and removing duplicated records

Figure 7 displays the code for showing the duplicate records and removing them along with the output. The df.duplicated().sum() function displays the total number of duplicate records and df.drop_duplicates() function is used to remove them.

6 Exploratory Data Analysis

6.1 Stroke Proportion

```
# pie chart for stroke and no stroke proportion
plt.figure(figsize=(8,5))
labels = "No Stroke", "Stroke"

df['Stroke'].value_counts().plot.pie( autopct='%1.1f%%', startangle = 90,  shadow=True, explode=[0.1,0.1], labels=labels,
                                      colors=['#B7C3F3','salmon'])
```

Figure 8: Code for stroke proportion pie chart

The df['Stroke'].value_counts().plot.pie() function in figure 8 is used for the pie chart that represents the proportion of people who have suffered from stroke and people who have not suffered stroke.

6.2 Univariate Analysis of Numerical Features

The sns.distplot() function in figure 9 is used for plotting the histograms of the numerical features present in the data.

```
# histograms of numerical features
fig,ax = plt.subplots(2,2, figsize=(15,12))
sns.distplot(df['PhysicalHealth'], kde=False, bins=25, hist=True, ax=ax[0][0])
sns.distplot(df['MentalHealth'], kde=False, bins=25, hist=True, ax=ax[0][1])
sns.distplot(df['BMI'], kde=False, bins=25, hist=True, ax=ax[1][0])
sns.distplot(df['SleepTime'], kde=False, bins=25, hist=True, ax=ax[1][1])
plt.show()
```

Figure 9: Code for histograms of numerical features

6.3 Bivariate Analysis of Categorical Features

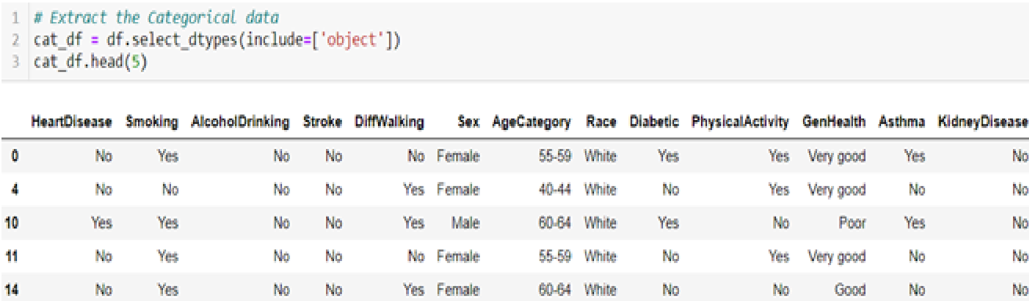
```
# barplots of some of the categorical features with respect to stroke
def percentage_above_bar_relative_to_xgroup(ax):
    all_heights = [[p.get_height() for p in bars] for bars in ax.containers]
    for bars in ax.containers:
        for i, p in enumerate(bars):
            total = sum(xgroup[i] for xgroup in all_heights)
            percentage = f'{(100 * p.get_height() / total) :.1f}%'
            ax.annotate(percentage, (p.get_x() + p.get_width() / 2, p.get_height()), size=17, ha='center', va='bottom')

cat1=df[['HeartDisease', 'Smoking', 'DiffWalking', 'Diabetic']]

fig=plt.figure(figsize=(28,35))
background_color = '#f6f5f7'
fig.patch.set_facecolor(background_color)
for indx,val in enumerate(cat1.columns):
    ax=plt.subplot(4,2,indx+1)
    ax.set_facecolor(background_color)
    ax.set_title(val,fontweight='bold',fontfamily='serif', fontsize= 22)
    for i in ['top','right']:
        ax.spines[i].set_visible(False)
    ax.grid(linestyle=':',axis='y')
    sns.countplot(cat1[val],palette='OrRd_r',hue=df['Stroke'])
    plt.yticks(fontsize=18)
    plt.xticks(fontsize=15)
    plt.setp(ax.get_legend().get_texts(), fontsize='20') # for legend text
    plt.setp(ax.get_legend().get_title(), fontsize='20') # for legend title
    percentage_above_bar_relative_to_xgroup(ax)
```

Figure 10: Code for barplots of numerical features with respect to stroke

The sns.countplot() function in figure 10 is used to plot the bar graphs with respect to stroke. The percentage_above_bar_relative_to_xgroup() function is manually defined so that the percentage is visible on top of each bars.

6.4 Categorical Features

Figure 11 shows the code and output for displaying the categorical features in the data frame.



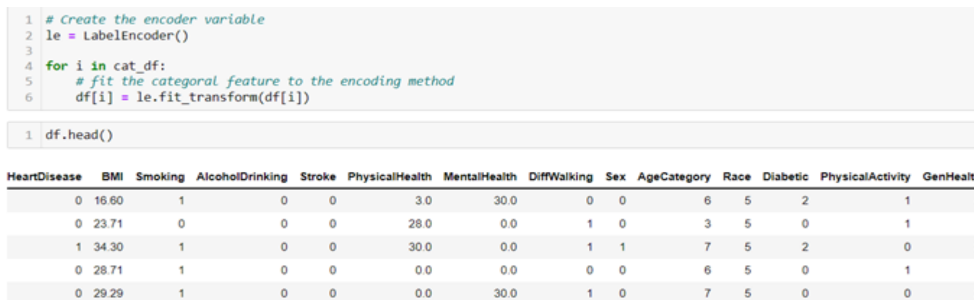
```
1 # Extract the Categorical data
2 cat_df = df.select_dtypes(include=['object'])
3 cat_df.head(5)
```

