BIG DATA APPLICATIONS

Stroke Prediction Using Machine Learning

Brian Collins

Word Count: 4378

Introduction

The purpose of this report is to analyse the Stroke Prediction Dataset as found on Kaggle, firstly describing the data contained therein alongside descriptive statistics of the underlying variables before conducting a correlation analysis. It will then detail several models trained on the described dataset, with the direct purpose of facilitating the accurate prediction of stroke risk based on other overall markers of health and personal factors.

Dataset Overview

Gender – listing gender groups into three distinct categories. Male, Female and Other. Of 5,110 observations in the dataset, one is listed as 'Other'. For the purposes of this work, this single value will be removed from the dataset to prevent any interference on model accuracy. The data set indicates a slight preference favouring female patients, with the latter representing 59% of the dataset.

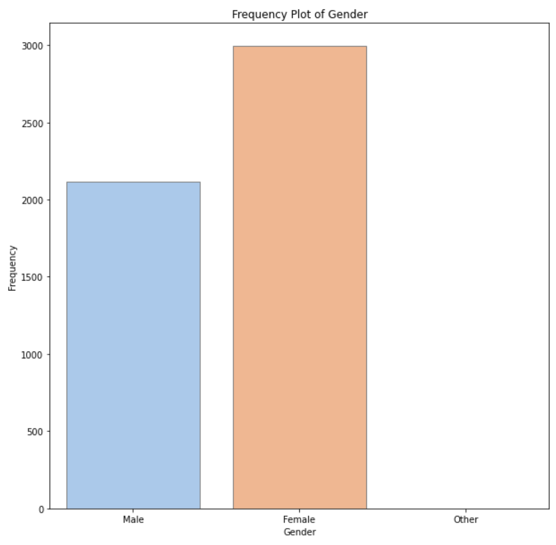


Figure 1: Frequency of genders within the dataset

Age – listing age of the patient in years, with continuous values ranging between 0.08 and 82. Values between both Male and Females follow a broadly similar distribution, with the variation found in frequency as per above.

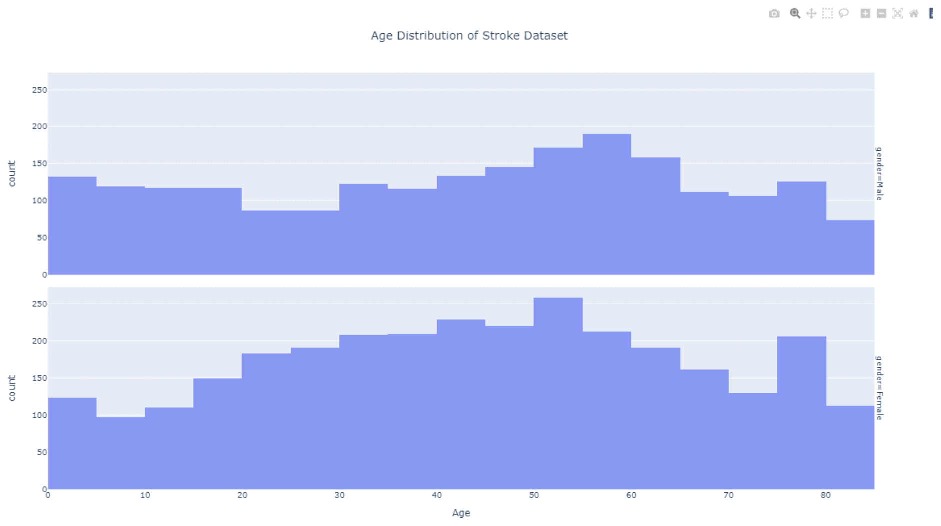


Figure 2: Histogram of patient age separated by Gender

To further examine the impact of age on stroke values, the below graph compares the raw frequency of stroke values by patient age, indicating a strong tendency for strokes occurring as patients reported age increases.

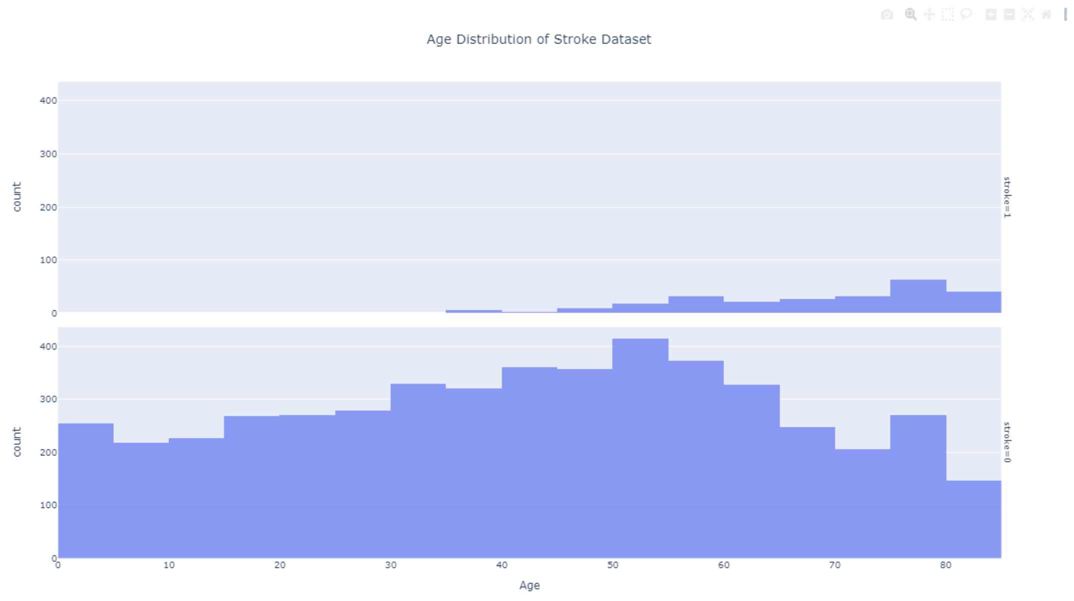


Figure 3: Histogram displaying incidence of stroke by patient age

Hypertension – A binary value indicating whether a given patient showed signs of hypertension (high blood pressure). One indicates the presence of high blood pressure, 0 indicates normal blood pressure. Of those with Hypertension, 13.25% suffered a stroke, compared to 3.97% of healthy patients, suggesting hypertension presents an increased risk factor for stroke in patients.

