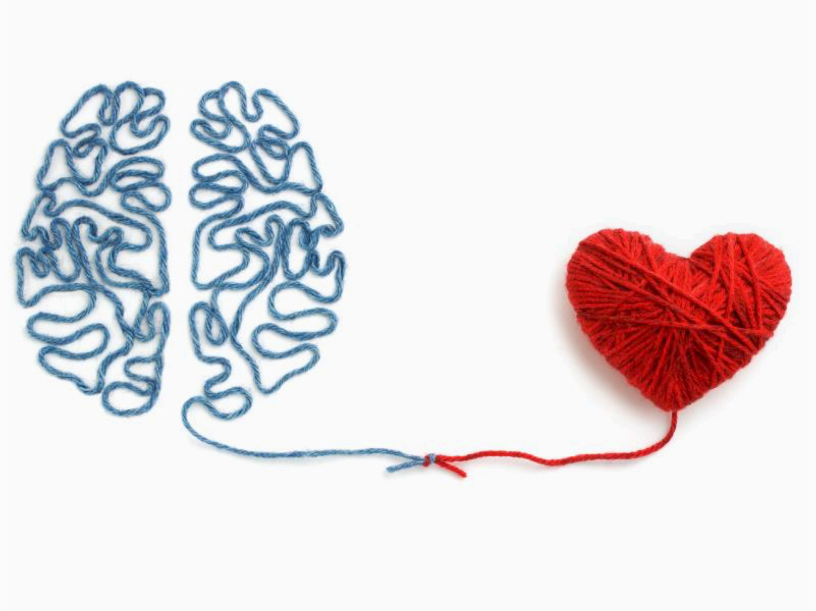
Heart Stroke Data Science Project Report

Table of Contents



1. Executive Summary

A heart stroke, not to be confused with a heart attack, is a brain attack in which vital blood flow and oxygen flow to the brain is abruptly cut. It can happen when a blood vessel feeding the brain bursts or gets clogged. Heart strokes constitute a strain on human lives, as it is the cause of 11% of death worldwide according to the World Health Organization and contributes to the decline of economies. One of the ways to prevent such events and limit their fatal and disastrous impacts is through predicting in advance the probability that a person could suffer from a heart stroke, using, for instance, binary classification machine learning. We were given a dataset consisting of a list of 12 attributes for over 5000 patients, a minority of whom suffered from a stroke, but most of whom did not. In other words, the dataset is unbalanced.

When running statistics on the data, we found that age was one of the best predictors of likelihood to have a stroke, as one would expect, but so is the average glucose level, BMI, and having been married to a smaller extent (indeed this is due to the fact that older people tend to have a higher chance of having been married in their life). Surprisingly heart disease is only weakly correlated, if at all, to stroke, which is due to the fact that heart stroke and heart attacks or heart disease have little to do with each other.

One challenge of this problem, as mentioned earlier, is the fact that the dataset is unbalanced which makes minimizing false positives more difficult. Indeed, all of our predictive models and classifiers used, such as logistic regression, ensemble methods like adaboost or random forests, neural networks, k-nearest-neighbors, support vector machines and Naive Bayes, report relatively high accuracy and recall but poor precision and F1 scores. Our best performing model, the multilayer perceptron classifier, has a recall score of 87%, which means that our machine learning model is good at identifying as many true positives as there are, with a low false negative rate, but a high false positive rate. In the context of medicine and predicting disease in general, a low precision is acceptable in the cases where we care about identifying as many of the true positive cases as possible and treating them, as false positive cases can undergo further physical testing to be determined negative. Thus, the preditive model can be used by hospitals, clinics and health care centers such as UCLA Health on their own patient's data so as to obtain a statistical idea of who's potentially at risk. The model can be further improved by adding more relevant attributes, or finding a more powerful set of parameters for the model.

In increasing order of performance, we trained and tested the following models: random forests with 35 decision trees, adaboost boosting with 25 decision trees, 70-nearest-neighbor, logistic regression with L1 regularization, support vector machine linearly kernelized, neural network with 4 hidden layers of 10 nodes each and a tanh activation function. After implementing principal component analysis, retaining 90% of the variance, and decreasing the dimensionality of the standardized data from 10 to 8, our model performed slightly worse which is to be expected.

2. Background/Introduction

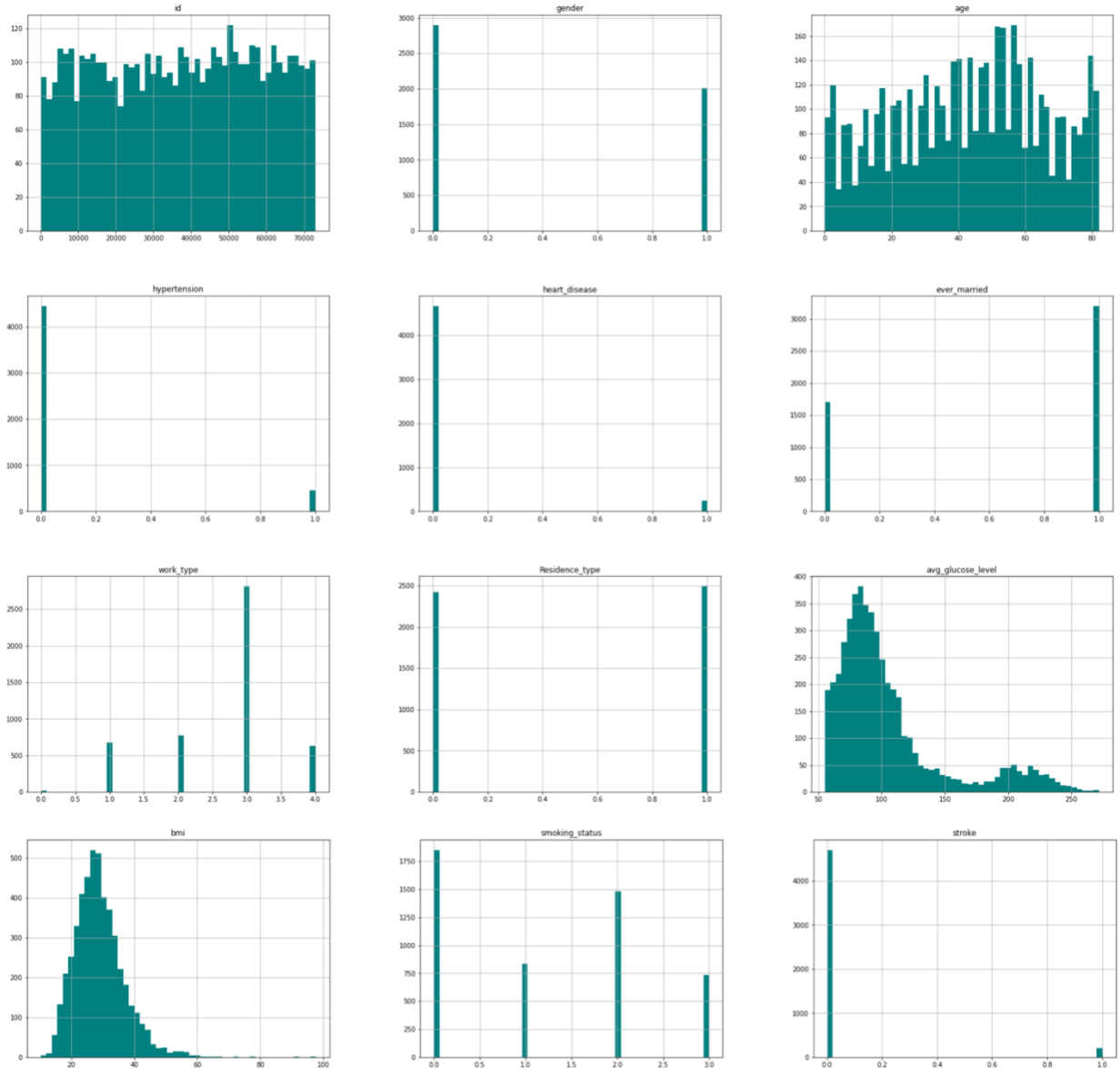
A heart stroke happens when a clot or rupure interrupts blood flow to the brain leading to the death of brain cells, as they lack oxygen-rich blood. In the US each year, about 1 in every 6 cardiovascular disease deaths is due to heart stroke. According to the World Health Organization, stroke is the 2nd leading cause of death globally and makes up a tenth of all deaths. But strokes do not only pose a threat to human life, they also impact the economy negatively tremendously. For example, in 2016 the total cost associated with strokes in the US was $103.5 billion. More than 795,000 people have a stroke every year in the US, and more than 1 in 4 strokes, are from people who already had one. This is why preventing such strokes by determining who's at risk of one based on contributing factors and provide them with the necessary health care is a deeply important matter, both medically and economically. A data scientist is hence fit for this job. We were given a dataset containing various information fields that were collected by hospitals for a list of patients, some of whom were recorded to have had a stroke. Our task in this project is to train efficient predictive models that can help us classify new patients whose information is known into two categories: "likely to have a stroke", and "unlikely to have a stroke". For this dataset, we are given the following fields:

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

3. Methodology

1. Basic Statistics

We first ran some basic statistics on our variables. Indeed, the first steps we took were to load our dataset and inspect it by looking at the distribution of the variables, encoding the categorical variables into a set of integer numerical values so that we could run correlations and plot their distributions in histograms.



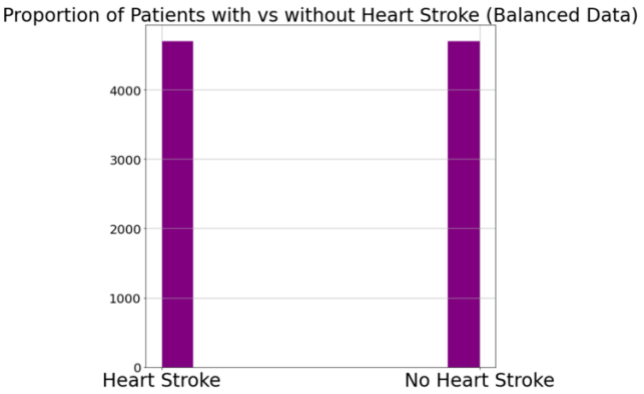
From our data distribution, we see that our dataset is not balanced. Specifically, we have many more patients without a stroke than with a stroke. We will thus need to balance our dataset. There are slightly fewer men than women, BMI is a bell shape distribution centered at an average of 28.8. We also have a lot more people healthy from heart disease than people sick with

it. It may be that heart disease is a great predictor of stroke, just looking at their two histograms side by side.

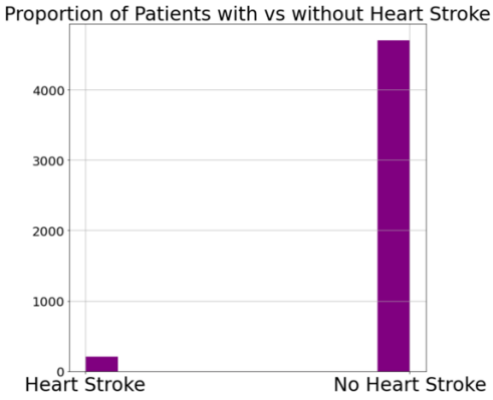
2. Data extraction, augmentation, balancing and pipeline

We then removed datapoints, such as missing values (N/A) from the BMI field, by simply imputing the rows from the dataset. Since we were given that there were 4909 non-null values for BMI out of 5110 data points, we concluded that 201 BMI datapoints were missing, which represented roughly 4% of our data frame. The BMI seems like such an important predictor for heart disease (intuitively) that it would be unwise to extrapolate or assign the median/average BMI on the missing rows. We hence simply removed the 201 rows altogether.

We removed the one and only row containing "Other" for the gender variable, as including it would likely increase the complexity of our model for little if any benefit to its predictive power. This allowed us to simply encode gender as a binary numerical variable.



Balanced Dataset



Unbalanced distribution

We then plotted the correlation matrix of our dataset so as to identify correlations after balancing our data using a SMOTE synthetic oversampling of the minority class.

```
# Balance the dataset
from imblearn.over_sampling import SMOTE

# Resample the minority class
sm = SMOTE(random_state=27, sampling_strategy='minority')
heartstrokeBalanced_X, heartstrokeBalanced_Y = sm.fit_resample(heartstrokeCopy.drop(['stroke'], axis=1), heartstrokeCopy['stroke'])
heartstrokeBalanced = pd.concat([pd.DataFrame(heartstrokeBalanced_X), pd.DataFrame(heartstrokeBalanced_Y)], axis=1)
```

We also augmented variables using feature crosses at the same time so as to make sure our augmented features had strong correlations with the stroke feature. Certain features showing little correlation could then be dropped as well, like ID, hypertension, or heart_disease.

See the next result section for a discussion of the correlations found and the features we augmented.

After our data extraction plan was finalized, we implemented our pipeline which consisted in normalizing our numerical features using the StandardScaler() so as to improve our model's

predictive abilities. It is indeed important to standardize the data so that the varying sizes of the different ranges of the features don't end up having any importance in the model (each feature is treated with the same weight to begin with in relation to the others, so there needs to be a common scale between them).

Our pipeline also takes care of categorical features (work_type and smoking_status). We one-hot encode work_type but not smoking_status as smoking_status contains a cardinal relationship that we discovered while inspecting the correlation matrix (see the results section of this report). Gender, ever_married and Residence_type can be either one-hot encoded or left as numerical features since they can only take one of two values (0 or 1), thus we don't need to one-hot encode them and can leave them as integers (0 or 1) and pass them into the numerical pipeline.

Finally, once our data pipeline is ready, we are able to perform an 80/20 test/train split (in order to train and test the subsequent models we will make use of) and then balance our prepared data. Note that we only balance the training set so as to not introduce errors in our performance metrics. We train on balanced data so that our models aren't biased towards predicting no strokes, but we then need to test on the unbalanced data so as to obtain performance metrics on a representative sample (on a balanced data set we would expect our model to perform a lor better due to the distribution of strokes being even).