	HeartDisease	Smoking	AlcoholDrinking	Stroke	DiffWalking	Sex	AgeCategory	Race	Diabetic	PhysicalActivity	GenHealth	Asthma	KidneyDisease
0	No	Yes	No	No	No	Female	55-59	White	Yes	Yes	Very good	Yes	No
4	No	No	No	No	Yes	Female	40-44	White	No	Yes	Very good	No	No
10	Yes	Yes	No	No	Yes	Male	60-64	White	Yes	No	Poor	Yes	No
11	No	Yes	No	No	No	Female	55-59	White	No	Yes	Very good	No	No
14	No	Yes	No	No	Yes	Female	60-64	White	No	No	Good	No	No

Figure 11: Categorical features in the data frame

7 Data Preprocessing and Transformation

7.1 Coverting the Categorical Features to Numeric



```
1 # Create the encoder variable
2 le = LabelEncoder()
3
4 for i in cat_df:
5     # fit the categoral feature to the encoding method
6     df[i] = le.fit_transform(df[i])
```

```
1 df.head()
```

HeartDisease	BMI	Smoking	AlcoholDrinking	Stroke	PhysicalHealth	MentalHealth	DiffWalking	Sex	AgeCategory	Race	Diabetic	PhysicalActivity	GenHealth
0	16.60	1	0	0	3.0	30.0	0	0	6	5	2	1	4
0	23.71	0	0	0	28.0	0.0	1	0	3	5	0	1	4
1	34.30	1	0	0	30.0	0.0	1	1	7	5	2	0	3
0	28.71	1	0	0	0.0	0.0	0	0	6	5	0	1	4
0	29.29	1	0	0	0.0	30.0	1	0	7	5	0	0	2

Figure 12: Code for Coverting the Categorical Features to Numeric

The LabelEncoder() function in figure 12 is used for converting the categorical features into numeric, so it becomes easy for the machine learning algorithms to process it. The output of the encoded data is also shown in the figure.

7.2 Outliers

```
# Outlier detection
df.plot(kind="box", subplots=True, layout=(6,4), figsize=(13,12))
```

Figure 13: Code for outlier detection

Figure 13 shows the code for detecting and displaying the outliers present in the data.

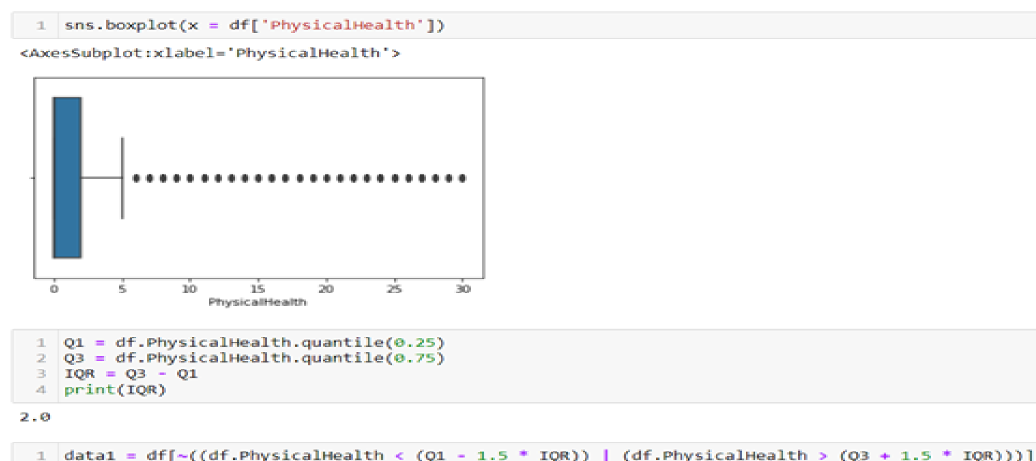


Figure 14: Removing outliers from PhysicalHealth

In figure 14, outliers of feature PhysicalHealth is detected and removed using the inter quartile range (IQR) and stored in a data frame named data1. The same process is followed for other numerical features such as MentalHealth, SleepTime, and BMI, and are stored in data2, data3, and data4 respectively.

7.3 Assigning the target variable

```
# Assigning target variable
x = data4.drop('Stroke',axis=1)
y = data4['Stroke']
```

Figure 15: Assigning the target variable

In figure 15, the target variable is assigned to y and the predictor variables are assigned to x.

7.4 Standardizing the data with numeric features

The StandardScaler() function in figure 16 is used to convert the numerical features with diverse range into the same scale of the categorical features. This function transforms the numerical features so that their mean is set to zero and the

```
1 # To convert numerical attributes with diverse ranges into the same scale of categorical attributes,
2 # feature scaling is implemented.
3 scl = StandardScaler()
4 x[['BMI', 'PhysicalHealth', 'MentalHealth', 'SleepTime']] = scl.fit_transform(x[['BMI', 'PhysicalHealth',
5                                                                                  'MentalHealth', 'SleepTime']])
```

```
1 x.head()
```

BMI	Smoking	AlcoholDrinking	PhysicalHealth	MentalHealth	DiffWalking	Sex	AgeCategory	Race	Diabetic	PhysicalActivity	GenHealth	SleepTime
0.091140	1	0	-0.391931	-0.531639	0	0	6	5	0	1	4	-1.786193
0.176730	0	0	0.516676	-0.531639	0	0	5	5	0	1	4	-0.894768
0.628356	1	0	4.151103	-0.531639	0	1	7	5	0	1	4	-1.786193
-0.604509	0	0	0.516676	-0.531639	0	0	6	5	0	1	4	-0.003344
1.524323	0	0	-0.391931	-0.531639	0	0	7	5	2	0	2	-1.786193

Figure 16: Feature scaling

resulting distribution has a unit standard deviation, and this can be seen in the figure as well. After this, again the target variable and the predictor variables are reassigned to y and x respectively as shown in figure 15.