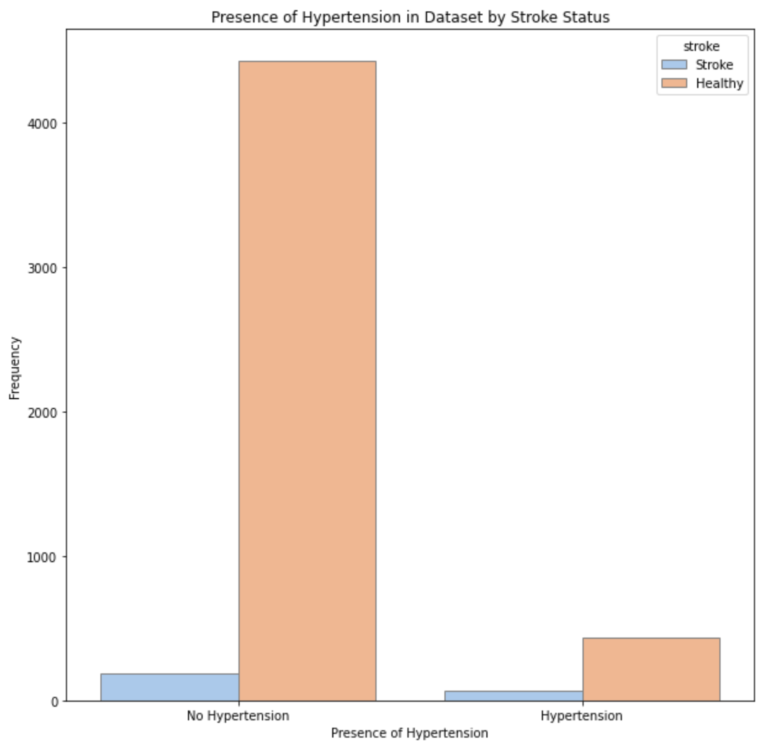


Figure 4: Count plot of patient hypertension status by stroke incidence

Heart Disease – Similar to Hypertension, presents a binary value indicating whether a given patient carried the markers of heart disease at the point of the data record. 17.02% of patients with heart disease suffered a stroke, compared to 4.18% in patients without indication of heart disease, again suggesting that heart disease presents an increased risk factor for stroke in patients.

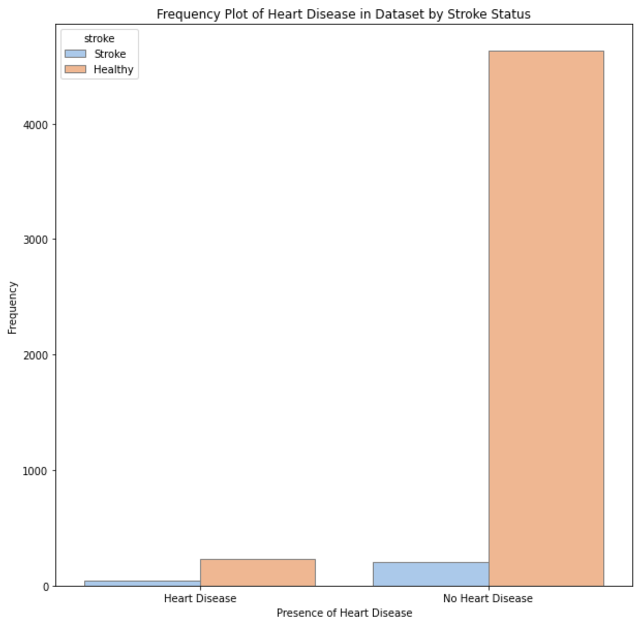


Figure 5: Count plot of heart disease status by stroke incidence

Ever Married – A binary value indicating whether a given patient in the dataset had ever been married. Indicating married patients suffer strokes at a higher incidence than their unmarried counterparts, with 6.56% of patients who had previously married their partner suffering a stroke, compared to 1.65% in patients who were previously unmarried.