3. Logistic Regression

Our next steps will be to train predictive models that can classify patients into "likely to have a stroke" and "unlikely to have a stroke." We start with a logistic regression model. For each model that we study, we plot the corresponding confusion matrix which is a breakdown of the true positives, true negatives, false positives, false negatives predicted by the model, the four main metrics (accuracy, precision, recall and F1 score) and the corresponding ROC curve which is a measure of our model's ability to separate classes (specifically whether that ability is better or worse than it would be if it were just due to chance, and by how much). We use L1 regularization (so as to force the features which are the least important to zero) and the liblinear solver.

4. Principal Component Analysis

Next, we implemented principal component analysis so as to reduce the complexity or dimentionality of our problem and see if it improves or not our logistic classification model's performance. We implement PCA directly on standardized data (from our prepared pipeline), and choose the number of components necessary to retain 90% of the variance (so as to lose as little information as possible). We currently have 10 features, after principal component analysis, it turns out that we are left with 8 principal components. After performing one more train/test split on the new data and balancing again the training cohort, we can compare how this dataset performs under logisitic regression. See the results section.

5. Ensemble Methods

We now employ two ensemble methods to solve the classification problem. We decided to train a Boosting algorithm (Adaboost) and a random forest algorithm. For both, we set max_depth to 1, that is, we set the depth of each decision tree in our ensemble to 1 (we made this choice because we know that decreasing the depth of each decision tree reduces variance and increases bias). n_estimators is the number of trees used: the more trees the more we overfit. After tweaking the parameters, we found satisfying performance metrics for Adaboost at n_estimators=25, with a learning rate of 3. For random forests, we used 35 trees and max_features=0.5, which means that 50% of the features are used (sampled randomly) for each new tree in the random forest. The results are discussed in the next section.

6. Neural Network Classifier

We can now move on to our neural network multilayer perceptron classifier that we implemented using sklearn.neural_network's MLPClassifier.
After trying out a combination of parameters, we found that our data performs best on a multilayer perceptron classifier with the following parameters:

- Adaptive learning rate which means that the model keeps the learning rate constant to our chosen value of 0.1 as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5
- Activation function is tanh
- There are 4 hidden layers made up each of 10 nodes
- The solver is "adam", which is a stochastic gradient-based optimizer, with exponential decay rates for first and second moment vectors of 0.1 and 0.2 respectively
- We're using a batch size of 100, meaning we use 100 training samples at each iteration of model training
- max_iter=500: We are running the solver for a maximum of 500 epochs to reach convergence
- We use an L2 regularization penalty term of 0.08
- We are using early_stopping to terminate training if the validation score is not improving

We noted that very slight changes of the parameters could lead to vastly different outcomes in terms of model performance in a way that is almost impossible to predict before running the parameters. Again, the results are discussed in the results section.

7. Cross-Validation

We used 10-fold cross validation on the random forest algorithm and neural network model so as to compare their relative performances, and displayed the mean accuracy, precision, recall and F1 scores so as to determine which was our best model. We made use of the function KFold so as to split our data into 10 folds, and cross_val_score so as to determine the mean scores.

8. Experimenting with my Own Models

I experimented with the nearest neighbor classifier (for different values of k through iteration and printing and comparing the resulting performance metrics for each model), the support vector machine classifier (with vairous different kernels available: linear, radial, sigmoid, polynomial (degree 2,3,4) again comparing the performance metrics for each) and finally the Naive Bayes classifier. I compared all of the resulting performance metrics, performed 10-fold cross-validation again on my two top-performing models and reported my highest-performing model.

4. Results

1. Basic Statistics

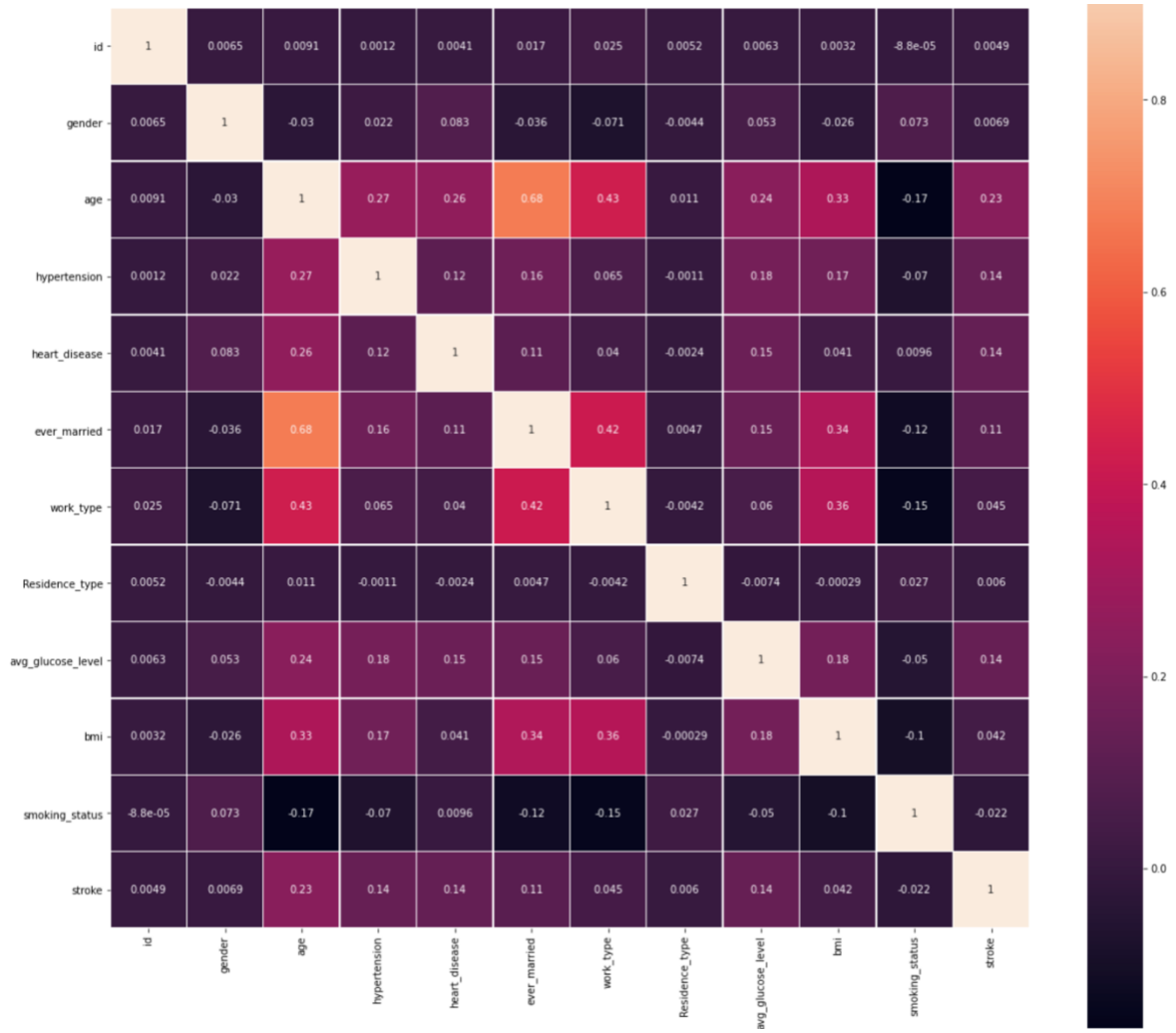
By inspecting our data, we came up with a strategy for encoding the categorical data so we could see correlations and implement an efficient pipeline. The main ideas are summarized here.

```
heartstroke.info() # show the types of the targets
```

```
<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
```

As we can see, we have 3 float fields, 4 integer fields and 5 categorical values. We should be able to convert some of these categorical value fields into numerical fields so as to operate on them. For example, some are booleans like "ever_married" which can be encoded as 1 for "Yes", 0 for "No". "Residence_type" can be encoded as 0="Rural", 1="Urban". Other categoricals will require more specific encoding. We can use an integer encoding, with 0="never smoked", 1="formerly smoked", 2="Unknown", 3="smokes" since we can assume a reasonable cardinality in the data (for example a linear relationship between the amount one smokes and the probability to get a heart stroke). The work_type seems like a good data point to one-hot encode since there is no obvious cardinal relationship between the different values it can take ("children", "Govt_jov", "Never_worked", "Private" or "Self-employed"). Gender is another categorical which we might want to convert into 0 or 1 (male=1, female=0), and "Other" can be simply dropped as only one row contains this value (as shown below), thus our model will perform better if we exclude it and encode gender as a simple boolean. ID likely serves no purpose when it comes to predicting and thus will be imputed completely.

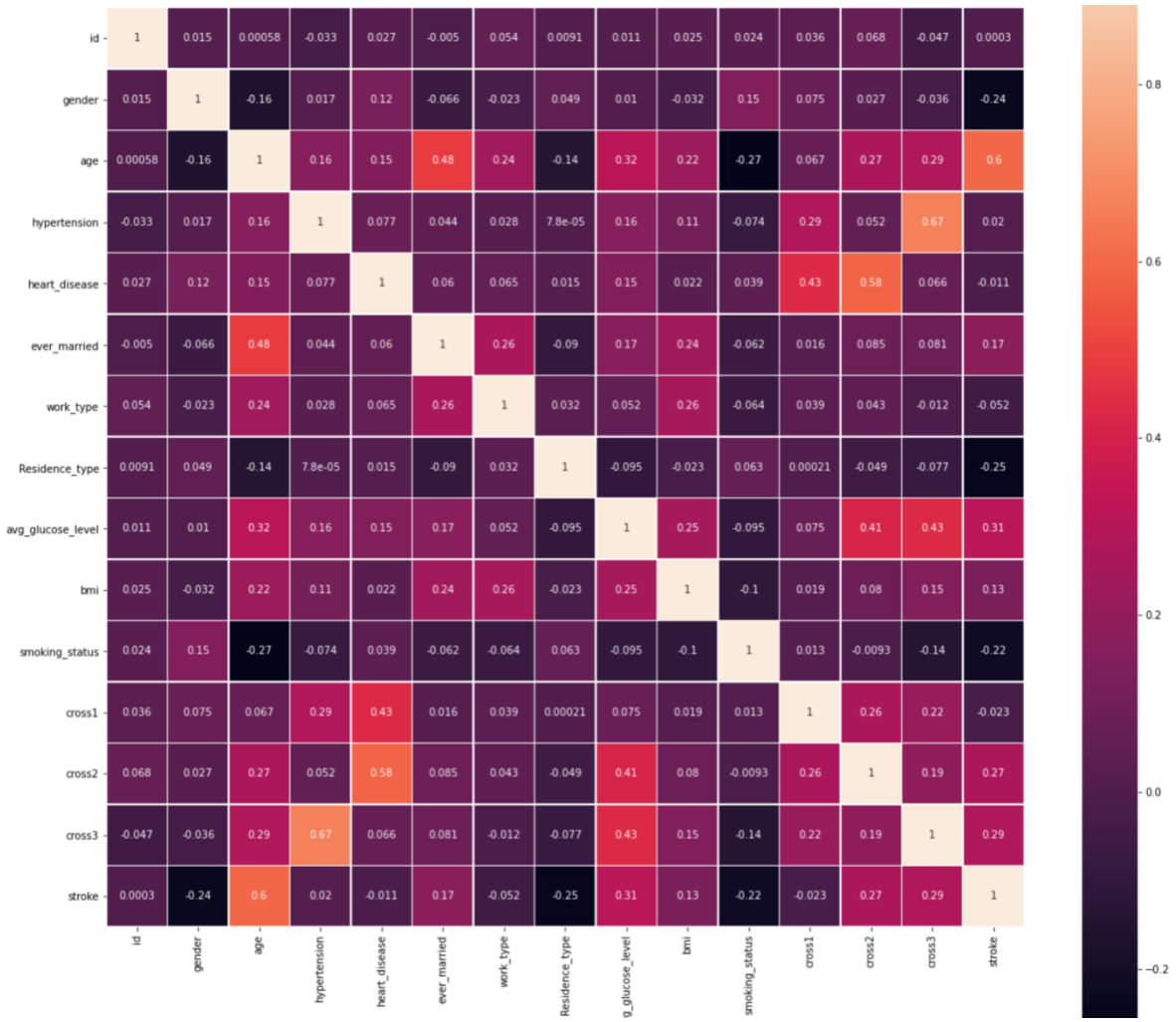
The results of the correlations found in the data are summarized in the below correlation matrices. The first matrix was created on unbalanced data.



We found slight correlation between a patient having a stroke and the rest of the variables (age has the largest positive correlation at 0.23). We also see positive correlations with hypertension, heart disease, having been married and the average glucose level (correlations at around 0.11-0.14). The fact that features show very little correlation may be due to the fact that our dataset is not balanced yet. We will thus plot another correlation matrix on balanced data later.

Other correlations found are between age and having been married (positive correlation at 0.63: the older one is the more likely they are to have been married, this makes sense). BMI is also correlated positively with age (0.33) and with having been married (0.34). The older one is the more likely they are to be overweight. You usually gain weight more easily as you age, so this makes sense. Average glucose levels are slightly positively correlated with age, hypertension, heart disease, having been married, bmi and of course stroke. The rest of the datapoints are very little correlated if at all (close to zero).

Then, after balancing data using a synthetic oversampling approach, we obtained another correlation matrix.



We can now see that the correlations are much more apparent now that we are working with a balanced dataset.

Stroke is still positively correlated with bmi (0.13), having been married (0.17), and strongly positively correlated with average glucose level (now at 0.31), and age (now at 0.6). We also see two more interesting correlations: gender (-0.24, which means that females are at a higher likelihood of having a stroke according to our data, as we encoded female as 0 and male as 1) and residence type (-0.25, which means that people living in rural areas are more likely to be hit with a heart stroke according to our data) as well as smoking_status (-0.22 correlation) which means that our choice of cardinal integer encoding was wise. Surprisingly, we lost correlation with heart_disease. ID can safely be dismissed and imputed from the data. It seems wise to keep most of the rest of our data.