7.5 Feature Selection

```
from sklearn import metrics
import statsmodels.api as sm

lr1 = sm.GLM(y, (sm.add_constant(x)), family = sm.families.Binomial())
lr1.fit().summary()
```

Figure 17: Code for Generalized Linear Model

The sm.GLM() function in figure 17 is used for building a generalized linear model (GLM) with the predictor variables and the family is set to binomial. This indicates the GLM that the target variable is binary in nature.

In backward elimination technique, the important features are selected by eliminating the feature with a higher p-value. Figure 18 shows the summary of the GLM, in which the feature BMI has the highest p-value 0.729. Hence, BMI is removed first.

Generalized Linear Model Regression Results

Dep. Variable:	Stroke	No. Observations:	130095
Model:	GLM	Df Residuals:	130077
Model Family:	Binomial	Df Model:	17
Link Function:	logit	Scale:	1.0000
Method:	IRLS	Log-Likelihood:	-8992.7
Date:	Mon, 12 Dec 2022	Deviance:	17985.
Time:	23:46:51	Pearson chi2:	1.19e+05
No. Iterations:	8		
Covariance Type:	nonrobust		

	coef	std err	z	P>|z|	[0.025	0.975]
const	-5.6532	0.214	-26.361	0.000	-6.074	-5.233
HeartDisease	1.5956	0.060	26.393	0.000	1.477	1.714
BMI	0.0015	0.004	0.346	0.729	-0.007	0.010
Smoking	0.4269	0.048	8.931	0.000	0.333	0.521
AlcoholDrinking	-0.2569	0.098	-2.609	0.009	-0.450	-0.064
PhysicalHealth	0.0497	0.018	2.795	0.005	0.015	0.084
MentalHealth	0.0123	0.009	1.323	0.186	-0.006	0.030
DiffWalking	1.1048	0.063	17.411	0.000	0.980	1.229
Sex	0.0556	0.048	1.161	0.246	-0.038	0.150
AgeCategory	0.2307	0.013	17.511	0.000	0.205	0.257
Race	-0.1010	0.017	-5.981	0.000	-0.134	-0.068
Diabetic	0.2092	0.029	7.253	0.000	0.153	0.266
PhysicalActivity	-0.1155	0.056	-2.066	0.039	-0.225	-0.006
GenHealth	-0.0272	0.016	-1.705	0.088	-0.058	0.004
SleepTime	0.0360	0.019	1.862	0.063	-0.002	0.074
Asthma	0.3571	0.062	5.782	0.000	0.236	0.478
KidneyDisease	0.5851	0.105	5.576	0.000	0.379	0.791
SkinCancer	0.1516	0.092	1.645	0.100	-0.029	0.332

Figure 18: Summary of GLM

```
1  col = x.columns
2
3  # Removing BMI
4  col = col.drop('BMI', 1)
5  col
```

```
Index(['HeartDisease', 'Smoking', 'AlcoholDrinking', 'PhysicalHealth',
       'MentalHealth', 'DiffWalking', 'Sex', 'AgeCategory', 'Race', 'Diabetic',
       'PhysicalActivity', 'GenHealth', 'SleepTime', 'Asthma', 'KidneyDisease',
       'SkinCancer'],
      dtype='object')
```

Figure 19: Removing BMI

Figure 19 shows the code to remove BMI from the list and also displays the remaining variables.

```
# re-running the model using the remaining variables
load = sm.add_constant(x[col])
lr2 = sm.GLM(y, load, family = sm.families.Binomial())
res = lr2.fit()
res.summary()
```

Figure 20: Code for re-running the GLM

In figure 20, the GLM is re-run with the remaining variables after removing BMI.

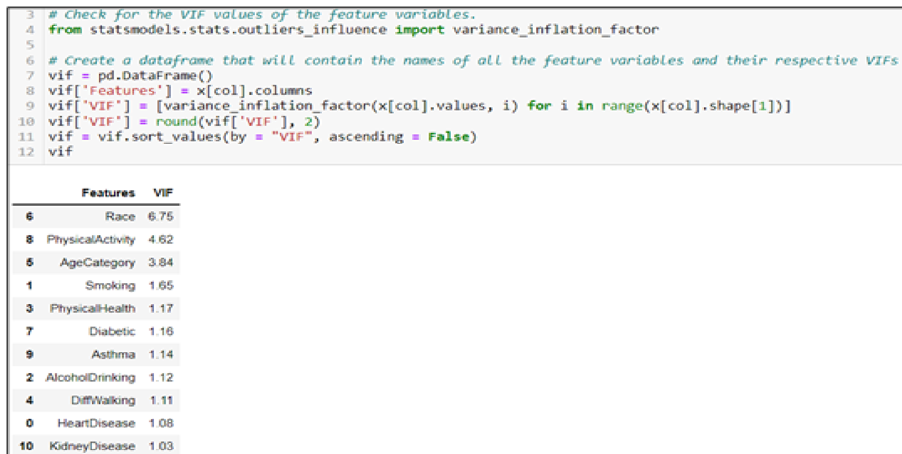
Generalized Linear Model Regression Results

Dep. Variable:	Stroke	No. Observations:	130095
Model:	GLM	Df Residuals:	130078
Model Family:	Binomial	Df Model:	16
Link Function:	logit	Scale:	1.0000
Method:	IRLS	Log-Likelihood:	-8992.8
Date:	Mon, 12 Dec 2022	Deviance:	17986.
Time:	23:46:51	Pearson chi2:	1.19e+05
No. Iterations:	8		
Covariance Type:	nonrobust		