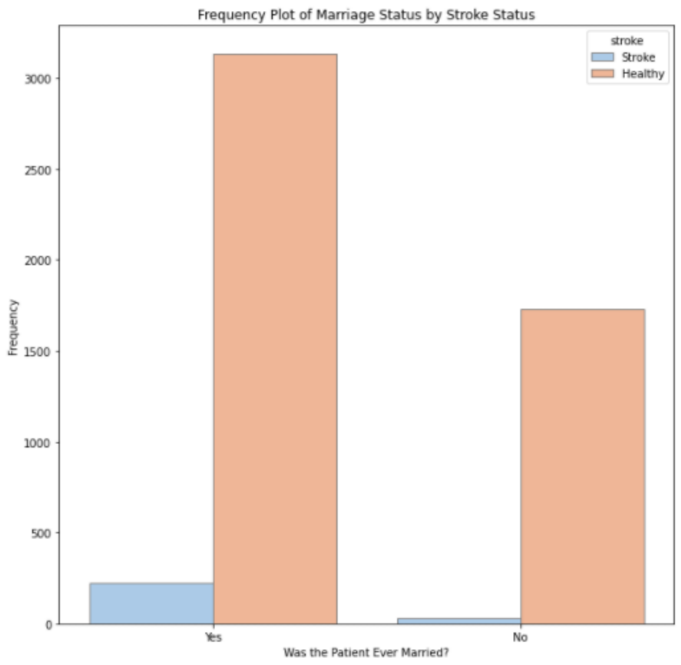


Figure 6: Count plot of patient marriage status by stroke incidence

Work Type – Presenting a categorical variable that indicated the patients reported employment type, categorised into three formal occupations (private sector, governmental (public) sector and self-employed. Patients identified as primary child carers and those without formal employment histories are represented. When clustering private, self-employed and government roles together as "Employed," it is interesting to note that 5.61% of patients suffered a stroke, compared to 0.28% of patients listed as unemployed (children and Never_worked).

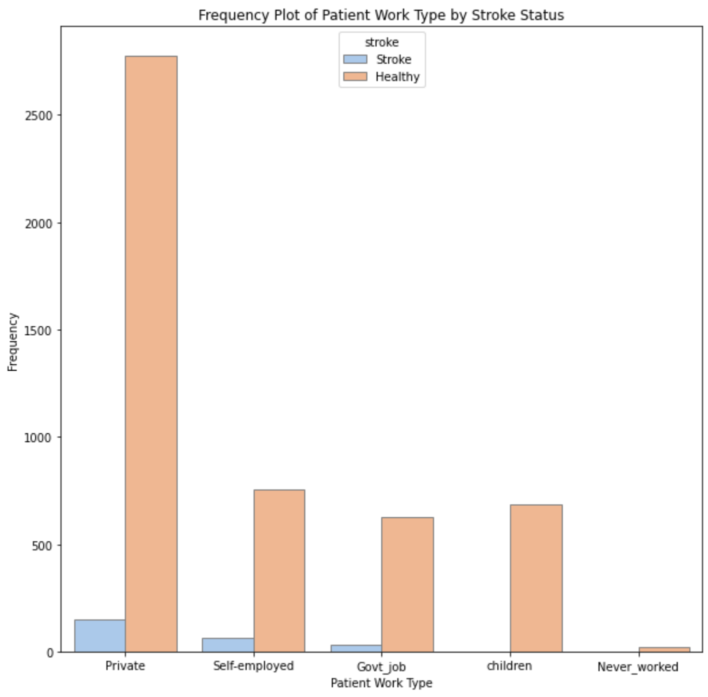


Figure 7: Count plot of patient employment status by stroke incidence

Residence Type – A binary variable indicating whether the patient lives in a Rural or Urban environment. 5.2% of patients living in an urban environment suffered a stroke, compared to 4.54% of patients living in a rural environment, suggesting very little difference in stroke incidence based on residence location.

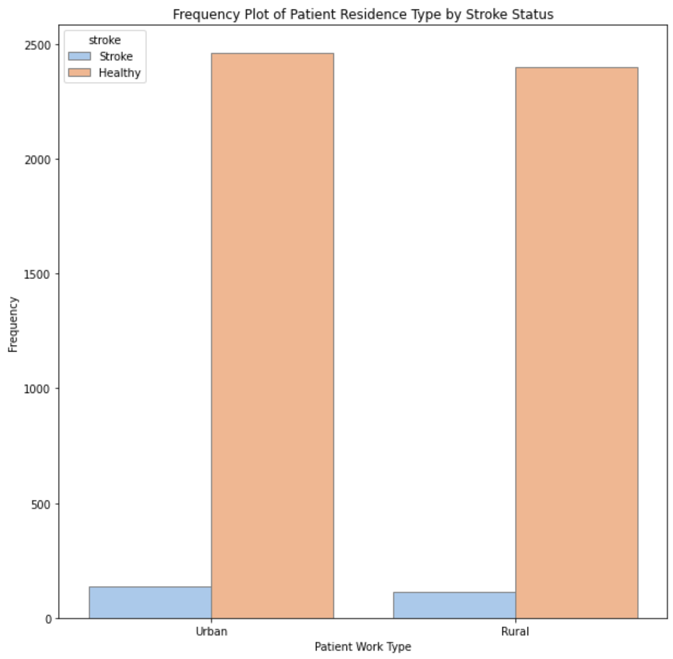


Figure 8: Count plot of patient location by stroke incidence

Smoking Status – Presenting the smoking history of a given patient, listing current smokers, lapsed smokers, no smoking history or unknown as a categorical variable. 4.76% of patients with no history of smoking suffered stokes, compared to 6.69% of patients who have smoking in their history, or 5.32% in the case of those actively smoking. This variation suggests no apparent relationship between smoking history and stroke risk.