2. Data extraction, augmentation, balancing and pipeline

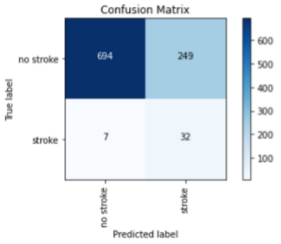
```
stroke               1.000000
age                  0.598934
avg_glucose_level    0.309983
cross3               0.286801
cross2               0.271490
ever_married         0.173244
bmi                  0.130947
hypertension         0.019821
id                   0.000295
heart_disease       -0.010723
cross1              -0.022722
work_type           -0.051729
smoking_status      -0.215496
gender              -0.239293
Residence_type      -0.249008
Name: stroke, dtype: float64
```

Now, we notice that heart disease and hypertension show little correlation with having a stroke or not. We see however that when crossed with avg_glucose_level, their predictive power is almost that of avg_glucose_level alone (their correlations is close to 0.3, see cross2 and cross3 respectively above with cross2=avg_glucose_level*heart_disease and cross3= avg_glucose_level*hypertension). As we can see, we've found 2 very interesting feature crosses, utilizing the heart_disease and hypertension features which give correlations with having a stroke when crossed with avg_glucose_level higher than the weighted sum of their parts. We can thus drop the features hypertension, id and heart_disease completely from our main dataset (since they all have weak correlations), and augment it with the crossed features found above. Since there is little correlation between work_type and stroke (likely due to our random choice of numerical values), we will simply one-hot encode that variable.

3. Logistic Regression

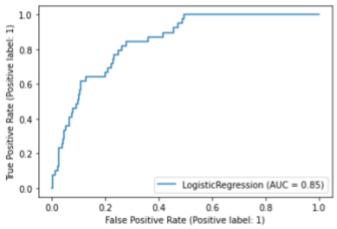
After pipelining our dataset, we moved on to our first model: logisitic regression. First, we implemented it with no penalty (or regularization).

```
Accuracy:        0.7393075356415478
Precision:       0.11387900355871886
Recall:          0.8205128205128205
F1 Score:        0.2
```




```
metrics.plot_roc_curve(LogisticRegression().fit(train_features,
                                                train_labels), test_features, test_labels)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x126af5b20>
```



By running my first logistic regression model, I find that my model gives high accuracy and recall scores, and relatively low precision and F1 scores. What this means is that my first logistic regression model is good at correctly identifying many positives present in the actual datase (high recall), but that it doesn't perform well at not mislabeling negatives as a positives (low precision). 32 strokes out of 39 were predicted correctly, but 249 healthy patients were predicted incorrectly to have a stroke. Usually, in fields like medicine, a high recall score is preferrable over a low one, because it is critical to identify as many of the positive cases as possible, even if that means negative cases will be incorrectly classified as being positive, since true positives need immediate medical care and false positives can undergo fuirther testing to determine that they were actually healthy. Nonetheless, we attempted to improve on our logisitic regression model by adding L1 or L2 regularization but it did not improve from using either regularization, nor from using the sag solver. Let us thus move onto the next model.

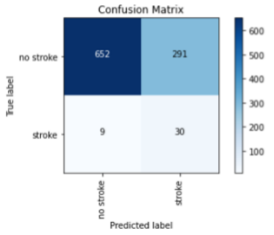
4. Principal Component Analysis

After applying PCA to our dataset, we are left with 8 dimensions in the feature space as opposed to 10.

	pc1	pc2	pc3	pc4	pc5	pc6	pc7	pc8
0	3.866884	4.220221	-0.981397	0.171600	0.740736	-3.806074	3.259292	0.291216
1	2.283763	0.884100	0.586161	-1.338283	-0.335083	-2.561618	0.980061	0.866738
2	0.878777	0.152974	-1.339958	0.418035	1.423048	1.020784	0.525651	-1.249726
3	2.941694	1.059676	2.099047	1.400971	-0.062669	0.975800	-2.671236	0.337873
4	1.781526	0.532277	-1.090521	-0.226099	-0.948890	-0.138875	-0.153028	-1.729764

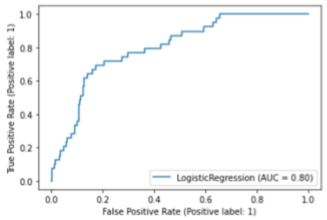
The dataset performed slightly worse on logistic regression. This is due to the fact that some data is removed through dimensionality reduction. Indeed, we lost 10% of the variance in the data.

```
Accuracy:        0.6945010183299389
Precision:       0.09345794392523364
Recall:          0.7692307692307693
F1 Score:        0.16666666666666666
```




```
metrics.plot_roc_curve(LogisticRegression().fit(train_features_pca,
                                                train_labels_pca), test_features_pca, test_labels_pca)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1264f0e20>
```



5. Ensemble Methods

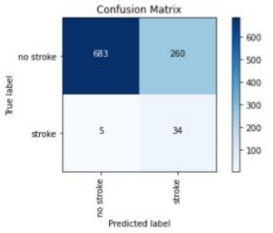
We are able to, through the adaboost ensemble method, increase our recall score from the logistic regression result by using a low number of trees and low maximum depth for each tree, as well as a learning rate of 3. Our recall is 84% this time, with an accuracy of 66%.
For our random forest algorithm, we still obtain precision and F1 scores on the lower end, and a 68% accuracy as well as a 76% recall. See our code below for the ROC curves and related confusion matrices. It seems that our test cohort cannot achieve high precision due to its mislabeling of negatives as positives (high rate of false positives due to unbalanced data that presents an overhwleming amount of negatives).

6. Neural Network Classifier

The neural network model performs well when it comes to recall (87%) and accuracy (73%). The detail of the parameters is given in the methodology section of this report.

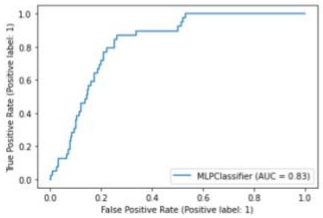
```
# Display statistics from our model
# Report 4 metrics
print("Accuracy:\t", metrics.accuracy_score(test_labels, neuralNetPreds))
print("Precision:\t", metrics.precision_score(test_labels, neuralNetPreds))
print("Recall:\t\t", metrics.recall_score(test_labels, neuralNetPreds))
print("F1 Score:\t", metrics.f1_score(test_labels, neuralNetPreds))
# Plot confusion matrix
draw_confusion_matrix(test_labels, neuralNetPreds, ["no stroke", "stroke"])
```

```
Accuracy:       0.730142566191446
Precision:      0.11564625850340136
Recall:         0.8717948717948718
F1 Score:       0.2042042042042042
```




```
metrics.plot_roc_curve(neuralNetFit, test_features, test_labels)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1263e1ee0>
```



Precision and F1 scores are still low at 11% and 20%. It seems that even after balancing our data prevents our predictive models have a high rate of false positives.

7. Cross-Validation

From our 10-fold cross validation, we see that overall, the neural network model performs better, with higher accuracy, precision and F1 score. We should nonetheless note that the k-fold cross validation method used is imperfect because it doesn't let us test our model on unbalanced data as we did throughout this project (each fold is trained and tested on balanced data which will inevitably lead to higher scores overall). This means that the high scores we obtain from 10-fold cross validation are not representative of our models, and one should keep in mind that our model performs worse on precision and f1 score if given a test cohort with the same unbalanced distribution as is found in the raw data. Nonetheless, we are able to use 10-fold cross-validation to assert that our neural network model likely would perform better than our ensemble (random forests) method, even on unbalanced data.

```
10-Fold Cross Validation for Random Forests:
Accuracy:        78.6019418575669
Precision:       73.97216429450818
Recall:          88.27784391110754
F1 Score:        80.47546497512046


10-Fold Cross Validation for Neural Network:
Accuracy:        79.63426460924931
Precision:       75.61200085016125
Recall:          87.96358027478274
F1 Score:        81.19729812091649
```

8. Experimenting with my Own Models

```
Accuracy, precision, recall, F1 score for k = 1 :        0.90 0.10 0.21 0.13
Accuracy, precision, recall, F1 score for k = 2 :        0.91 0.11 0.18 0.14
Accuracy, precision, recall, F1 score for k = 3 :        0.87 0.11 0.31 0.16
Accuracy, precision, recall, F1 score for k = 5 :        0.84 0.09 0.36 0.15
Accuracy, precision, recall, F1 score for k = 7 :        0.81 0.09 0.41 0.15
Accuracy, precision, recall, F1 score for k = 9 :        0.79 0.10 0.51 0.16
Accuracy, precision, recall, F1 score for k = 10 :       0.81 0.10 0.51 0.17
Accuracy, precision, recall, F1 score for k = 20 :       0.74 0.09 0.62 0.16
Accuracy, precision, recall, F1 score for k = 50 :       0.69 0.09 0.69 0.15
Accuracy, precision, recall, F1 score for k = 60 :       0.69 0.09 0.72 0.16
Accuracy, precision, recall, F1 score for k = 65 :       0.68 0.09 0.74 0.16
Accuracy, precision, recall, F1 score for k = 70 :       0.68 0.09 0.77 0.16
Accuracy, precision, recall, F1 score for k = 75 :       0.66 0.08 0.74 0.15
Accuracy, precision, recall, F1 score for k = 100 :      0.65 0.08 0.74 0.14
Accuracy, precision, recall, F1 score for k = 200 :      0.63 0.08 0.82 0.15
Accuracy, precision, recall, F1 score for k = 300 :      0.62 0.08 0.85 0.15
Accuracy, precision, recall, F1 score for k = 500 :      0.63 0.09 0.85 0.15
```

From our nearest neighbor model, we see that the precision and f1 scores are again pretty low and that accuracy is at the highest for smaller values of k and decreases as k increases, whereas recall increase as k increases and is large for large values of k. Our model thus performs best overall at around k=70 where we have an accuracy of 68% and a recall of 77%.

```
Accuracy, precision, recall, F1 score for kernel = linear :              0.72 0.11 0.85 0.19
Accuracy, precision, recall, F1 score for kernel = rbf :                 0.79 0.11 0.62 0.19
Accuracy, precision, recall, F1 score for kernel = sigmoid :             0.69 0.08 0.64 0.14
Accuracy, precision, recall, F1 score for kernel = poly of degree 2 :    0.76 0.11 0.69 0.19
Accuracy, precision, recall, F1 score for kernel = poly of degree 3 :    0.79 0.11 0.56 0.18
Accuracy, precision, recall, F1 score for kernel = poly of degree 4 :    0.84 0.12 0.51 0.20
```

We see that our support vector machine model performs best for a linear kernel. Indeed it returns a 72% accuracy and an 85% recall. This means that our linear kernel SVM model performs better than our 70-nearest neighbor predictive model.

```
Accuracy, precision, recall, F1 score for variance smoothing = 1e-09 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 1e-08 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 1e-07 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 1e-06 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 1e-05 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.0001 :  0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.001 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.01 :    0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.1 :     0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.5 :     0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.9 :     0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 1 :       0.84 0.12 0.51 0.20
```

The Naive Bayes algorithm seems to perform worse compared to our two other models in terms of recall score, with 84% accuracy and 51% recall.

After running 10-fold cross validation (balanced data for both training and testing cohorts as before) on the nearest neighbor model at k=70 and the SVM model with linear kernel, we found that our nearest neighbor algorithm seems to perform best on fully balanced data (on the 10-fold cross validation), whereas the support vector machine algorithm performed better on test data that's not synthetically balanced, as we saw reported just above. We stress the importance of that

distinction, as the eventual distribution of data that one's model is tested on can thus interfere with the model's performance metrics to a great degree.

5. Discussion

The results obtained in this project, from all models, had relatively high accuracy and recall scores. A recall score is the ratio of the number of true positives identified by a predictive model to the number of true positives and false negatives identified by the predictive model. What this example showed us is that when working with unbalanced data (with high numbers of helahty people and few people with stroke, in this case), it is difficult for predictive models to limit the amount of false positives they will identify, if they are to identify as many of the true positives as necessary. Another way to say this is that recall is likely one of the most important performance metric when it comes to classifying binary risks of critical health conditions such as stroke. This is due to the fact that health care professionals would like for most of the true positives (truly at risk for stroke) to be identified as such, so that they can receive the necessary health care and preventative measures. The high flase positive rate is not as worrisome, as further testing can show that these people are helahty but identifying everyone who's at risk, ie minimizing the false negative rate (misses) is critical to achieving our real world goals of minimizing deaths, injuries, poor health and prevent further economic impact.

Our neural network model performed the best in terms of recall out of all the other models (at an 87% recall score on unbalanced, or unaltered, test data). I would thus recommend the UCLA hospital to use that predictive model preferentially and warn them of its shortcomings (ie the low precision or the high false positive rate). They could hence use this tool as one predictor of stroke for their patients in addition to further physical testing/office visits. They should however tell their patients about the high false positive rate that it incurs.

The next steps for analytic work should be the refine the model through multiple ways. Firstly, more data needs to be collected (more fields so as to come up with an even more powerful model). Secondly, an even higher-performing neural network algorithm could be found by iterating through the possible paramters and hidden layer sizes, number of nodes etc. This task could take up a lot of computing power and time. For this project I could only come up with a recall of 87%, however, the metrics can surely be improved by a more efficient combination of the parameters.

6. Conclusion

Stroke is the second leading cause of death in the world according to the WHO, but machine learning could help lessen the impacts on human lives and economies that this health condition has every year. Indeed, if warning signs are heeded, proper health care can be administered and preventative measures such as diet and lifestyle adjustments can be implemented so as to lead to better health and economic outcome for individuals and collectives. High-performance predictive models such as neural networks, decision trees and logistic regressions can correctly identify a majority of patients at risk of strokes from a list of attributes, while minimizing the number of people at risk that they miss, as demonstrated throughout this project.

CSM148 Project 3 - Binary Classification for Heart Stroke Data

For this project we're going to attempt a binary classification of a dataset for determining whether a patient has a heart stroke or not based on a host of potential medical factors.

Background: The Dataset

For this exercise we will be using a dataset on health care of patients with or without heart strokes, leveraging the eleven most commonly used attributes.