	coef	std err	z	P>|z|	[0.025	0.975]
const	-5.6107	0.176	-31.928	0.000	-5.955	-5.266
HeartDisease	1.5964	0.060	26.421	0.000	1.478	1.715
Smoking	0.4263	0.048	8.923	0.000	0.333	0.520
AlcoholDrinking	-0.2580	0.098	-2.622	0.009	-0.451	-0.065
PhysicalHealth	0.0498	0.018	2.802	0.005	0.015	0.085
MentalHealth	0.0123	0.009	1.323	0.186	-0.006	0.030
DiffWalking	1.1074	0.063	17.573	0.000	0.984	1.231
Sex	0.0567	0.048	1.185	0.236	-0.037	0.150
AgeCategory	0.2307	0.013	17.514	0.000	0.205	0.257
Race	-0.1009	0.017	-5.976	0.000	-0.134	-0.068
Diabetic	0.2110	0.028	7.431	0.000	0.155	0.267
PhysicalActivity	-0.1173	0.056	-2.106	0.035	-0.226	-0.008
GenHealth	-0.0269	0.016	-1.688	0.091	-0.058	0.004
SleepTime	0.0359	0.019	1.855	0.064	-0.002	0.074
Asthma	0.3584	0.062	5.816	0.000	0.238	0.479
KidneyDisease	0.5851	0.105	5.575	0.000	0.379	0.791
SkinCancer	0.1510	0.092	1.639	0.101	-0.030	0.332

Figure 21: Summary of GLM

In figure 21, it can be seen that the feature Sex has the highest p-value of 0.236, hence Sex is removed next. It is removed using the same code as in figure 19, just that Sex is replaced in place of BMI. This process follows until all the features have a p-value less than 0.05.

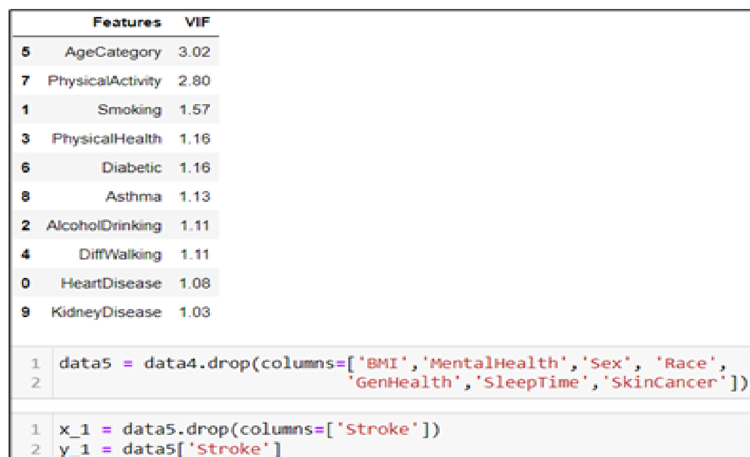


```
3  # Check for the VIF values of the feature variables.
4  from statsmodels.stats.outliers_influence import variance_inflation_factor
5
6  # Create a dataframe that will contain the names of all the feature variables and their respective VIFs
7  vif = pd.DataFrame()
8  vif['Features'] = x[col].columns
9  vif['VIF'] = [variance_inflation_factor(x[col].values, i) for i in range(x[col].shape[1])]
10 vif['VIF'] = round(vif['VIF'], 2)
11 vif = vif.sort_values(by = "VIF", ascending = False)
12 vif
```

	Features	VIF
6	Race	6.75
8	PhysicalActivity	4.62
5	AgeCategory	3.84
1	Smoking	1.65
3	PhysicalHealth	1.17
7	Diabetic	1.16
9	Asthma	1.14
2	AlcoholDrinking	1.12
4	DiffWalking	1.11
0	HeartDisease	1.08
10	KidneyDisease	1.03

Figure 22: Variance inflation factor values

Figure 22 shows the code and output for checking the variance inflation factor (VIF). Generally, features with VIF value less than 5 is considered. In the figure it can be seen that Race has a VIF value of 6.75, hence it is removed. The GLM is re-run, again checked for insignificant variables and then the VIF is checked again.



	Features	VIF
5	AgeCategory	3.02
7	PhysicalActivity	2.80
1	Smoking	1.57
3	PhysicalHealth	1.16
6	Diabetic	1.16
8	Asthma	1.13
2	AlcoholDrinking	1.11
4	DiffWalking	1.11
0	HeartDisease	1.08
9	KidneyDisease	1.03

```
1 data5 = data4.drop(columns=['BMI','MentalHealth','Sex', 'Race',
2                            'GenHealth','SleepTime','SkinCancer'])
```

```
1 x_1 = data5.drop(columns=['Stroke'])
2 y_1 = data5['Stroke']
```

Figure 23: VIF values and removing insignificant features

In figure 23, it can be seen that all the ten features have a VIF value less than 0.05, so these are the only features used for building machine learning algorithms

for the stroke prediction. The codes for removing the insignificant variables and re-assigning the predictor variables is also shown.

7.6 Splitting the data into train and test sets

```
# Train-test Split
# random_state is used for 2 purposes
# for the reproduceability of results
# the dataset must have its data randomly shuffled in order to avoid biases in the data prediction

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x_1, y_1, test_size = 0.2, random_state=100, stratify = y_1)
```

Figure 24: Data split into train and test

The code for splitting the data into 20% for test and 80% for train using stratified random sampling is displayed in figure 24.

8 Experiments

8.1 Random Undersampling (RUS)

```
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler(random_state = 1000)
x_train_res, y_train_res = rus.fit_resample(x_train, y_train)
```

Figure 25: RUS balancing method

The RandomUnderSampler() function in figure 25 is used for undersampling the train data.

8.2 SMOTE

```
from imblearn.over_sampling import SMOTE

sm = SMOTE(random_state = 2000)
x_train_sm, y_train_sm = sm.fit_resample(x_train, y_train)
```

Figure 26: SMOTE balancing method

The SMOTE() function in figure 26 is used for oversampling the train data.