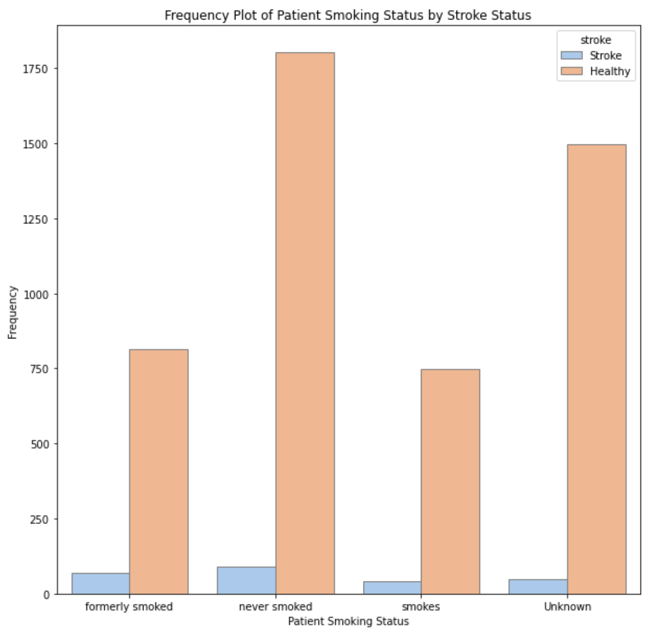


Figure 9: Count plot of smoking history by stroke incidence

Average Glucose Level – Continuous variable Indicating patient average glucose level at the recording time. Initial comparison of the below chart suggests a far higher interquartile range indicating the potential for a positive relationship between glucose levels and stroke risk. While it is important to note that comparatively healthy patients far outweigh the number of patients who have suffered a stroke, the smaller number of patients results in far more variability. Average glucose ranges are almost identical for both groups, but the longstanding and well-documented relationship between high blood sugar and stroke risk (Chen et al., 2016) also infers a positive relationship between the variables.

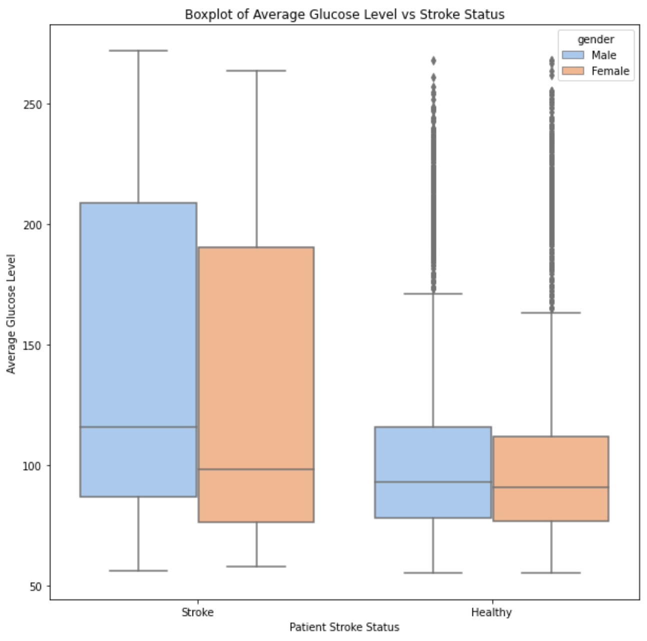


Figure 10: Boxplot of average glucose level by Gender and stroke incidence

BMI – Continuous variable providing insight into patients' body mass index, indicating no significant impact on stroke risk. Of the total number of observations in the dataset, 201 values are NULL, representing 3.9%. As a result, BMI is the only variable within the dataset with missing values. In order to account for this when conducting analysis, these missing values will be dropped, replaced with the column mean and imputed, to assess the impact on overall model accuracy.

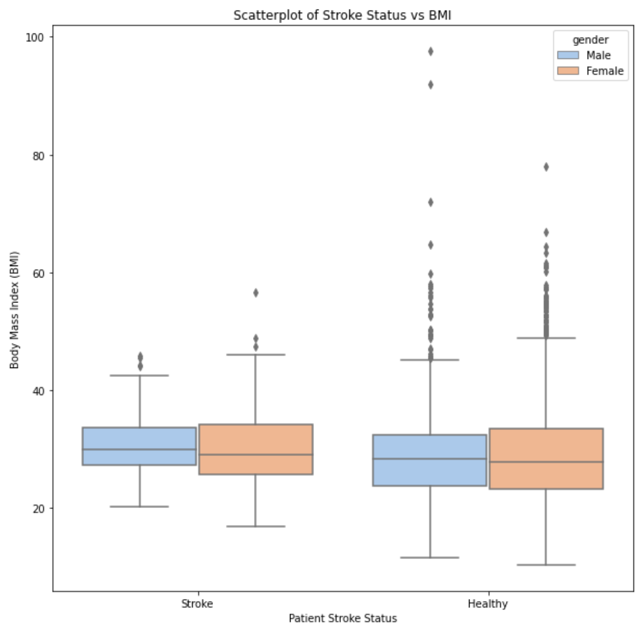


Figure 11: Boxplot displaying BMI values by Gender and stroke incidence

To further support the above assertations, refer to the below scatter comparing average glucose levels against BMI. Stroke values are not weighted towards patients with either low BMI or low glucose levels, or any mix of both, indicating a weak relationship if one exists.

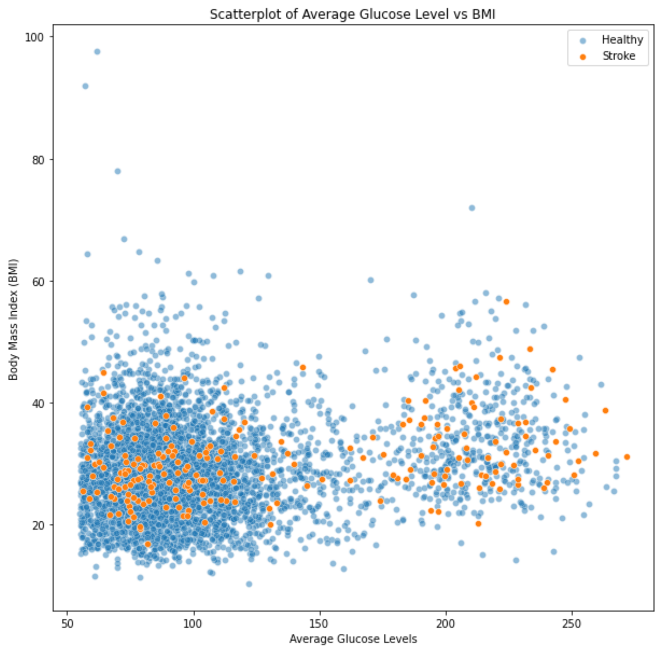


Figure 12: Scatterplot of average glucose level against patient BMI, marker colour indicating stroke status