The dataset includes 12 columns. The information provided by each column is as follows:

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

Loading Essentials and Helper Functions

```
]:
#Here are a set of libraries we imported to complete this assignment.
#Feel free to use these or equivalent libraries for your implementation
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt # this is used for the plot the graph
import os
import seaborn as sns # used for plot interactive graph.
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn import metrics
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix
import sklearn.metrics.cluster as smc
from sklearn.model_selection import KFold


from matplotlib import pyplot
import itertools

%matplotlib inline

import random

random.seed(42)
```

```
]:
# Helper function allowing you to export a graph
def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

```
# Helper function that allows you to draw nicely formatted confusion matrices
def draw_confusion_matrix(y, yhat, classes):
    '''
        Draws a confusion matrix for the given target and predictions
        Adapted from scikit-learn and discussion example.
    '''
    plt.cla()
    plt.clf()
    matrix = confusion_matrix(y, yhat)
    plt.imshow(matrix, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")
    plt.colorbar()
    num_classes = len(classes)
    plt.xticks(np.arange(num_classes), classes, rotation=90)
    plt.yticks(np.arange(num_classes), classes)

    fmt = 'd'
    thresh = matrix.max() / 2.
    for i, j in itertools.product(range(matrix.shape[0]), range(matrix.shape[1])):
        plt.text(j, i, format(matrix[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if matrix[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.show()
```

1. Load the Data and Run Basic Statistics on Your Variables

Let's first load our dataset so we'll be able to work with it. (correct the relative path if your notebook is in a different directory than the csv file.)

```
def load_heartstroke_data(heartstroke_path):
    '''
        loads heartstroke.csv dataset stored

        Args:
            heartstroke_path (str): path to folder containing heartstroke dataset

        Returns:
            pd.DataFrame
    '''
    csv_path = os.path.join(heartstroke_path, "healthcare-dataset-stroke-data.csv")
    return pd.read_csv(csv_path)

DATASET_PATH = os.path.join(".")

heartstroke = load_heartstroke_data(DATASET_PATH) # we load the pandas dataframe
```

Now that our data is loaded, let's take a closer look at the dataset we're working with. Use the head method, the describe method, and the info method to display some of the rows so we can visualize the types of data fields we'll be working with.

```
heartstroke.head() # show the first few elements of the dataframe
```

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

```
heartstroke.describe() # show the overall statistics for our dataset
```

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

```
heartstroke.info() # show the types of the targets
```

```
<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
```

As we can see, we have 3 float fields, 4 integer fields and 5 categorical values. We should be able to convert some of these categorical value fields into numerical fields so as to operate on them. For example, some are booleans like "ever_married" which can be encoded as 1 for "Yes", 0 for "No". "Residence_type" can be encoded as 0="Rural", 1="Urban". Other categoricals will require more specific encoding. We can use an integer encoding, with 0="never smoked", 1="formerly smoked", 2="Unknown", 3="smokes" since we can assume a reasonable cardinality in the data (for example a linear relationship between the amount one smokes and the probability to get a heart stroke). The work_type seems like a good data point to one-hot encode since there is no obvious cardinal relationship between the different values it can take ("children", "Govt_jov", "Never_worked", "Private" or "Self-employed"). Gender is another categorical which we might want to convert into 0 or 1 (male=1, female=0), and "Other" can be simply dropped as only one row contains this value (as shown below), thus our model will perform better if we exclude it and encode gender as a simple boolean. ID likely serves no purpose when it comes to predicting and thus will be imputed completely.

```
# Look at distribution of the gender categorical variable
heartstroke["gender"].value_counts()
```

```
Female    2994
Male      2115
Other        1
Name: gender, dtype: int64
```

```
# Look at distribution for the smoking_status categorical variable
heartstroke["smoking_status"].value_counts()
```

```
never smoked       1892
Unknown            1544
formerly smoked     885
smokes              789
Name: smoking_status, dtype: int64
```

```
heartstroke["Residence_type"].value_counts()
```

```
Urban    2596
Rural    2514
Name: Residence_type, dtype: int64
```

```
heartstroke["work_type"].value_counts()
```

```
Private          2925
Self-employed     819
children          687
Govt_job          657
Never_worked       22
Name: work_type, dtype: int64
```

Determine if we're dealing with any null values.

```
sample_incomplete_rows = heartstroke[heartstroke.isnull().any(axis=1)].head()
sample_incomplete_rows
```

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
1	51676	Female	61.0	0	0	Yes	Self-employed	Rural	202.21	NaN	never smoked	1
8	27419	Female	59.0	0	0	Yes	Private	Rural	76.15	NaN	Unknown	1
13	8213	Male	78.0	0	1	Yes	Private	Urban	219.84	NaN	Unknown	1
19	25226	Male	57.0	0	1	No	Govt_job	Urban	217.08	NaN	Unknown	1
27	61843	Male	58.0	0	0	Yes	Private	Rural	189.84	NaN	Unknown	1

As our code shows above, our heartstroke dataframe contains some null values, concentrated at the bmi data field. Since we are given that there's 4909 non-null values for bmi out of 5110 data points, we conclude that 201 bmi datapoints are missing, which represents roughly 4% of our data frame. The BMI seems like such an important predictor for heart disease (intuitively) that it would be unwise to extrapolate or assign the median/average BMI on the missing rows. Let us simply remove the 201 rows altogether.

```
heartstroke=heartstroke.dropna(subset=["bmi"]) # drop the incomplete rows
sample_incomplete_rows = heartstroke[heartstroke.isnull().any(axis=1)].head() # check that the rows were dropped
sample_incomplete_rows
```

id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke

```
# Now let's remove the row with an "other" for gender
heartstroke.loc[heartstroke['gender'] == 'Other']
```

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

```
# remove it
heartstroke = heartstroke.drop(3116)
heartstroke.loc[heartstroke['gender'] == 'Other']
```

id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke

Before we begin our analysis we need to fix the field(s) that will be problematic. Specifically convert our gender, Residence_type and ever_married variables into binary numeric target variables (values of either '0' or '1')

```
# Change gender, Residence_type, ever_married, smoking_status and work_type
cleanup = {"gender":     {"Male": 1, "Female": 0},
           "Residence_type": {"Rural": 0, "Urban": 1},
           "ever_married": {"Yes": 1, "No": 0},
           "smoking_status": {"never smoked": 0, "Unknown": 2, "formerly smoked": 1, "smokes": 3},
           "work_type": {"children": 1, "Govt_job": 4, "Never_worked": 0, "Private": 3, "Self-employed": 2}
          }
heartstroke = heartstroke.replace(cleanup)
# Print first 5 rows to check
heartstroke.head()
```

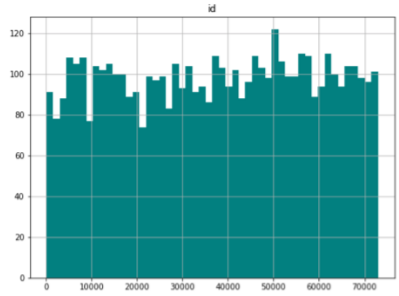
	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	9046	1	67.0	0	1	1	3	1	228.69	36.6	1	1
2	31112	1	80.0	0	1	1	3	0	105.92	32.5	0	1
3	60182	0	49.0	0	0	1	3	1	171.23	34.4	3	1
4	1665	0	79.0	1	0	1	2	0	174.12	24.0	0	1
5	56669	1	81.0	0	0	1	3	1	186.21	29.0	1	1

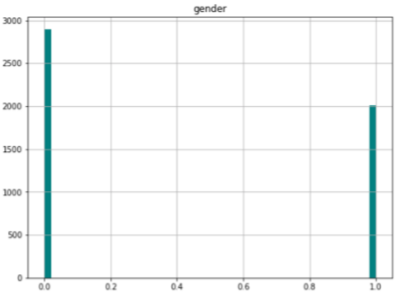
```
# Check the type of the sick variable
type(heartstroke["gender"][0])
```

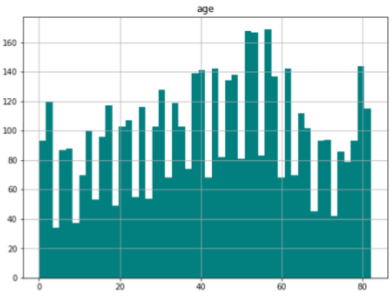
```
numpy.int64
```

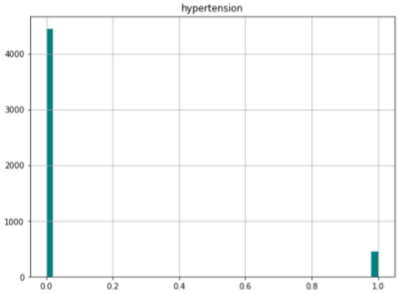
Now that we have a feel for the data-types for each of the variables, let's plot histograms of each field.

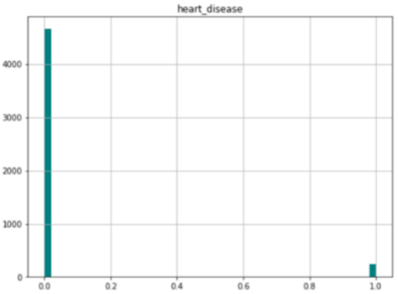
```
# We can draw a histogram for each of the dataframes features
# using the hist function
heartstroke.hist(bins=50, figsize=(30,30), color="teal")
# save_fig("attribute_histogram_plots")
plt.show() # pandas internally uses matplotlib, and to display all the figures
           # the show() function must be called
```

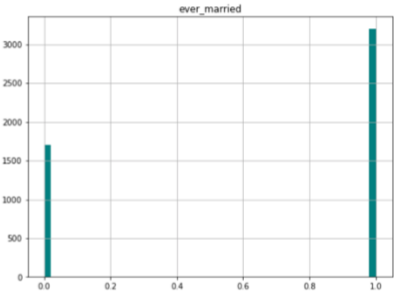

id
0
10000
20000
30000
40000
50000
60000
70000
0
20
40
60
80
100
120

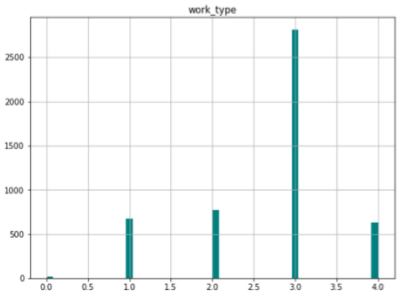

gender
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
2500
3000

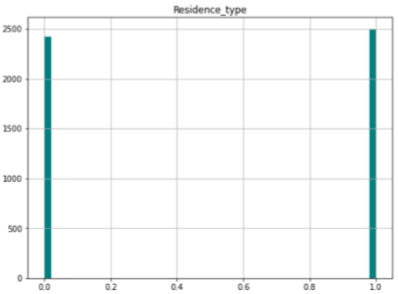

age
0
20
40
60
80
0
20
40
60
80
100
120
140
160

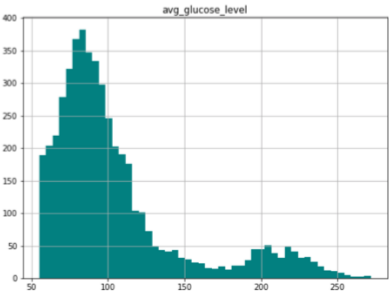

hypertension
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000

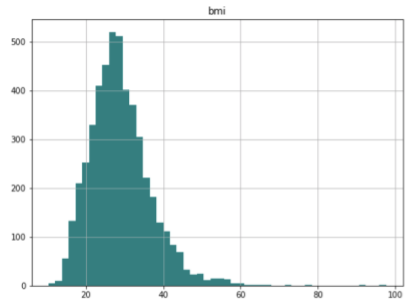

heart_disease
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000

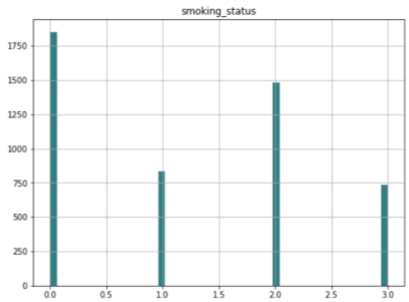

ever_married
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
2500
3000

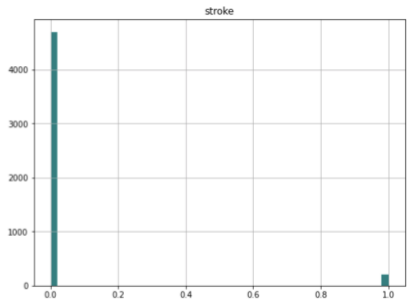

work_type
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
500
1000
1500
2000
2500


Residence_type
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
2500


avg_glucose_level
50
100
150
200
250
0
50
100
150
200
250
300
350
400



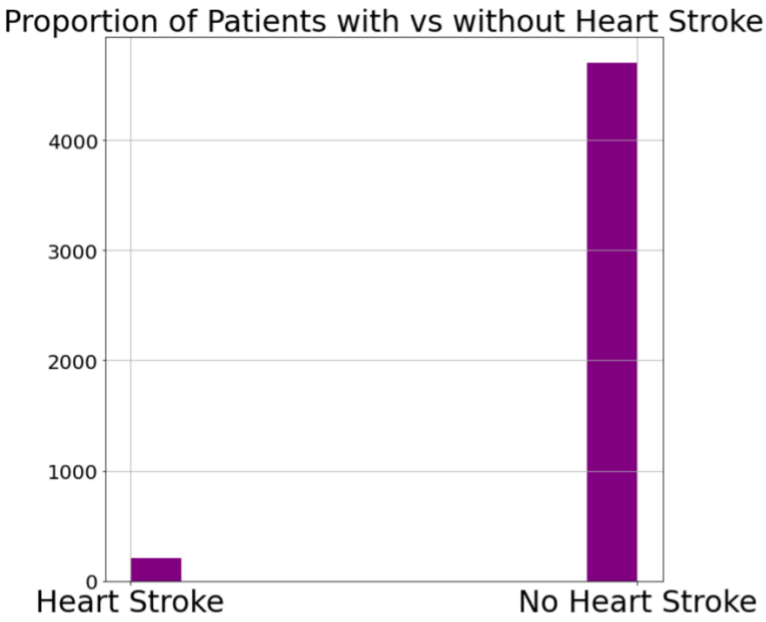




From our data distribution, we see that our dataset is not balanced. Specifically, we have many more patients without a stroke than with a stroke. We will thus need to balance our dataset. There are slightly fewer men than women, bmi is a bell shape distribution centered at an average of 28.8. We also have a lot more people healthy from heart disease than people sick with it. It may be that heart disease is a great predictor of stroke, just looking at their two histograms side by side.

Let's look at our inequitable number of sick and healthy individuals (insufficiently balanced dataset) more closely. Let's plot a histogram specifically of the stroke target, and conduct a count of the number of sick and healthy individuals and report on the results:

```
pd.cut(heartstroke["stroke"],
       bins=[-1, 0.5, np.inf],
       labels=["No Heart Stroke", "Heart Stroke"]).hist(color="purple",
                                        xlabelsize=30,
                                        figsize=(10,10),
                                        ylabelsize=20)
plt.title("Proportion of Patients with vs without Heart Stroke", size=30)
plt.show()
```



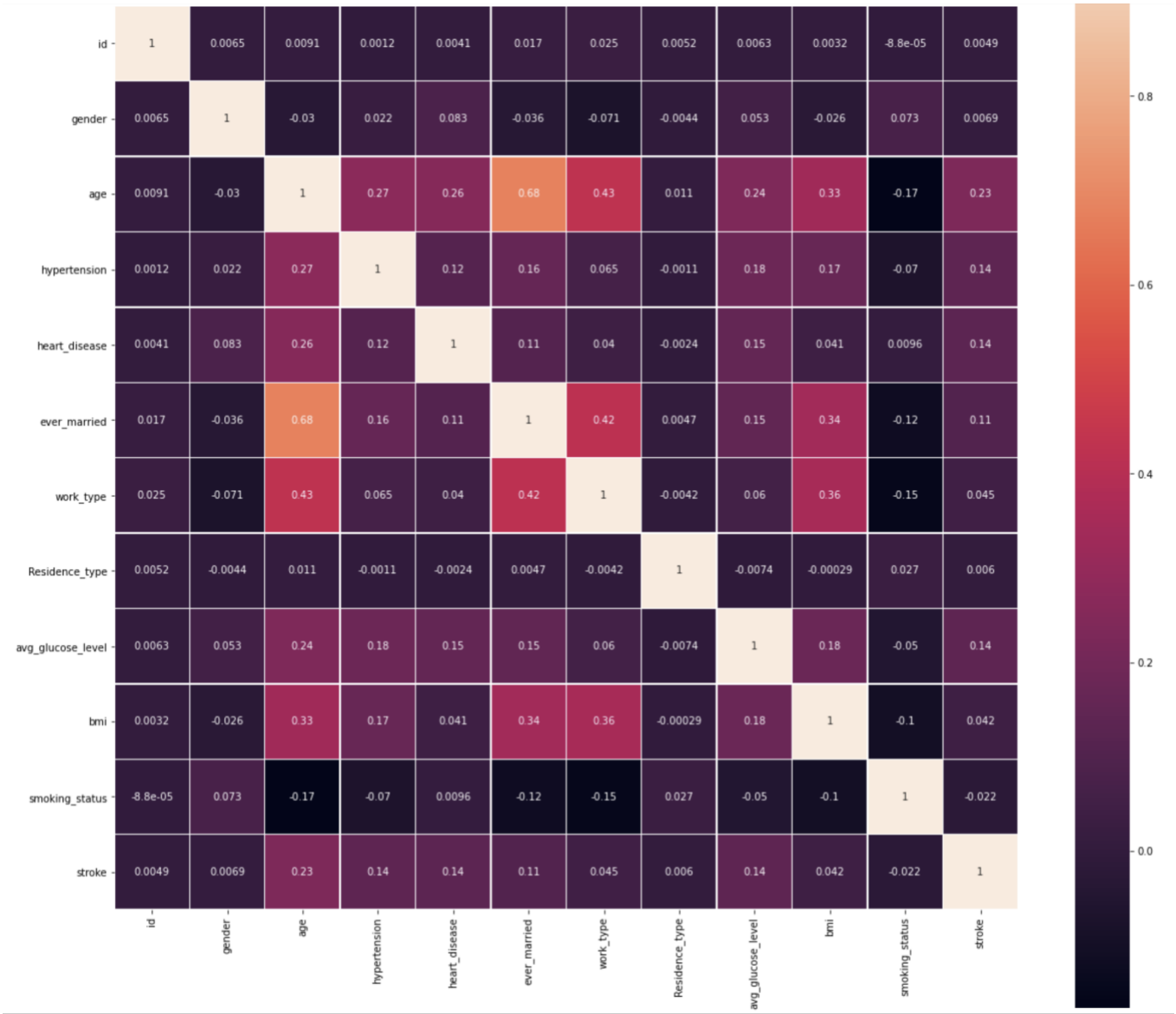
[20]:
```
pd.cut(heartstroke["stroke"], bins=[-1, 0.5, np.inf],
       labels=["No Heart Stroke", "Heart Stroke"]).value_counts()
```

[20]:
```
No Heart Stroke    4699
Heart Stroke        209
Name: stroke, dtype: int64
```

We found that there were 209 patients who got a heart stroke and 4699 patients who didn't in the dataset. We need to balance our dataset. We will use the following approach. Let's use an oversampling method (make up synthetic data using k nearest neighbors). Although, we should first pipeline and split our data into train and test cohorts so as to not introduce bias into our test cohort.

Now that we have our dataframe prepared let's start analyzing our data. Let's look at the correlations of our variables to our target value.

[21]:
```
# Create the corr_matrix
corr_matrix=heartstroke.corr()

# Plot the heatmap for the correlation matrix
fig, ax = plt.subplots(figsize=(20,20))
sns.heatmap(corr_matrix,
            xticklabels=corr_matrix.columns,
            yticklabels=corr_matrix.columns,
            linewidths=.5,
            square=True,
            annot=True,
            ax=ax)
plt.show()
```


id 1 0.0065 0.0091 0.0012 0.0041 0.017 0.025 0.0052 0.0063 0.0032 -8.8e-05 0.0049
gender 0.0065 1 -0.03 0.022 0.083 -0.036 -0.071 -0.0044 0.053 -0.026 0.073 0.0069
age 0.0091 -0.03 1 0.27 0.26 0.68 0.43 0.011 0.24 0.33 -0.17 0.23
hypertension 0.0012 0.022 0.27 1 0.12 0.16 0.065 -0.0011 0.18 0.17 -0.07 0.14
heart_disease 0.0041 0.083 0.26 0.12 1 0.11 0.04 -0.0024 0.15 0.041 0.0096 0.14
ever_married 0.017 -0.036 0.68 0.16 0.11 1 0.42 0.0047 0.15 0.34 -0.12 0.11
work_type 0.025 -0.071 0.43 0.065 0.04 0.42 1 -0.0042 0.06 0.36 -0.15 0.045
Residence_type 0.0052 -0.0044 0.011 -0.0011 -0.0024 0.0047 -0.0042 1 -0.0074 -0.00029 0.027 0.006
avg_glucose_level 0.0063 0.053 0.24 0.18 0.15 0.15 0.06 -0.0074 1 0.18 -0.05 0.14
bmi 0.0032 -0.026 0.33 0.17 0.041 0.34 0.36 -0.00029 0.18 1 -0.1 0.042
smoking_status -8.8e-05 0.073 -0.17 -0.07 0.0096 -0.12 -0.15 0.027 -0.05 -0.1 1 -0.022
stroke 0.0049 0.0069 0.23 0.14 0.14 0.11 0.045 0.006 0.14 0.042 -0.022 1
id
gender
age
hypertension
heart_disease
ever_married
work_type
Residence_type
avg_glucose_level
bmi
smoking_status
stroke
0.8
0.6
0.4
0.2
0.0

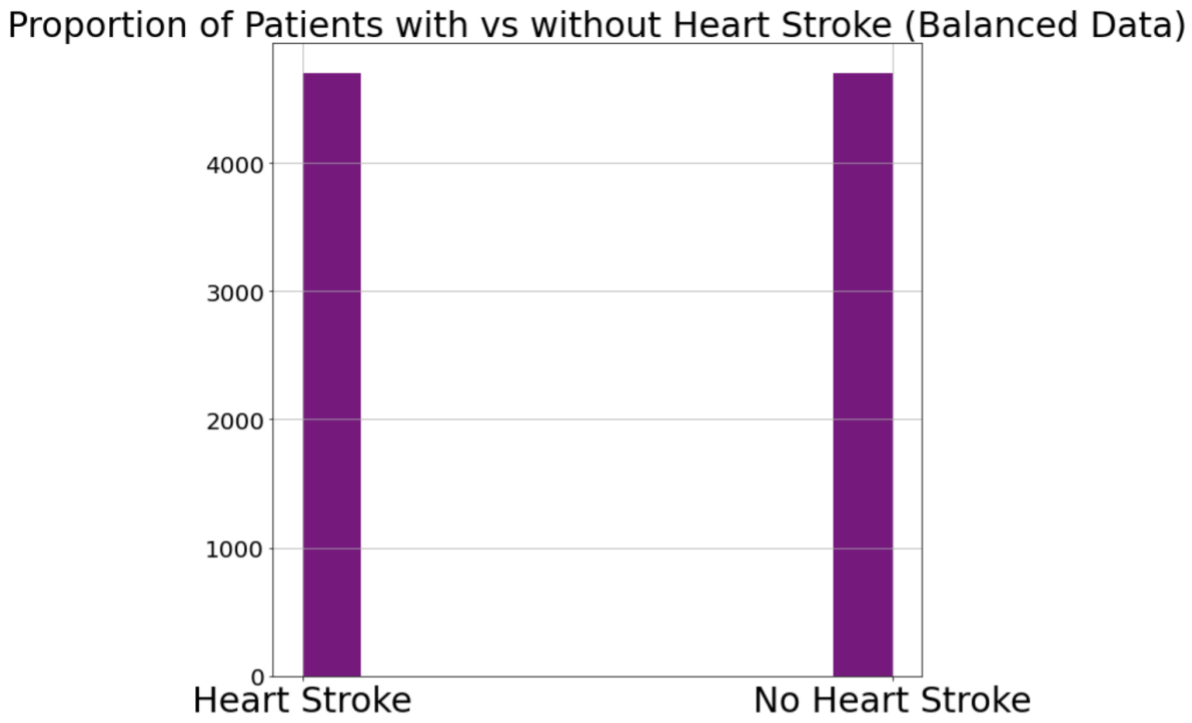
We found slight correlation between a patient having a stroke and the rest of the variables (age has the largest positive correlation at 0.23). We also see positive correlations with hypertension, heart disease, having been married and the average glucose level (correlations at around 0.11-0.14). The fact that features show very little correlation may be due to the fact that our dataset is not balanced yet. We will thus plot another correlation matrix on balanced data later.

Other correlations found are between age and having been married (positive correlation at 0.63: the older one is the more likely they are to have been married, this makes sense). BMI is also correlated positively with age (0.33) and with having been married (0.34). The older one is the more likely they are to be overweight. You usually gain weight more easily as you age, so this makes sense. Average glucose levels are slightly positively correlated with age, hypertension, heart disease, having been married, bmi and of course stroke. The rest of the datapoints are very little correlated if at all (close to zero).

Now, let us plot the correlation matrix on a balanced dataset. Let us use an oversampling method to do so.

```
# Make a copy and augment some features, then balance
heartstrokeCopy=heartstroke.copy()
# Perform multiple feature crosses/augmentations and add them to our balanced dataset
heartstrokeCopy["cross1"] = heartstrokeCopy["heart_disease"]*heartstrokeCopy["hypertension"]
heartstrokeCopy["cross2"] = heartstrokeCopy["avg_glucose_level"]*heartstrokeCopy["heart_disease"]
heartstrokeCopy["cross3"] = heartstrokeCopy["avg_glucose_level"]*heartstrokeCopy["hypertension"]
```

```
# Balance the dataset
from imblearn.over_sampling import SMOTE

# Resample the minority class
sm = SMOTE(random_state=27, sampling_strategy='minority')
heartstrokeBalanced_X, heartstrokeBalanced_Y = sm.fit_resample(heartstrokeCopy.drop(['stroke'], axis=1), heartstrokeCopy['stroke'])
heartstrokeBalanced = pd.concat([pd.DataFrame(heartstrokeBalanced_X), pd.DataFrame(heartstrokeBalanced_Y)], axis=1)

# Make sure that the dataset is now balanced
pd.cut(heartstrokeBalanced["stroke"],
       bins=[-1, 0.5, np.inf],
       labels=["No Heart Stroke", "Heart Stroke"]).hist(color="purple",
                                    xlabelsize=30,
                                    figsize=(10,10),
                                    ylabelsize=20)
plt.title("Proportion of Patients with vs without Heart Stroke (Balanced Data)", size=30)
plt.show()
```

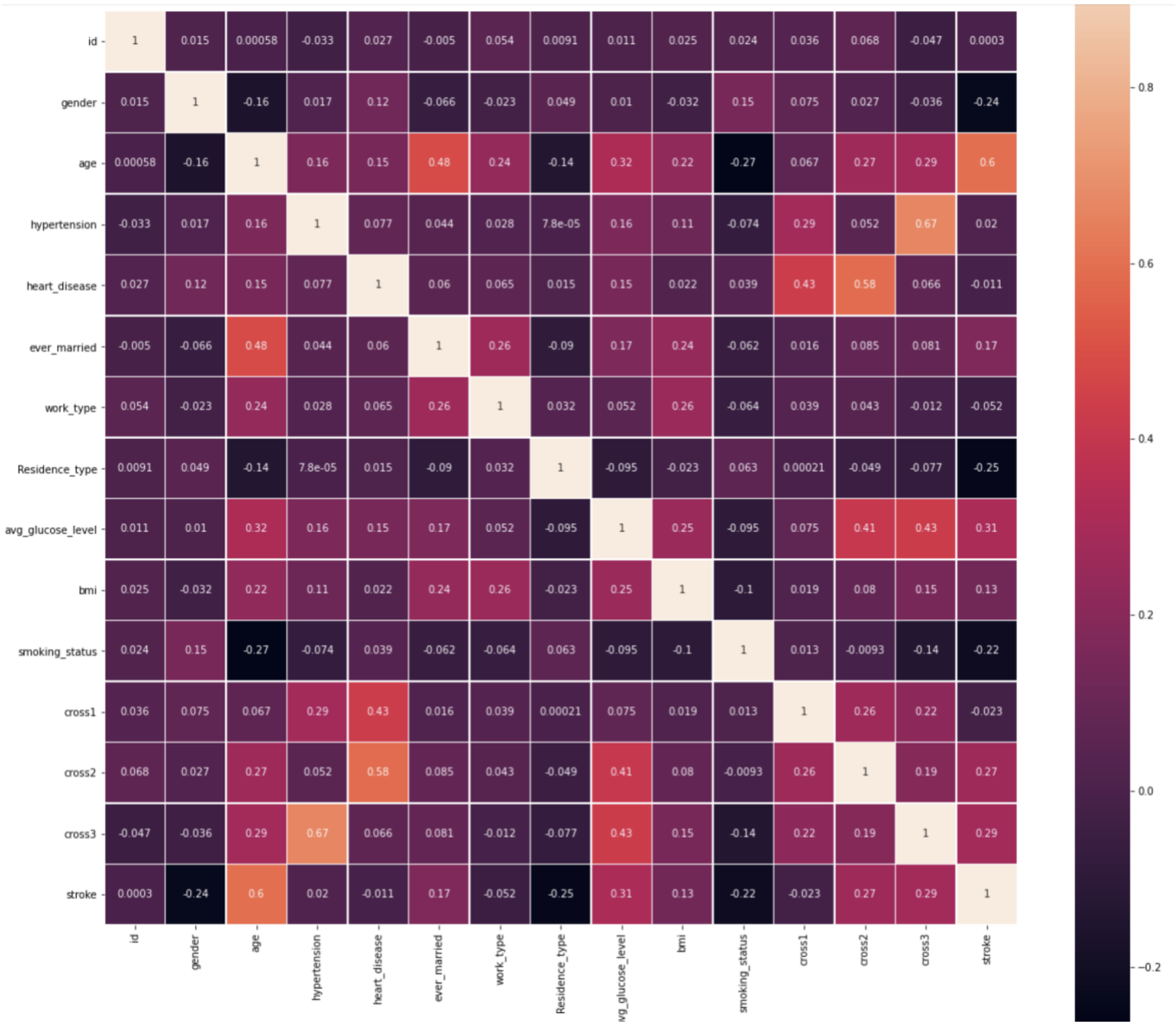