9 Implementation and Evaluation of Models

9.1 Random Forest (RF)

```
from sklearn.ensemble import RandomForestClassifier

rf_rus = RandomForestClassifier(random_state = 32)
rf_rus.fit(x_train_res, y_train_res)
y_pred_rf_rus = rf_rus.predict(x_test)
rf_rus_auc = roc_auc_score(y_test, y_pred_rf_rus)

print(f"AUC Score of Random Forest: {rf_rus_auc}")
print (classification_report(y_test, y_pred_rf_rus))
```

```
AUC Score of Random Forest: 0.6724520406027255
              precision    recall  f1-score   support

           0       0.99      0.69      0.81     25623
           1       0.03      0.65      0.06       396

    accuracy                           0.69     26019
   macro avg       0.51      0.67      0.44     26019
weighted avg       0.98      0.69      0.80     26019
```

Figure 27: Base version of RF

The code and the result for the base version of RF classifier is displayed in figure 27.

```
# Hyperparameter tuning

rf_rus1 = RandomForestClassifier(random_state = 42)

scope_rf_rus1 = {'n_estimators': (10, 30, 50, 70, 90, 100),
                 'criterion': ('gini', 'entropy'),
                 'max_depth': (3, 5, 7, 9, 10),
                 'max_features': ('auto', 'sqrt'),
                 'bootstrap': ('True', 'False'),
                 'min_samples_split': (2, 4, 6)}
```

```
rf_rus_randomgrid = RandomizedSearchCV (estimator = rf_rus1, param_distributions = scope_rf_rus1,
                                        random_state = 42, cv = 10, verbose = 2, n_jobs = 4)

rf_rus_randomgrid.fit(x_train_res, y_train_res)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits

RandomizedSearchCV(cv=10, estimator=RandomForestClassifier(random_state=42),
                   n_jobs=4,
                   param_distributions={'bootstrap': ('True', 'False'),
                                        'criterion': ('gini', 'entropy'),
                                        'max_depth': (3, 5, 7, 9, 10),
                                        'max_features': ('auto', 'sqrt'),
                                        'min_samples_split': (2, 4, 6),
                                        'n_estimators': (10, 30, 50, 70, 90,
                                                         100)},
                   random_state=42, verbose=2)
```

```
rf_rus_randomgrid.best_params_
```

```
{'n_estimators': 70,
 'min_samples_split': 4,
 'max_features': 'sqrt',
 'max_depth': 5,
 'criterion': 'gini',
 'bootstrap': 'True'}
```

Figure 28: RF parameter tuning

The code and the output for finding the best parameters using randomized search along with 10-fold cross validation for RF is displayed in figure 28.

```
1  rf_rus2 = RandomForestClassifier(n_estimators = 70, min_samples_split = 4, max_features = 'sqrt',
2                                   max_depth = 5, criterion = 'gini', bootstrap = 'True', random_state = 40)
3
4  rf_rus2.fit(x_train_res, y_train_res)
5  y_pred_rf_rus2 = rf_rus2.predict(x_test)
6  rf_rus2_auc = roc_auc_score(y_test, y_pred_rf_rus2)
7
8  print(f"AUC Score of Random Forest: {rf_rus2_auc}")
9  print (classification_report(y_test, y_pred_rf_rus2))
```

```
AUC Score of Random Forest: 0.7044518773970829
              precision    recall  f1-score   support

           0       0.99      0.72      0.83     25623
           1       0.04      0.69      0.07       396

    accuracy                           0.72     26019
   macro avg       0.51      0.70      0.45     26019
weighted avg       0.98      0.72      0.82     26019
```

Figure 29: Tuned version of RF

The code and the result for the tuned version of RF classifier is displayed in figure 29. The RF built under SMOTE is similar to this with the same parameters, instead _rus and _res is replaced with _sm, and the random_state is set to 70.

9.2 Bernoulli Naive Bayes (BNB)

```
1  from sklearn.naive_bayes import BernoulliNB
2
3  bnb_rus = BernoulliNB()
4  bnb_rus.fit(x_train_res, y_train_res)
5  y_pred_bnb_rus = bnb_rus.predict(x_test)
6  bnb_rus_auc = roc_auc_score(y_test, y_pred_bnb_rus)
7
8  print(f"AUC Score of BernoulliNB: {bnb_rus_auc}")
9  print (classification_report(y_test, y_pred_bnb_rus))
```

```
AUC Score of BernoulliNB: 0.6791842733623555
              precision    recall  f1-score   support

           0       0.99      0.82      0.89     25623
           1       0.04      0.54      0.08       396

    accuracy                           0.81     26019
   macro avg       0.52      0.68      0.49     26019
weighted avg       0.98      0.81      0.88     26019
```

Figure 30: Base version of BNB

The code and the result for the base version of BNB is displayed in figure 30.

```
1 # Hyperparameter tuning
2
3 bnb_rus1 = BernoulliNB()
4
5 scope_bnb_rus1 = {'alpha': [0.00001 ,0.0001 ,0.001, 0.01, 0.1, 0.5, 1.0, 2.0, 5.0, 10.0, 15.0, 20.0]}
```

```
1 bnb_rus_randomgrid = RandomizedSearchCV (estimator = bnb_rus1, param_distributions = scope_bnb_rus1,
2                                          random_state = 42, cv = 10, verbose = 2, n_jobs = 4)
3
4 bnb_rus_randomgrid.fit(x_train_res, y_train_res)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits

RandomizedSearchCV(cv=10, estimator=BernoulliNB(), n_jobs=4,
                   param_distributions={'alpha': [1e-05, 0.0001, 0.001, 0.01,
                                                  0.1, 0.5, 1.0, 2.0, 5.0, 10.0,
                                                  15.0, 20.0]},
                   random_state=42, verbose=2)
```

```
1 bnb_rus_randomgrid.best_params_
```

```
{'alpha': 15.0}
```