Stroke – The binary target variable for the dataset, indicating whether a given patient has suffered a stroke. Values are heavily weighted in favour of healthy patients, meaning that some allowances will need to be made in order to be able to train accurate models that can be generalised on data from outside the training and test set.

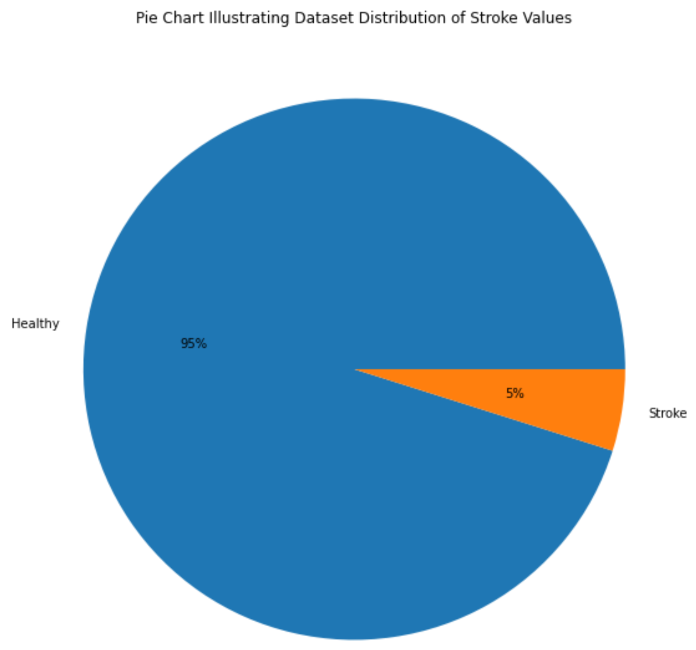


Figure 13: Pie chart indicating stroke incidence within the dataset

Variable Correlation

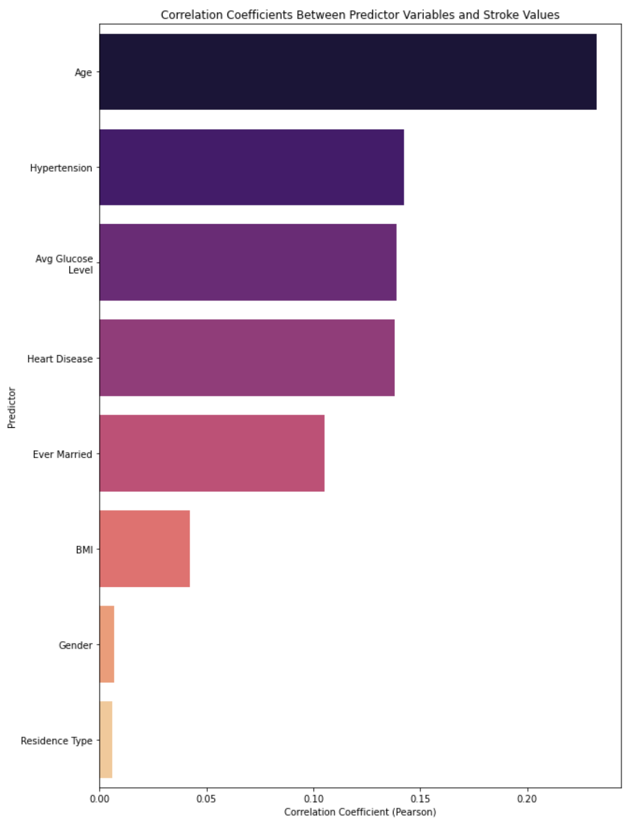
As part of the data preparation process to be detailed later, continuous variables (BMI, Average Glucose Level and Age) were all normalised using SciKit-Learn's MinMaxScaler method. These three variables were then run through a Shapiro-Wilk test using SciPy's Stats package to test the null hypothesis that the data has been drawn from a population with a normal distribution. Results of the three tests can be seen below:

Variable	Statistic	P-Value
Age	0.9682105779647827	1.258497064878765e-31
BMI	0.9535498023033142	6.63012866537492e-37
Average Glucose Level	0.8058329820632935	0.0*[1]

Table 1: Shapiro-Wilk test results for continuous variables within the dataset

In the case of all three variables, the p-value was below the cut-off of 0.05, enabling the rejection of the null hypothesis. As such, we can proceed with a Pearson correlation coefficient, having removed any categorical variables in the dataset with >2 potential categories. The correlation was then established using Numpy's 'corrcoef' function, which produced the following results:

[1] It is presumed that the Shaprio-Wilk p-value in this case is not actually 0.0, rather the number of decimal places is so large it is rounded by SciPy to present the observed figure.



Predictor	Correlation Coefficient
Age	0.232331
Hypertension	0.142515
Average Glucose Level	0.138936
Heart Disease	0.137938
Ever Married	0.105089
BMI	0.042374
Gender	0.006757
Residence Type	0.006031

Table 2: Pearson correlation coefficients for all predictor variables

It should also be noted that a level of multicollinearity exists between variables in the dataset. The figure below displays a correlation matrix indicating correlation coefficients between each variable in the dataset.