```
pd.cut(heartstrokeBalanced["stroke"], bins=[-1, 0.5, np.inf],
       labels=["No Heart Stroke", "Heart Stroke"]).value_counts()
```

```
Heart Stroke       4699
No Heart Stroke    4699
Name: stroke, dtype: int64
```

As one can see, our balanced dataset now has as many heart stroke datapoints (synthetically created using SMOTE) as no heart stroke data points. Let us now plot the correlation matrix again.

```
# Plot the new correlation matrix
corr_matrix=heartstrokeBalanced.corr()

# Plot the heatmap for the correlation matrix
fig, ax = plt.subplots(figsize=(20,20))
sns.heatmap(corr_matrix,
            xticklabels=corr_matrix.columns,
            yticklabels=corr_matrix.columns,
            linewidths=.5,
            square=True,
            annot=True,
            ax=ax)
plt.show()
```


id 1 0.015 0.00058 -0.033 0.027 -0.005 0.054 0.0091 0.011 0.025 0.024 0.036 0.068 -0.047 0.0003
gender 0.015 1 -0.16 0.017 0.12 -0.066 -0.023 0.049 0.01 -0.032 0.15 0.075 0.027 -0.036 -0.24
age 0.00058 -0.16 1 0.16 0.15 0.48 0.24 -0.14 0.32 0.22 -0.27 0.067 0.27 0.29 0.6
hypertension -0.033 0.017 0.16 1 0.077 0.044 0.028 7.8e-05 0.16 0.11 -0.074 0.29 0.052 0.67 0.02
heart_disease 0.027 0.12 0.15 0.077 1 0.06 0.065 0.015 0.15 0.022 0.039 0.43 0.58 0.066 -0.011
ever_married -0.005 -0.066 0.48 0.044 0.06 1 0.26 -0.09 0.17 0.24 -0.062 0.016 0.085 0.081 0.17
work_type 0.054 -0.023 0.24 0.028 0.065 0.26 1 0.032 0.052 0.26 -0.064 0.039 0.043 -0.012 -0.052
Residence_type 0.0091 0.049 -0.14 7.8e-05 0.015 -0.09 0.032 1 -0.095 -0.023 0.063 0.00021 -0.049 -0.077 -0.25
avg_glucose_level 0.011 0.01 0.32 0.16 0.15 0.17 0.052 -0.095 1 0.25 -0.095 0.075 0.41 0.43 0.31
bmi 0.025 -0.032 0.22 0.11 0.022 0.24 0.26 -0.023 0.25 1 -0.1 0.019 0.08 0.15 0.13
smoking_status 0.024 0.15 -0.27 -0.074 0.039 -0.062 -0.064 0.063 -0.095 -0.1 1 0.013 -0.0093 -0.14 -0.22
cross1 0.036 0.075 0.067 0.29 0.43 0.016 0.039 0.00021 0.075 0.019 0.013 1 0.26 0.22 -0.023
cross2 0.068 0.027 0.27 0.052 0.58 0.085 0.043 -0.049 0.41 0.08 -0.0093 0.26 1 0.19 0.27
cross3 -0.047 -0.036 0.29 0.67 0.066 0.081 -0.012 -0.077 0.43 0.15 -0.14 0.22 0.19 1 0.29
stroke 0.0003 -0.24 0.6 0.02 -0.011 0.17 -0.052 -0.25 0.31 0.13 -0.22 -0.023 0.27 0.29 1
id
gender
age
hypertension
heart_disease
ever_married
work_type
Residence_type
avg_glucose_level
bmi
smoking_status
cross1
cross2
cross3
stroke
0.8
0.6
0.4
0.2
0.0
−0.2

We can now see that the correlations are much more apparent now that we are working with a balanced dataset.

Stroke is still positively correlated with bmi (0.13), having been married (0.17), and strongly positively correlated with average glucose level (now at 0.31), and age (now at 0.6). We also see two more interesting correlations: gender (−0.24, which means that females are at a higher likelihood of having a stroke according to our data, as we encoded female as 0 and male as 1) and residence type (−0.25, which means that people living in rural areas are more likely to be hit with a heart stroke according to our data) as well as smoking_status (−0.22 correlation) which means that our choice of cardinal integer encoding was wise. Surprisingly, we lost correlation with heart_disease. ID can safely be dismissed and imputed from the data. It seems wise to keep the rest of our data.

Now, we notice that heart disease and hypertension show little correlation with having a stroke or not. We see however that when crossed with avg_glucose_level, their predictive power is almost that of avg_glucose_level alone (their correlations is close to 0.3).

2. Create a data feature extraction plan and implement a pipeline to execute it

```
# Display new correlation matrix
corr_matrix = heartstrokeBalanced.corr()
corr_matrix["stroke"].sort_values(ascending=False)
#heartstrokeBalanced.head()
```

```
stroke               1.000000
age                  0.598934
avg_glucose_level    0.309983
cross3               0.286801
cross2               0.271490
ever_married         0.173244
bmi                  0.130947
hypertension         0.019821
id                   0.000295
heart_disease       -0.010723
cross1              -0.022722
work_type           -0.051729
smoking_status      -0.215496
gender              -0.239293
Residence_type      -0.249008
Name: stroke, dtype: float64
```

As we can see, we've found 2 very interesting feature crosses, utilizing the heart_disease and hypertension features which give correlations with having a stroke when crossed with avg_glucose_level higher than the weighted sum of their parts. We can thus drop the features hypertension, id and heart_disease completely from our main dataset (since they all have weak correlations), and augment it with the crossed features found above. Since there is little correlation between work_type and stroke (likely due to our random choice of numerical values), we will simply one-hot encode that variable.

```
# Drop features
# Drop the id column altogether
heartstroke=heartstroke.drop("id", axis=1)
# Check that it was dropped correclty
heartstroke.head()
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	1	67.0	0	1	1	3	1	228.69	36.6	1	1
2	1	80.0	0	1	1	3	0	105.92	32.5	0	1
3	0	49.0	0	0	1	3	1	171.23	34.4	3	1
4	0	79.0	1	0	1	2	0	174.12	24.0	0	1
5	1	81.0	0	0	1	3	1	186.21	29.0	1	1

```
# Augment on the main dataset
heartstroke["cross1"] = heartstroke["avg_glucose_level"]*heartstroke["heart_disease"]
heartstroke["cross2"] = heartstroke["avg_glucose_level"]*heartstroke["hypertension"]
# Display first few columns
heartstroke.head()
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke	cross1	cross2
0	1	67.0	0	1	1	3	1	228.69	36.6	1	1	228.69	0.00
2	1	80.0	0	1	1	3	0	105.92	32.5	0	1	105.92	0.00
3	0	49.0	0	0	1	3	1	171.23	34.4	3	1	0.00	0.00
4	0	79.0	1	0	1	2	0	174.12	24.0	0	1	0.00	174.12
5	1	81.0	0	0	1	3	1	186.21	29.0	1	1	0.00	0.00

```
# Drop heart_disease and hypertension
heartstroke=heartstroke.drop("heart_disease", axis=1)
heartstroke=heartstroke.drop("hypertension", axis=1)
# Check
heartstroke.head()
```

	gender	age	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke	cross1	cross2
0	1	67.0	1	3	1	228.69	36.6	1	1	228.69	0.00
2	1	80.0	1	3	0	105.92	32.5	0	1	105.92	0.00
3	0	49.0	1	3	1	171.23	34.4	3	1	0.00	0.00
4	0	79.0	1	2	0	174.12	24.0	0	1	0.00	174.12
5	1	81.0	1	3	1	186.21	29.0	1	1	0.00	0.00

Now, let us implement the pipleline.

```
# remove the stroke column (which we are trying to predict with the model)
heartstroke_features=heartstroke.drop("stroke",axis=1)
heartstroke_labels=heartstroke["stroke"].copy()
heartstroke_features.head() # check that stroke is gone
```

	gender	age	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	cross1	cross2
0	1	67.0	1	3	1	228.69	36.6	1	228.69	0.00
2	1	80.0	1	3	0	105.92	32.5	0	105.92	0.00
3	0	49.0	1	3	1	171.23	34.4	3	0.00	0.00
4	0	79.0	1	2	0	174.12	24.0	0	0.00	174.12
5	1	81.0	1	3	1	186.21	29.0	1	0.00	0.00

```python
# Pipeline
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder

# Categorical features are work_type and smoking_status
# One-hot encode work_type but not smoking_status as it contains a cardinal relationship
# Gender, ever_married and Residence_type can be either one-hot encoded or left as numerical features since they can only
# take one of two values (0 or 1), thus we don't need to one-hot encode them
heartstroke_num = heartstroke_features.drop(["work_type"], axis=1)

# This will be our numerical pipeline
# Normalize using StandardScaler()
num_pipeline = Pipeline([
        ('std_scaler', StandardScaler())
    ])

numerical_features = list(heartstroke_num)
categorical_features = ["work_type"]

# Use a one-hot encoder for the categorical features
full_pipeline = ColumnTransformer([
        ("num", num_pipeline, numerical_features),
        ("cat", OneHotEncoder(), categorical_features)
    ])

heartstroke_prepared = full_pipeline.fit_transform(heartstroke_features)
heartstroke_prepared
```

```
array([[ 1.20024032,  1.06993757,  0.72927032, ...,  0.        ,
         1.        ,  0.        ],
       [ 1.20024032,  1.64633634,  0.72927032, ...,  0.        ,
         1.        ,  0.        ],
       [-0.83316648,  0.27184695,  0.72927032, ...,  0.        ,
         1.        ,  0.        ],
       ...,
       [-0.83316648, -0.34889019,  0.72927032, ...,  1.        ,
         0.        ,  0.        ],
       [ 1.20024032,  0.36052369,  0.72927032, ...,  0.        ,
         1.        ,  0.        ],
       [-0.83316648,  0.05015511,  0.72927032, ...,  0.        ,
         0.        ,  1.        ]])
```

Finally, create training and test sets on balanced data. Use 80/20 split.

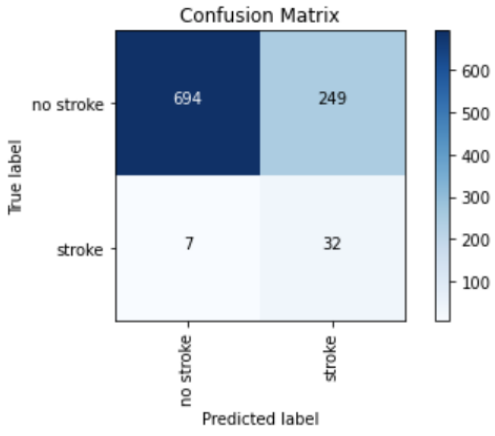
```
# Setting up testing and training sets
train_features, test_features, train_labels, test_labels = train_test_split(
                                            heartstroke_prepared,
                                            heartstroke_labels,
                                            test_size=0.2,
                                            random_state=27)

# Resample the minority class
sm = SMOTE(random_state=27, sampling_strategy='minority')

# Balance the training set only
train_features, train_labels = sm.fit_resample(train_features, train_labels)
```

3. Run a Logistic Regression Model

```
# Logistic Regression
logRegPreds=LogisticRegression().fit(train_features, train_labels).predict(test_features)
```

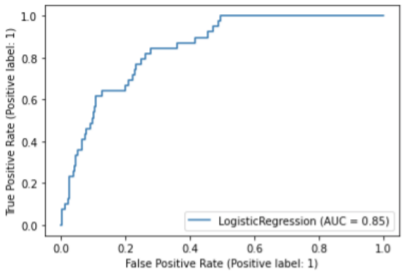
```
# Report 4 metrics
print("Accuracy:\t", metrics.accuracy_score(test_labels, logRegPreds))
print("Precision:\t", metrics.precision_score(test_labels, logRegPreds))
print("Recall:\t\t", metrics.recall_score(test_labels, logRegPreds))
print("F1 Score:\t", metrics.f1_score(test_labels, logRegPreds))
# Plot confusion matrix
draw_confusion_matrix(test_labels, logRegPreds, ["no stroke", "stroke"])
```

```
Accuracy:       0.7393075356415478
Precision:      0.11387900355871886
Recall:         0.8205128205128205
F1 Score:       0.2
```



```
metrics.plot_roc_curve(LogisticRegression().fit(train_features,
                                                train_labels), test_features, test_labels)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x126af5b20>
```

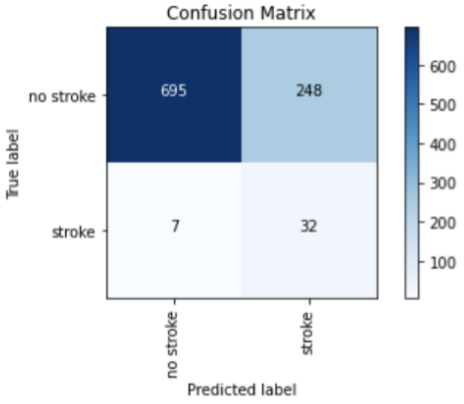


By running my first logistic regression model, I find that my model gives high accuracy and recall scores, and relatively low precision and F1 scores. What this means is that my first logistic regression model is good at correctly identifying many positives present in the actual datase (high recall), but that it doesn't perform well at not mislabeling negatives as a positives (low precision). 32 strokes out of 39 were predicted correctly, but 249 healthy patients were predicted incorrectly to have a stroke. Usually, in fields like medicine, a high recall score is preferrable over a low one, because it is critical to identify as many of the positive cases as possible, even if that means negative cases will be incorrectly classified as being positive, since true positives need immediate medical care and false positives can undergo fuirther testing to determine that they were actually healthy. Nonetheless, let's attempt to improve on our logisitic regression model.