Figure 31: BNB parameter tuning

The code and the output for finding the best parameters using randomized search along with 10-fold cross validation for BNB is displayed in figure 31.

```
1 bnb_rus2 = BernoulliNB(alpha=15.0, fit_prior=True)
2
3 bnb_rus2.fit(x_train_res, y_train_res)
4 y_pred_bnb_rus2 = bnb_rus2.predict(x_test)
5 bnb_rus2_auc = roc_auc_score(y_test, y_pred_bnb_rus2)
6
7 print(f"AUC Score of BernoulliNB: {bnb_rus2_auc}")
8 print (classification_report(y_test, y_pred_bnb_rus2))
```

```
AUC Score of BernoulliNB: 0.6805054407794133
              precision    recall  f1-score   support

           0       0.99      0.82      0.89     25623
           1       0.04      0.55      0.08       396

    accuracy                           0.81     26019
   macro avg       0.52      0.68      0.49     26019
weighted avg       0.98      0.81      0.88     26019
```

Figure 32: Tuned version of BNB

The code and the result for the tuned version of BNB is displayed in figure 32. The BNB built under SMOTE is similar to this, instead _rus and _res is replaced with _sm.

9.3 Stochastic Gradient Descent (SGD)

```
1  from sklearn.linear_model import SGDClassifier
2
3  sgd_rus = SGDClassifier(random_state = 52)
4  sgd_rus.fit(x_train_res, y_train_res)
5  y_pred_sgd_rus = sgd_rus.predict(x_test)
6  sgd_rus_auc = roc_auc_score(y_test, y_pred_sgd_rus)
7
8  print(f"AUC Score of Stochastic Gradient Descent: {sgd_rus_auc}")
9  print (classification_report(y_test, y_pred_sgd_rus))
```

```
AUC Score of Stochastic Gradient Descent: 0.6943380059818416
              precision    recall  f1-score   support

           0       0.99      0.81      0.89     25623
           1       0.04      0.58      0.08       396

    accuracy                           0.80     26019
   macro avg       0.52      0.69      0.49     26019
weighted avg       0.98      0.80      0.88     26019
```

Figure 33: Base version of SGD

The code and the result for the base version of SGD classifier is displayed in figure 33.

```
1  # Hyperparameter tuning
2
3  sgd_rus1 = SGDClassifier()
4
5  scope_sgd_rus1 = {'loss': ('log', 'modified_huber', 'perceptron'),
6                    'penalty': ('l1', 'l2', 'elasticnet'),
7                    'alpha': (0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000),
8                    'learning_rate': ('constant', 'optimal', 'invscaling', 'adaptive'),
9                    'eta0' : (1, 10, 100)}
```

```
1  sgd_rus_randomgrid = RandomizedSearchCV (estimator = sgd_rus1, param_distributions = scope_sgd_rus1,
2                                           random_state = 42, cv = 10, verbose = 2, n_jobs = 4)
3
4  sgd_rus_randomgrid.fit(x_train_res, y_train_res)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits

RandomizedSearchCV(cv=10, estimator=SGDClassifier(), n_jobs=4,
                   param_distributions={'alpha': (0.0001, 0.001, 0.01, 0.1, 1,
                                                  10, 100, 1000),
                                        'eta0': (1, 10, 100),
                                        'learning_rate': ('constant', 'optimal',
                                                          'invscaling',
                                                          'adaptive'),
                                        'loss': ('log', 'modified_huber',
                                                 'perceptron'),
                                        'penalty': ('l1', 'l2', 'elasticnet')},
                   random_state=42, verbose=2)
```

```
1  sgd_rus_randomgrid.best_params_
```

```
{'penalty': 'l2',
 'loss': 'log',
 'learning_rate': 'optimal',
 'eta0': 100,
 'alpha': 0.01}
```

Figure 34: SGD parameter tuning

The code and the output for finding the best parameters using randomized search along with 10-fold cross validation for SGD is displayed in figure 34.

```
1 sgd_rus2 = SGDClassifier(loss = 'log', penalty = 'l2', alpha = 0.01, learning_rate = 'optimal',
2                          eta0 = 100, random_state = 32)
3
4 sgd_rus2.fit(x_train_res, y_train_res)
5 y_pred_sgd_rus2 = sgd_rus2.predict(x_test)
6 sgd_rus2_auc = roc_auc_score(y_test, y_pred_sgd_rus2)
7
8 print(f"AUC Score of Stochastic Gradient Descent: {sgd_rus2_auc}")
9 print (classification_report(y_test, y_pred_sgd_rus2))
```

```
AUC Score of Stochastic Gradient Descent: 0.6955372126605004
              precision    recall  f1-score   support

           0       0.99      0.67      0.80     25623
           1       0.03      0.72      0.06       396

    accuracy                           0.67     26019
   macro avg       0.51      0.70      0.43     26019
weighted avg       0.98      0.67      0.79     26019
```

Figure 35: Tuned version of SGD

The code and the result for the tuned version of SGD classifier is displayed in figure 35. The SGD built under SMOTE is similar to this with the same parameters, instead _rus and _res is replaced with _sm, and the random_state is set to 22 in base version and 50 in tuned version.