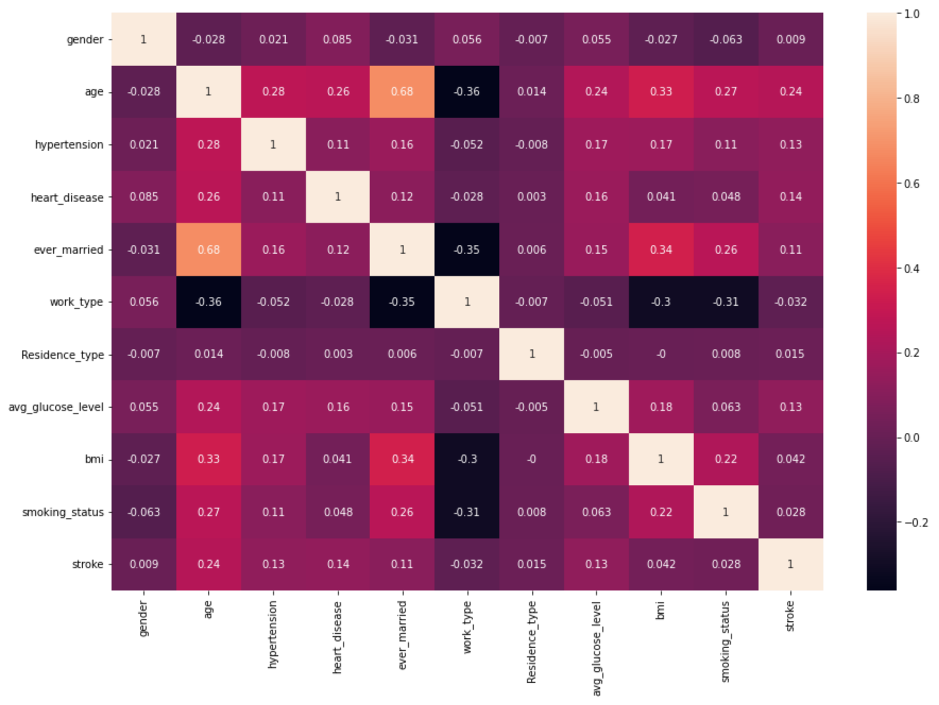


Figure 14: Pearson correlation matrix for variables in stroke dataset

For this research work, we will consider a variable with a correlation coefficient of >0.3 to have an exceptional level of multicollinearity which will be reviewed for its impact on model efficacy. In the case of this dataset, Work Type has a significant correlation with Smoking Status, BMI, Ever Married and Age. An investigation will be conducted into the impact of removing these variables on a per-model basis with particular reference to the impact on model accuracy and f1-score.

Data Preparation

Missing Values

As previously mentioned, the dataset suffered from missing values in the BMI column. While the number of missing values is small compared to the overall dataset size, at 3.9%, it is essential to note that the rows with missing values account for 16% of the datasets stroke values. Consequently, dropping all rows with missing data may hamper overall model accuracy. With that in mind, datasets will be replicated several times, one with NaN values dropped, one with NaN values replaced by the column mean, and the final having NaN values imputed using Scikit-Learn's K-Nearest Neighbour imputation algorithm with K set to 2. These datasets can then be assessed independently for overall accuracy to establish the most suitable course of action.

Class Imbalance

A similar approach to handling NaN values was selected regarding the imbalance outlined above of stroke values in the dataset. Independent datasets were created using both random undersampling, whereby the number of rows with 0 in the stroke column is reduced to match the stroke values—combined with SMOTE oversampling, where stroke values are replicated using present data to ensure that stroke incidence in the dataset matches that of healthy patients (Chawla et al., 2002). Again the different datasets could be analysed through several machine learning techniques to enable appropriate selection or further recommendations.

Label/One-hot Encoding

In the case of the numerous categorical (binary or multi-category) or continuous variables, it is essential to note that before these variables can be passed towards machine learning models to make accurate predictions, it is first necessary to ensure that data is processed to promote proper training and subsequent predictions (Goodfellow et al., 2016). These variables were primarily represented in the original dataset using string identifiers (Gender, ever_married, work_type and Residence_type. While these strings are helpful when conducting an exploratory data analysis to identify any relationships between variables, they cannot be processed by machine learning models. In response to this, Scikit-Learn's LabelEncoder and OneHotEncoder packages were used to convert these categorical variables to integers that machine learning algorithms can interpret. In the case of LabelEncoder, strings are replaced by integers while the single-column format is maintained.

To enable comparative analysis of the impact of encoding approaches on overall model accuracy, categorical values will also be one-hot encoded, involving splitting single multi-category columns into multiple binary columns. Table one displays a before and after picture of the changes implemented by LabelEncoder.

Before Encoding	Label Encoding	Onehot Rural	Onehot Urban
Urban	1	0	1
Rural	0	1	0
Rural	0	1	0
Urban	1	0	1
Rural	0	1	0

Table 3: Comparison of Label Encoding vs One-hot Encoding of Categorical Variables

Train/Test Split

Before being passed to the models for training, data received a train/test split, with a 30/70 split in favour of training data. This split aims to ensure that a portion of the original dataset is set aside for judging a given model's overall accuracy on data unseen during training.