```
# Logistic Regression modified with L1  Regularization
logRegPreds=LogisticRegression(solver="liblinear", penalty="l1").fit(train_features, train_labels).predict(test_features)
```

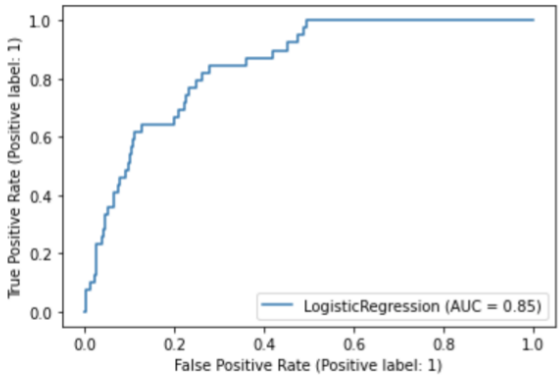
```
# Report 4 metrics
print("Accuracy:\t", metrics.accuracy_score(test_labels, logRegPreds))
print("Precision:\t", metrics.precision_score(test_labels, logRegPreds))
print("Recall:\t\t", metrics.recall_score(test_labels, logRegPreds))
print("F1 Score:\t", metrics.f1_score(test_labels, logRegPreds))
# Plot confusion matrix
draw_confusion_matrix(test_labels, logRegPreds, ["no stroke", "stroke"])
```

```
Accuracy:         0.7403258655804481
Precision:        0.11428571428571428
Recall:           0.8205128205128205
F1 Score:         0.2006269592476489
```




```
metrics.plot_roc_curve(LogisticRegression(solver="liblinear", penalty="l1").fit(train_features,
                                                  train_labels), test_features, test_labels)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x126565fa0>
```



Our model did not improve from using L1 or L2 regularization, nor from using the sag solver. Let us thus move onto the next model.

4. Implement Principle Component Analysis (PCA)

Let's first reduce the dimensionality of our problem.

```
from sklearn.decomposition import PCA

# Implement PCA on standardized data (on heartstroke_prepared)
heartstroke_pca=heartstroke_prepared

# We have currently 10 features. Let's choose the number of components necessary to retain 90% of the variance
pca = PCA(0.9)

principalComponents = pca.fit_transform(heartstroke_pca)

# It turns out we will need 8 principal components
heartstroke_pca = pd.DataFrame(data = principalComponents,
                               columns = ['pc1', 'pc2', 'pc3', 'pc4', 'pc5', 'pc6', 'pc7', 'pc8'])

# Print first few columns of dataset
heartstroke_pca.head()
```

	pc1	pc2	pc3	pc4	pc5	pc6	pc7	pc8
0	3.866884	4.220221	-0.981397	0.171600	0.740736	-3.806074	3.259292	0.291216
1	2.283763	0.884100	0.586161	-1.338283	-0.335083	-2.561618	0.980061	0.866738
2	0.878777	0.152974	-1.339958	0.418035	1.423048	1.020784	0.525651	-1.249726
3	2.941694	1.059676	2.099047	1.400971	-0.062669	0.975800	-2.671236	0.337873
4	1.781526	0.532277	-1.090521	-0.226099	-0.948890	-0.138875	-0.153028	-1.729764

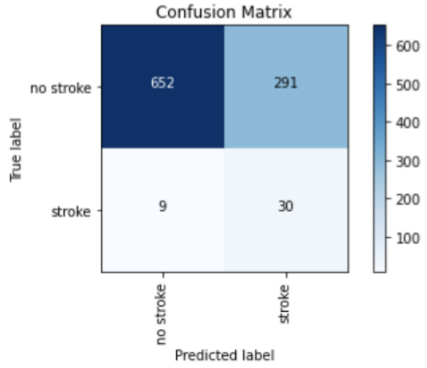
```
# Now let us split our new dataset into train and test cohorts again
train_features_pca, test_features_pca, train_labels_pca, test_labels_pca = train_test_split(
                                            heartstroke_pca,
                                            heartstroke_labels,
                                            test_size=0.2,
                                            random_state=27)
```

```
# Now let us balance the datasets once again
# Resample the minority class
sm = SMOTE(random_state=27, sampling_strategy='minority')

# Balance the training set only
train_features_pca, train_labels_pca = sm.fit_resample(train_features_pca, train_labels_pca)
```

```
# Now test the logistic regression algorithm again
logRegPreds=LogisticRegression(solver="liblinear", penalty="l1").fit(train_features_pca, train_labels_pca).predict(test_features_pca)
```

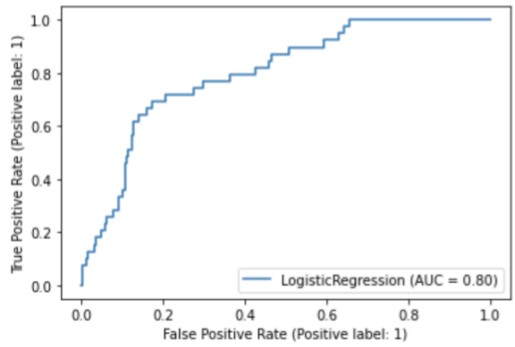
```
# Report 4 metrics
print("Accuracy:\t", metrics.accuracy_score(test_labels_pca, logRegPreds))
print("Precision:\t", metrics.precision_score(test_labels_pca, logRegPreds))
print("Recall:\t\t", metrics.recall_score(test_labels_pca, logRegPreds))
print("F1 Score:\t", metrics.f1_score(test_labels_pca, logRegPreds))
# Plot confusion matrix
draw_confusion_matrix(test_labels_pca, logRegPreds, ["no stroke", "stroke"])
```

```
Accuracy:       0.6945010183299389
Precision:      0.09345794392523364
Recall:         0.7692307692307693
F1 Score:       0.16666666666666666
```




```
metrics.plot_roc_curve(LogisticRegression().fit(train_features_pca,
                                                train_labels_pca), test_features_pca, test_labels_pca)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1264f0e20>
```



As one can see, our model performs slightly worse under PCA. This is due to the fact that some data is removed through dimensionality reduction.

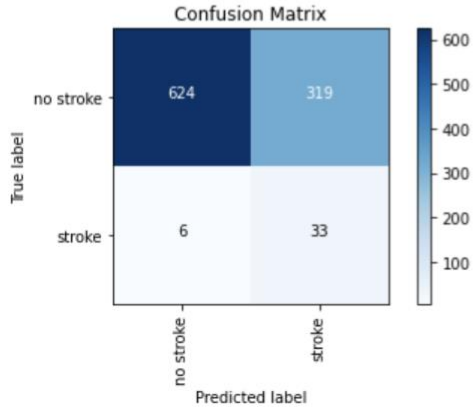
5. Employ an ensemble method

```
from sklearn import tree
from sklearn.model_selection import cross_val_score
from sklearn.utils import resample
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
```

```
# Use the Boosting algorithm (Adaboost)

# max_depth is the depth of each tree, decrease it to reduce variance and increase bias
# n_estimators is the number of trees used, the more trees the more we overfit

adaboost=AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=1), n_estimators=25, learning_rate=3)
ensembleFit=adaboost.fit(train_features, train_labels)
ensemblePreds=ensembleFit.predict(test_features)
```

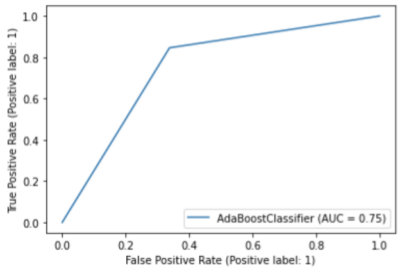
```
# Display statistics from our model
# Report 4 metrics
print("Accuracy:\t", metrics.accuracy_score(test_labels, ensemblePreds))
print("Precision:\t", metrics.precision_score(test_labels, ensemblePreds))
print("Recall:\t\t", metrics.recall_score(test_labels, ensemblePreds))
print("F1 Score:\t", metrics.f1_score(test_labels, ensemblePreds))
# Plot confusion matrix
draw_confusion_matrix(test_labels, ensemblePreds, ["no stroke", "stroke"])
```

```
Accuracy:       0.6690427698574338
Precision:      0.09375
Recall:         0.8461538461538461
F1 Score:       0.16879795396419436
```



```
metrics.plot_roc_curve(ensembleFit, test_features, test_labels)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x126df6ca0>
```

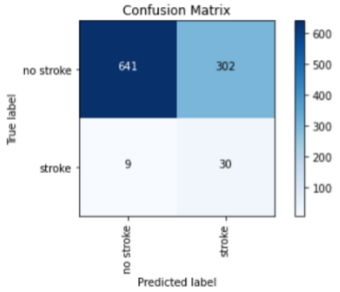


We are able to, through the adaboost ensemble method, increase our recall score from the logistic regression result by using a low number of trees and low maximum depth for each tree, as well as a learning rate of 3. Our recall is 84% this time.

```
# Random Forests
ntrees=35
maxf=0.5
randFor=RandomForestClassifier(oob_score=True, n_estimators=ntrees, max_features=maxf, max_depth=1, n_jobs=-1)
randForFit=randFor.fit(train_features, train_labels)
randForPreds=randForFit.predict(test_features)
```

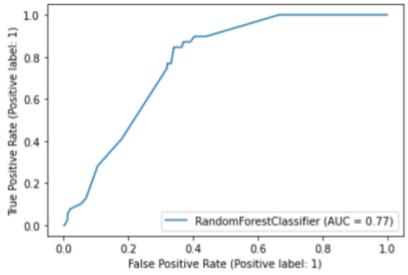
```
# Display statistics from our model
# Report 4 metrics
print("Accuracy:\t", metrics.accuracy_score(test_labels, randForPreds))
print("Precision:\t", metrics.precision_score(test_labels, randForPreds))
print("Recall:\t\t", metrics.recall_score(test_labels, randForPreds))
print("F1 Score:\t", metrics.f1_score(test_labels, randForPreds))
# Plot confusion matrix
draw_confusion_matrix(test_labels, randForPreds, ["no stroke", "stroke"])
```

```
Accuracy:         0.6832993890020367
Precision:        0.09036144578313253
Recall:           0.7692307692307693
F1 Score:         0.16172506738544473
```




```
metrics.plot_roc_curve(randForFit, test_features, test_labels)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x128ad1df0>
```



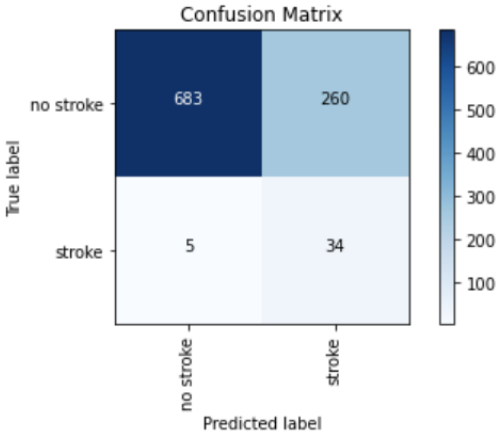
We obtain once again similar results by changing the number of trees and max_features parameter (maximum percentage of features for each tree to randomly split on) for the random forest classifier. 84% seems to be the best recall score attainable using these ensemble methods.

6. Develop a Neural Network classifier

```
from sklearn.neural_network import MLPClassifier

# Our multilayer perceptron classifier
# max_iter = maximum number of iterations to reach convergence
# learning_rate = learning rate type of our model
# alpha = L2 regularization penalty scalar
# activation = activation function
# solver = which solver to perform optimization
# hidden_layer_sizes = sets # of hidden layers and number of nodes in them
# batch_size = Batch size is how many training examples you use at each iteration of training
neuralNetFit=MLPClassifier(random_state=27, max_iter=500,
                           learning_rate="adaptive", learning_rate_init=0.1, alpha=0.08,
                           activation="tanh", solver="adam", beta_1=0.1, beta_2=0.2,
                           batch_size=100, early_stopping=True,
                           hidden_layer_sizes=(10, 10, 10, 10)).fit(train_features, train_labels)
neuralNetPreds=neuralNetFit.predict(test_features)
```

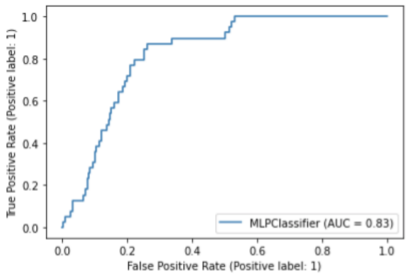
```
# Display statistics from our model
# Report 4 metrics
print("Accuracy:\t", metrics.accuracy_score(test_labels, neuralNetPreds))
print("Precision:\t", metrics.precision_score(test_labels, neuralNetPreds))
print("Recall:\t\t", metrics.recall_score(test_labels, neuralNetPreds))
print("F1 Score:\t", metrics.f1_score(test_labels, neuralNetPreds))
# Plot confusion matrix
draw_confusion_matrix(test_labels, neuralNetPreds, ["no stroke", "stroke"])
```