9.4 Adaptive Boosting (AB)

```
1 from sklearn.ensemble import AdaBoostClassifier
2
3 ab_rus = AdaBoostClassifier()
4 ab_rus.fit(x_train_res, y_train_res)
5 y_pred_ab_rus = ab_rus.predict(x_test)
6 ab_rus_auc = roc_auc_score(y_test, y_pred_ab_rus)
7
8 print(f"AUC Score of Adaboost: {ab_rus_auc}")
9 print (classification_report(y_test, y_pred_ab_rus))
```

```
AUC Score of Adaboost: 0.7028038552011155
              precision    recall  f1-score   support

           0       0.99      0.75      0.85     25623
           1       0.04      0.66      0.07       396

    accuracy                           0.75     26019
   macro avg       0.52      0.70      0.46     26019
weighted avg       0.98      0.75      0.84     26019
```

Figure 36: Base version of AB

The code and the result for the base version of AB classifier is displayed in figure 36.

```
1  # Hyperparameter tuning
2
3  ab_rus1 = AdaBoostClassifier()
4
5  scope_ab_rus1 = {'n_estimators': (10, 30, 50, 70, 90, 100),
6                   'learning_rate': (0.0001, 0.001, 0.01, 0.1, 1.0, 5.0)}
```

```
1  ab_rus_randomgrid = RandomizedSearchCV (estimator = ab_rus1, param_distributions = scope_ab_rus1,
2                                      random_state = 42, cv = 10, verbose = 2, n_jobs = 4)
3
4  ab_rus_randomgrid.fit(x_train_res, y_train_res)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits

RandomizedSearchCV(cv=10, estimator=AdaBoostClassifier(), n_jobs=4,
                   param_distributions={'learning_rate': (0.0001, 0.001, 0.01,
                                                          0.1, 1.0, 5.0),
                                        'n_estimators': (10, 30, 50, 70, 90,
                                                         100)},
                   random_state=42, verbose=2)
```

```
1  ab_rus_randomgrid.best_params_
```

```
{'n_estimators': 50, 'learning_rate': 1.0}
```

Figure 37: AB parameter tuning

The code and the output for finding the best parameters using randomized search along with 10-fold cross validation for AB is displayed in figure 37.

```
1  ab_rus2 = AdaBoostClassifier(n_estimators = 50, learning_rate = 1.0)
2
3  ab_rus2.fit(x_train_res, y_train_res)
4  y_pred_ab_rus2 = ab_rus2.predict(x_test)
5  ab_rus2_auc = roc_auc_score(y_test, y_pred_ab_rus2)
6
7  print(f"AUC Score of Adaboost: {ab_rus2_auc}")
8  print (classification_report(y_test, y_pred_ab_rus2))
```

```
AUC Score of Adaboost: 0.7028038552011155
              precision    recall  f1-score   support

           0       0.99      0.75      0.85     25623
           1       0.04      0.66      0.07       396

    accuracy                           0.75     26019
   macro avg       0.52      0.70      0.46     26019
weighted avg       0.98      0.75      0.84     26019
```

Figure 38: Tuned version of AB

The code and the result for the tuned version of AB classifier is displayed in figure 38. The AB built under SMOTE is similar to this, instead _rus and _res is replaced with _sm. Additionally, AB is not hyper tuned because under RUS even after hype tuning the results remained the same.

9.5 Logistic Regression (LR)

```
1 from sklearn.linear_model import LogisticRegression
2
3 logreg_rus = LogisticRegression()
4 logreg_rus.fit(x_train_res, y_train_res)
5 y_pred_logreg_rus = logreg_rus.predict(x_test)
6 logreg_rus_auc = roc_auc_score(y_test, y_pred_logreg_rus)
7
8 print(f"AUC Score of Logistic Regression: {logreg_rus_auc}")
9 print (classification_report(y_test, y_pred_logreg_rus))
```

```
AUC Score of Logistic Regression: 0.7047747407336449
              precision    recall  f1-score   support

           0       0.99      0.75      0.85     25623
           1       0.04      0.66      0.07       396

    accuracy                           0.75     26019
   macro avg       0.52      0.70      0.46     26019
weighted avg       0.98      0.75      0.84     26019
```

Figure 39: Base version of LR

The code and the result for the base version of LR is displayed in figure 39.

```
1 # Hyperparameter tuning
2
3 logreg_rus1 = LogisticRegression()
4
5 scope_logreg_rus1 = {'C' : (0.0001, 0.001, 0.01, 0.1, 0.5, 1, 10),
6                      'solver' : ('lbfgs','newton-cg','sag'),
7                      'max_iter' : (10, 30, 50, 70, 90, 110, 130, 150, 170, 190, 210, 230, 250)}
```

```
1 logreg_rus_randomgrid = RandomizedSearchCV (estimator = logreg_rus1, param_distributions = scope_logreg_rus1,
2                                             random_state = 42, cv = 10, verbose = 2, n_jobs = 4)
3
4 logreg_rus_randomgrid.fit(x_train_res, y_train_res)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits

RandomizedSearchCV(cv=10, estimator=LogisticRegression(), n_jobs=4,
                   param_distributions={'C': (0.0001, 0.001, 0.01, 0.1, 0.5, 1,
                                              10),
                                        'max_iter': (10, 30, 50, 70, 90, 110,
                                                     130, 150, 170, 190, 210,
                                                     230, 250),
                                        'solver': ('lbfgs', 'newton-cg',
                                                   'sag')},
                   random_state=42, verbose=2)
```

```
1 logreg_rus_randomgrid.best_params_
```

```
{'solver': 'newton-cg', 'max_iter': 150, 'C': 0.1}
```

Figure 40: LR parameter tuning

The code and the output for finding the best parameters using randomized search along with 10-fold cross validation for LR is displayed in figure 40.

```
1 logreg_rus2 = LogisticRegression(solver = 'newton-cg', max_iter = 150, C = 0.1)
2
3 logreg_rus2.fit(x_train_res, y_train_res)
4 y_pred_logreg_rus2 = logreg_rus2.predict(x_test)
5 logreg_rus2_auc = roc_auc_score(y_test, y_pred_logreg_rus2)
6
7 print(f"AUC Score of Logistic Regression: {logreg_rus2_auc}")
8 print (classification_report(y_test, y_pred_logreg_rus2))
```

```
AUC Score of Logistic Regression: 0.7037037037037037
              precision    recall  f1-score   support

           0       0.99      0.74      0.85     25623
           1       0.04      0.67      0.07       396

    accuracy                           0.74     26019
   macro avg       0.52      0.70      0.46     26019
weighted avg       0.98      0.74      0.84     26019
```

Figure 41: Tuned version of LR

The code and the result for the tuned version of LR is displayed in figure 41. LR built under SMOTE is similar to this with the same parameters, instead _rus and _res is replaced with _sm.