Range Scaling

Finally, as noted by Müller and Guido (2016), some machine learning techniques such as Deep Learning are susceptible to the scaling of the underlying data. For instance, in their raw format, BMI values range between 10.3 and 97.6, while average glucose levels range from 45.28 to 271.74. If left unaddressed, any trained model may weigh in favour of one variable over the other, regardless of the impact on the accuracy of the overall prediction. With this in mind, Scikit-Learn's MinMaxScaler package will be implemented to scale data for the Age, Average Glucose Level and BMI to a range between 0 and 1. In order to prevent data leakage (Weiran, 2021), datasets were subjected to a train/test split prior to scaling values.

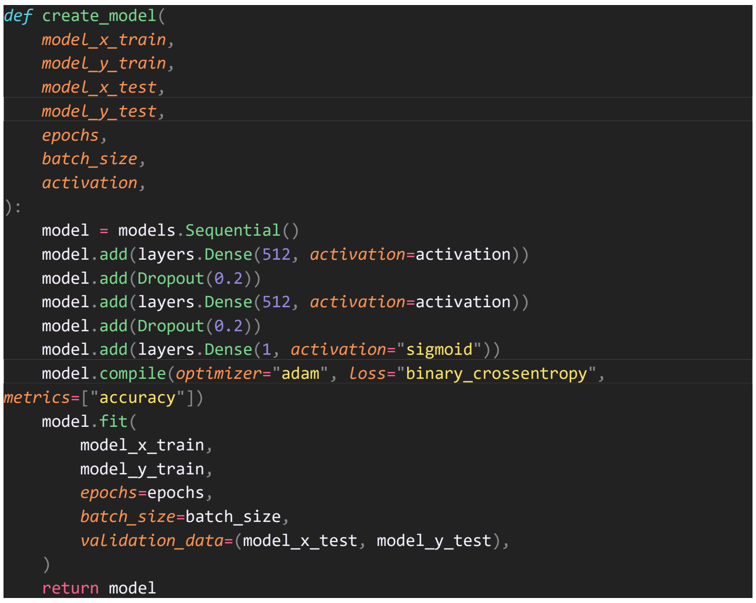
Implementation of Machine Learning

Given the fact that the appropriate machine learning technique to make predictions for a given dataset is impacted by a large number of variables, the decision was made to produce a large number of models to identify those that displayed the most significant efficacy before completing any hyperparameter tuning on any one given model. With this in mind, an Artificial Neural Network consisting of dense layers alongside a Convolutional Neural Network, Random Forest, Logistic Regression, Linear Regression, and K-Nearest Neighbour models. The remainder of this report will outline their architecture and accuracy and make recommendations for further study.

Artificial Neural Network

Model Construction

First, a network consisting of two fully connected dense layers with 512 neurons, separated by dropout layers with a 20% probability to address the potential for training overfit. Both dense layers utilised a Relu activation function to address non-linearity. The model utilised a binary cross-entropy loss function, implemented using the Adam optimiser algorithm. Full details of the model creation function can be seen in the code below:



```
def create_model(
    model_x_train,
    model_y_train,
    model_x_test,
    model_y_test,
    epochs,
    batch_size,
    activation,
):
    model = models.Sequential()
    model.add(layers.Dense(512, activation=activation))
    model.add(Dropout(0.2))
    model.add(layers.Dense(512, activation=activation))
    model.add(Dropout(0.2))
    model.add(layers.Dense(1, activation="sigmoid"))
    model.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["accuracy"])
    model.fit(
        model_x_train,
        model_y_train,
        epochs=epochs,
        batch_size=batch_size,
        validation_data=(model_x_test, model_y_test),
    )
    return model
```

Code 1: ANN model creation function

Through creating a function, it was possible to replicate a similar architecture with updated parameters for the number of epochs, batch size and activation function while focusing the model itself on different datasets as outlined above. When models were trained, they were then assessed for fit by making predictions on the reserved test data and comparing them to the known test values. The outcome of this comparison can then be visualised in a confidence matrix.

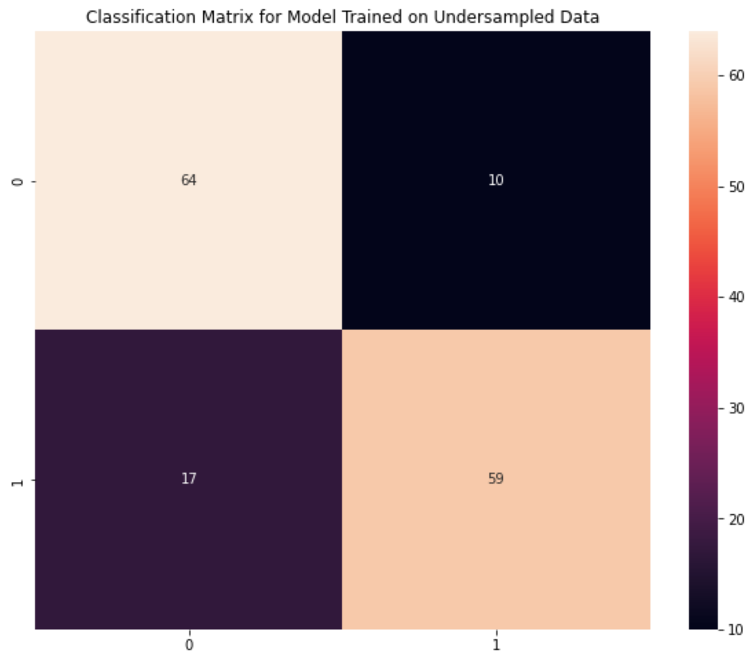


Figure 15: Example confidence matrix for ANN model

Model Results

In the case of the datasets where NaN BMI values were dropped, replaced with column mean and imputed using KNN, the model displayed high test accuracy of approx. 95% (Table 2). However, on inspection, it is possible that the models in each case did not predict any positive results. Given the imbalance of negative results in the dataset of 95% in favour of healthy patients, it is clear that while test accuracy is high, model efficacy is negligible.