```
Accuracy:       0.730142566191446
Precision:      0.11564625850340136
Recall:         0.8717948717948718
F1 Score:       0.2042042042042042
```




```
metrics.plot_roc_curve(neuralNetFit, test_features, test_labels)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1263e1ee0>
```



After trying out a combination of parameters, we found that our data performs best on a multilayer perceptron classifier with the following parameters:

• Adaptive learning rate which means that the model keeps the learning rate constant to our chosen value of 0.1 as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5

• Activation function is tanh

• There are 4 hidden layers made up each of 10 nodes

• The solver is "adam", which is a stochastic gradient-based optimizer, with exponential decay rates for first and second moment vectors of 0.1 and 0.2 respectively

• We're using a batch size of 100, meaning we use 100 training samples at each iteration of model training

• max_iter=500: We are running the solver for a maximum of 500 epochs to reach convergence

• We use an L2 regularization penalty term of 0.08

• We are using early_stopping to terminate training if the validation score is not improving

The result is that our model gives high accuracy and recall scores, of 73% and 87% respectively. Precision and F1 scores are still low at 11% and 20%. It seems that even after balancing our data prevents our predictive models have a high rate of false positives.

7. Cross-Validate Ensemble Method and Neural Network Classifer

```
# 10-fold cross valication for random forests
from sklearn import model_selection
from sklearn.model_selection import StratifiedShuffleSplit

# Balance the dataset
heartstroke_features_balanced, heartstroke_labels_balanced=sm.fit_resample(heartstroke_prepared, heartstroke_labels)

ntrees=35
maxf=0.5

model=RandomForestClassifier(random_state=27, oob_score=True, n_estimators=ntrees,
                             max_features=maxf, max_depth=1, n_jobs=-1)
cv=model_selection.KFold(n_splits=10, random_state=27, shuffle=True)
kfolds_accuracies=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="accuracy")
kfolds_precisions=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="precision")
kfolds_recalls=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="recall")
kfolds_f1_scores=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="f1")
mean_accuracy=100*kfolds_accuracies.mean()
mean_precision=100*kfolds_precisions.mean()
mean_recall=100*kfolds_recalls.mean()
mean_f1_score=100*kfolds_f1_scores.mean()
print("10-Fold Cross Validation for Random Forests: ")
print("Accuracy:\t", mean_accuracy)
print("Precision:\t", mean_precision)
print("Recall:\t\t", mean_recall)
print("F1 Score:\t", mean_f1_score)
```

```
10-Fold Cross Validation for Random Forests:
Accuracy:       78.6019418575669
Precision:      73.97216429450818
Recall:         88.27784391110754
F1 Score:       80.47546497512046
```

```
# 10-fold cross valication for neural network
model=MLPClassifier(random_state=27, max_iter=500,
                        learning_rate="adaptive", learning_rate_init=0.1, alpha=0.08,
                        activation="tanh", solver="adam", beta_1=0.1, beta_2=0.2,
                        batch_size=100, early_stopping=True,
                        hidden_layer_sizes=(10, 10, 10, 10))

cv=model_selection.KFold(n_splits=10, random_state=27, shuffle=True)
kfolds_accuracies=model_selection.cross_val_score(model,
                                            heartstroke_features_balanced,
                                            heartstroke_labels_balanced,
                                            cv=cv,
                                            scoring="accuracy")
kfolds_precisions=model_selection.cross_val_score(model,
                                            heartstroke_features_balanced,
                                            heartstroke_labels_balanced,
                                            cv=cv,
                                            scoring="precision")
kfolds_recalls=model_selection.cross_val_score(model,
                                            heartstroke_features_balanced,
                                            heartstroke_labels_balanced,
                                            cv=cv,
                                            scoring="recall")
kfolds_f1_scores=model_selection.cross_val_score(model,
                                            heartstroke_features_balanced,
                                            heartstroke_labels_balanced,
                                            cv=cv,
                                            scoring="f1")
mean_accuracy=100*kfolds_accuracies.mean()
mean_precision=100*kfolds_precisions.mean()
mean_recall=100*kfolds_recalls.mean()
mean_f1_score=100*kfolds_f1_scores.mean()
print("10-Fold Cross Validation for Neural Network: ")
print("Accuracy:\t", mean_accuracy)
print("Precision:\t", mean_precision)
print("Recall:\t\t", mean_recall)
print("F1 Score:\t", mean_f1_score)
```

```
10-Fold Cross Validation for Neural Network:
Accuracy:       79.63426460924931
Precision:      75.61200085016125
Recall:         87.96358027478274
F1 Score:       81.19729812091649
```

From our 10-fold cross validation, we see that overall, the neural network model performs better, with higher accuracy, precision and F1 score. We should nonetheless note that the k-fold cross validation method used is imperfect because it doesn't let us test our model on unbalanced data as we did throughout this project (each fold is trained and tested on balanced data which will inevitably lead to higher scores overall). This means that the high scores we obtain from 10-fold cross validation are not representative of our models, and one should keep in mind that our model performs worse on precision and f1 score if given a test cohort with the same unbalanced distribution as is found in the raw data. Nonetheless, we are able to use 10-fold cross-validation to assert that our neural network model likely would perform better than our ensemble (random forests) method, even on unbalanced data.

8. Experiment with your own custom models

8.1 k-Nearest Neighbors

```
# Try the algorithm for several values of k
nList=[1,2,3,5,7,9,10,20,50,60,65,70,75,100,200,300,500]
for n in nList:
    kNNpreds=KNeighborsClassifier(n_neighbors=n).fit(train_features,
                                                     train_labels).predict(test_features)
    print("Accuracy, precision, recall, F1 score for k =", n, ":\t",
          "{:.2f}".format(metrics.accuracy_score(test_labels, kNNpreds)),
          "{:.2f}".format(metrics.precision_score(test_labels, kNNpreds)),
          "{:.2f}".format(metrics.recall_score(test_labels, kNNpreds)),
          "{:.2f}".format(metrics.f1_score(test_labels, kNNpreds)))
```

```
Accuracy, precision, recall, F1 score for k = 1 :        0.90 0.10 0.21 0.13
Accuracy, precision, recall, F1 score for k = 2 :        0.91 0.11 0.18 0.14
Accuracy, precision, recall, F1 score for k = 3 :        0.87 0.11 0.31 0.16
Accuracy, precision, recall, F1 score for k = 5 :        0.84 0.09 0.36 0.15
Accuracy, precision, recall, F1 score for k = 7 :        0.81 0.09 0.41 0.15
Accuracy, precision, recall, F1 score for k = 9 :        0.79 0.10 0.51 0.16
Accuracy, precision, recall, F1 score for k = 10 :       0.81 0.10 0.51 0.17
Accuracy, precision, recall, F1 score for k = 20 :       0.74 0.09 0.62 0.16
Accuracy, precision, recall, F1 score for k = 50 :       0.69 0.09 0.69 0.15
Accuracy, precision, recall, F1 score for k = 60 :       0.69 0.09 0.72 0.16
Accuracy, precision, recall, F1 score for k = 65 :       0.68 0.09 0.74 0.16
Accuracy, precision, recall, F1 score for k = 70 :       0.68 0.09 0.77 0.16
Accuracy, precision, recall, F1 score for k = 75 :       0.66 0.08 0.74 0.15
Accuracy, precision, recall, F1 score for k = 100 :      0.65 0.08 0.74 0.14
Accuracy, precision, recall, F1 score for k = 200 :      0.63 0.08 0.82 0.15
Accuracy, precision, recall, F1 score for k = 300 :      0.62 0.08 0.85 0.15
Accuracy, precision, recall, F1 score for k = 500 :      0.63 0.09 0.85 0.15
```

From our nearest neighbor model, we see that the precision and f1 scores are again pretty low and that accuracy is at the highest for smaller values of k and decreases as k increases, whereas recall increase as k increases and is large for large values of k. Our model thus performs best overall at around k=70 where we have an accuracy of 68% and a recall of 77%.

8.2 Support Vector Machine

```
# Try several SVM algorithms, linear and kernelized

# SVM for linear, rbf and precomputed kernels
for kernelType in ["linear", "rbf", "sigmoid"]:
    svmFit=SVC(probability=True, kernel=kernelType, coef0=0.5, gamma="scale").fit(train_features, train_labels)
    svmPreds=svmFit.predict(test_features)
    print("Accuracy, precision, recall, F1 score for kernel =", kernelType, ":\t\t",
          "{:.2f}".format(metrics.accuracy_score(test_labels, svmPreds)),
          "{:.2f}".format(metrics.precision_score(test_labels, svmPreds)),
          "{:.2f}".format(metrics.recall_score(test_labels, svmPreds)),
          "{:.2f}".format(metrics.f1_score(test_labels, svmPreds)))

# SVM for polynomial kernel
for num_degrees in [2,3,4]:
    svmFit=SVC(probability=True, kernel="poly", degree=num_degrees, gamma="scale").fit(train_features, train_labels)
    svmPreds=svmFit.predict(test_features)
    print("Accuracy, precision, recall, F1 score for kernel = poly of degree", num_degrees, ":\t",
          "{:.2f}".format(metrics.accuracy_score(test_labels, svmPreds)),
          "{:.2f}".format(metrics.precision_score(test_labels, svmPreds)),
          "{:.2f}".format(metrics.recall_score(test_labels, svmPreds)),
          "{:.2f}".format(metrics.f1_score(test_labels, svmPreds)))
```

```
Accuracy, precision, recall, F1 score for kernel = linear :              0.72 0.11 0.85 0.19
Accuracy, precision, recall, F1 score for kernel = rbf :                 0.79 0.11 0.62 0.19
Accuracy, precision, recall, F1 score for kernel = sigmoid :             0.69 0.08 0.64 0.14
Accuracy, precision, recall, F1 score for kernel = poly of degree 2 :    0.76 0.11 0.69 0.19
Accuracy, precision, recall, F1 score for kernel = poly of degree 3 :    0.79 0.11 0.56 0.18
Accuracy, precision, recall, F1 score for kernel = poly of degree 4 :    0.84 0.12 0.51 0.20
```

We see that our support vector machine model performs best for a linear kernel. Indeed it returns a 72% accuracy and an 85% recall. This means that our linear kernel SVM model performs better than our 70-nearest neighbor predictive model.

8.3 Naive Bayes

```
# Implement Naive Bayes
from sklearn.naive_bayes import GaussianNB

for vsmooth in [1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 0.5, 0.9, 1]:
    gnb=GaussianNB(var_smoothing=vsmooth)
    nbFit=gnb.fit(train_features, train_labels)
    nbPreds=nbFit.predict(test_features)
    print("Accuracy, precision, recall, F1 score for variance smoothing =", vsmooth, ":\t",
          "{:.2f}".format(metrics.accuracy_score(test_labels, svmPreds)),
          "{:.2f}".format(metrics.precision_score(test_labels, svmPreds)),
          "{:.2f}".format(metrics.recall_score(test_labels, svmPreds)),
          "{:.2f}".format(metrics.f1_score(test_labels, svmPreds)))
```

```
Accuracy, precision, recall, F1 score for variance smoothing = 1e-09 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 1e-08 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 1e-07 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 1e-06 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 1e-05 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.0001 :  0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.001 :   0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.01 :    0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.1 :     0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.5 :     0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 0.9 :     0.84 0.12 0.51 0.20
Accuracy, precision, recall, F1 score for variance smoothing = 1 :       0.84 0.12 0.51 0.20
```

The Naive Bayes algorithm seems to perform worse compared to our two other models in terms of recall score, with 84% accuracy and 51% recall.

8.4 Perform 10-fold cross validation to see which model is the best

```
# Cross validate for 70-nearest neighbor model
model=KNeighborsClassifier(n_neighbors=70)
cv=model_selection.KFold(n_splits=10, random_state=27, shuffle=True)
kfolds_accuracies=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="accuracy")
kfolds_precisions=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="precision")
kfolds_recalls=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="recall")
kfolds_f1_scores=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="f1")
mean_accuracy=100*kfolds_accuracies.mean()
mean_precision=100*kfolds_precisions.mean()
mean_recall=100*kfolds_recalls.mean()
mean_f1_score=100*kfolds_f1_scores.mean()
print("10-Fold Cross Validation for Nearest Neighbor: ")
print("Accuracy:\t", mean_accuracy)
print("Precision:\t", mean_precision)
print("Recall:\t\t", mean_recall)
print("F1 Score:\t", mean_f1_score)
```

```
10-Fold Cross Validation for Nearest Neighbor:
Accuracy:       81.27284571635738
Precision:      74.03578814303157
Recall:         96.35469999787365
F1 Score:       83.7185585855287
```

```
# Cross validate for SVM
model=SVC(probability=True, kernel="linear", coef0=0.5, gamma="scale")
cv=model_selection.KFold(n_splits=10, random_state=27, shuffle=True)
kfolds_accuracies=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="accuracy")
kfolds_precisions=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="precision")
kfolds_recalls=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="recall")
kfolds_f1_scores=model_selection.cross_val_score(model,
                                                  heartstroke_features_balanced,
                                                  heartstroke_labels_balanced,
                                                  cv=cv,
                                                  scoring="f1")
mean_accuracy=100*kfolds_accuracies.mean()
mean_precision=100*kfolds_precisions.mean()
mean_recall=100*kfolds_recalls.mean()
mean_f1_score=100*kfolds_f1_scores.mean()
print("10-Fold Cross Validation for SVM: ")
print("Accuracy:\t", mean_accuracy)
print("Precision:\t", mean_precision)
print("Recall:\t\t", mean_recall)
print("F1 Score:\t", mean_f1_score)
```

```
10-Fold Cross Validation for SVM:
Accuracy:       79.15524664083566
Precision:      75.67974447500406
Recall:         85.93942809028091
F1 Score:       80.46704011530686
```

Our nearest neighbor algorithm seems to perform best on fully balanced data, whereas the support vector machine algorithm performed better on test data that's not synthetically balanced, as we saw previously. We stress the importance of that distinction, as the eventual distribution of data that one's model is tested on can thus interfere with the model's performance metrics to a great degree.