9.6 Support Vector Machine (SVM)

```
1 from sklearn.svm import SVC
2
3 svm_rus = SVC()
4 svm_rus.fit(x_train_res, y_train_res)
5 y_pred_svm_rus = svm_rus.predict(x_test)
6 svm_rus_auc = roc_auc_score(y_test, y_pred_svm_rus)
7
8 print(f"AUC Score of Support Vector Machine: {svm_rus_auc}")
9 print (classification_report(y_test, y_pred_svm_rus))
```

```
AUC Score of Support Vector Machine: 0.6938718942143599
              precision    recall  f1-score   support

           0       0.99      0.69      0.81     25623
           1       0.03      0.70      0.06       396

    accuracy                           0.69     26019
   macro avg       0.51      0.69      0.44     26019
weighted avg       0.98      0.69      0.80     26019
```

Figure 42: Base version of SVM

The code and the result for the base version of SVM is displayed in figure 42.

```
1 # Hyperparameter tuning
2
3 svm_rus1 = SVC()
4
5 scope_svm_rus1 = {'C': (0.01, 0.1, 1, 5, 10, 50, 100, 200), 'gamma': (0.001, 0.01, 0.1, 1, 5, 10)}
```

```
1 svm_rus_randomgrid = RandomizedSearchCV (estimator = svm_rus1, param_distributions = scope_svm_rus1,
2                                          random_state = 42, cv = 10, verbose = 2, n_jobs = 4)
3
4 svm_rus_randomgrid.fit(x_train_res, y_train_res)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits

RandomizedSearchCV(cv=10, estimator=SVC(), n_jobs=4,
                   param_distributions={'C': (0.01, 0.1, 1, 5, 10, 50, 100,
                                              200),
                                        'gamma': (0.001, 0.01, 0.1, 1, 5, 10)},
                   random_state=42, verbose=2)
```

```
1 svm_rus_randomgrid.best_params_
```

```
{'gamma': 0.01, 'C': 100}
```

Figure 43: SVM parameter tuning

The code and the output for finding the best parameters using randomized search along with 10-fold cross validation for SVM is displayed in figure 43.

```
1 svm_rus2 = SVC(C = 100 , gamma = 0.01)
2
3 svm_rus2.fit(x_train_res, y_train_res)
4 y_pred_svm_rus2 = svm_rus2.predict(x_test)
5 svm_rus2_auc = roc_auc_score(y_test, y_pred_svm_rus2)
6
7 print(f"AUC Score of Support Vector Machine: {svm_rus2_auc}")
8 print (classification_report(y_test, y_pred_svm_rus2))
```

```
AUC Score of Support Vector Machine: 0.6994643484369512
              precision    recall  f1-score   support

           0       0.99      0.70      0.82     25623
           1       0.04      0.69      0.07       396

    accuracy                           0.70     26019
   macro avg       0.51      0.70      0.45     26019
weighted avg       0.98      0.70      0.81     26019
```

Figure 44: Tuned version of SVM

The code and the result for the tuned version of SVM is displayed in figure 44. The SVM built under SMOTE is similar to this, instead _rus and _res is replaced with _sm. Additionally, SVM is not hyper tuned because under RUS, after hype tuning, the recall value decreased by 1%.

9.7 Voting Classifier (VC)

```
1  from sklearn.ensemble import VotingClassifier
2
3  vc_rus = VotingClassifier(estimators = [("AdaBoostClassifier", ab_rus), ("SGDClassifier", sgd_rus2),
4                                          ("LogisticRegression", logreg_rus2)], voting = "hard")
5
6  vc_rus.fit(x_train_res, y_train_res)
7  y_pred_vc_rus = vc_rus.predict(x_test)
8  vc_rus_auc = roc_auc_score(y_test, y_pred_vc_rus)
9
10 print(f"AUC Score of Voting Classifier: {vc_rus_auc}")
11 print (classification_report(y_test, y_pred_vc_rus))
```

```
AUC Score of Voting Classifier: 0.7018051076270254
              precision    recall  f1-score   support

           0       0.99      0.73      0.84     25623
           1       0.04      0.67      0.07       396

    accuracy                           0.73     26019
   macro avg       0.52      0.70      0.46     26019
weighted avg       0.98      0.73      0.83     26019
```

Figure 45: Code and result for VC

The code and the result for VC is displayed in figure 45. The VC built under SMOTE is similar to this, instead _rus and _res is replaced with _sm.

9.8 Stacking Classifier (SC)

```
1  from sklearn.ensemble import StackingClassifier
2
3  stack_rus = StackingClassifier(estimators = [('ab_rus', AdaBoostClassifier()),
4                                               ('sgd_rus2', SGDClassifier(random_state = 71))],
5                                                 final_estimator = logreg_rus2)
6
7  stack_rus.fit(x_train_res, y_train_res)
8  y_pred_stack_rus = stack_rus.predict(x_test)
9  stack_rus_auc = roc_auc_score(y_test, y_pred_stack_rus)
10
11 print(f"AUC Score of Stacking Classifier: {stack_rus_auc}")
12 print (classification_report(y_test, y_pred_stack_rus))
```

```
AUC Score of Stacking Classifier: 0.6975963534182712
              precision    recall  f1-score   support

           0       0.99      0.66      0.79     25623
           1       0.03      0.73      0.06       396

    accuracy                           0.66     26019
   macro avg       0.51      0.70      0.43     26019
weighted avg       0.98      0.66      0.78     26019
```

Figure 46: Code and result for SC

The code and the result for SC is displayed in figure 46. The SC built under SMOTE is similar to this, instead _rus and _res is replaced with _sm and the random_state of SGD classifier which is a base learner is set to 50.