	Test Loss	Test Accuracy	True Negative	True Positives	False Positives	False Negatives
NaN Dropped	0.190651	0.947047	930	0	0	52
NaN to Mean	0.177736	0.947815	1453	0	0	80
NaN Imputed	0.172180	0.953686	1467	0	2	64
Mean SMOTE	0.377154	0.838134	1161	1283	170	302
Undersampled	0. 455508	0.819999	64	59	17	10
One-hot SMOTE	0.456998	0.838028	1052	19	35	172

Table 4: ANN Model Performance Results

With this in mind, the SMOTE class imbalance algorithm will be implemented to address this dataset imbalance. The below code snippet helps to illustrate how the algorithm was implemented in Python but was a simple case of applying the function to the x and y values as separate arguments. The outcome was an increase in stroke incidence within the dataset from 249 to 4861, the same figure as healthy patients.

It should be noted that the most effective implementation of dataset scaling would take place after data was split into a training and testing set. However, during implementation, it was noted that, typically, only a minimal number of positive stroke values would be included in the training set for scaling due to the scale of the imbalance. This meant that even after scaling, the variation in predictor variables with positive stroke values was minimal, causing the model to have less information to build appropriate weights, resulting in model performance degradation to the tune of 10-15% accuracy.

```
oversampled_data = data.fillna(data.bmi.mean())
oversampled_y = oversampled_data["stroke"]
oversampled_x = oversampled_data
oversampled_x.drop("stroke", axis=1, inplace=True)

sm = SMOTE(random_state=0)
oversampled_x, oversampled_y = sm.fit_resample(oversampled_x, oversampled_y)
```

Code 2: SMOTE Scaling of Dataset

While the accuracy of the neural network trained on the oversampled data was lower than the other three attempts at 85%, the networks ability to accurately predict stroke risk was significantly improved with 1,300 true positives identified. However, given the medical nature of this work, it is essential to note the 143 false negatives identified, suggesting some further room for improvement. Application of the Random Under Sampler algorithm produced similar results with an overall accuracy of 80%, correctly predicting 63 true positives, with a further 12 false negatives.

The most successful models were trained on 25 epochs with a batch size of five. Model accuracy rarely improved with the addition of additional epochs. It is also worth noting that a fifth model was trained using one-hot encoding on predictor variables, oversampled with SMOTE to address the imbalance. This model produced a test accuracy of 83%, comparable to the same model with label encoded predictor variables. However, a significant consideration here is the far higher incidence of false negatives.

Model Optimisation

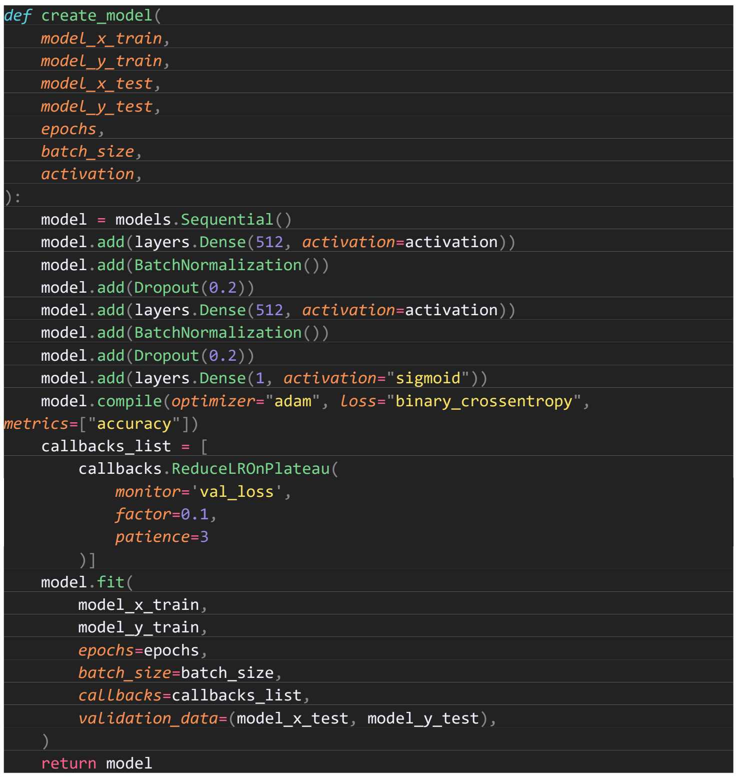
As a result of the label encoded, SMOTE oversampled, dataset status as the most promising candidate with the initial model parameters, the decision was made to use this as the dataset of focus for further model optimisation. This optimisation would have the core objectives of increasing validation accuracy and decreasing the number of false negatives observed in a produced confusion matrix through further hyperparameter tuning.

To decrease training loss, several different combinations of dense neurons quantity, batch size and epoch quantities were selected and changed the activation function of the first two dense layers. In all cases, this did not lead to an increase in model accuracy. Furthermore, an attempt was made to implement the ReduceLROnPlateau, which reduces model learning rate on loss plateau after a given number of epochs with no decrease in training loss, known as Patience. Finally, batch normalisation was added after each dense layer to make different samples of data seen by the model seem more similar to one another, allowing the model to generalise better to data from outside of the training set.

Finally, when considering the SMOTE scaling of the dataset, as mentioned previously a limitation of this approach is the benefit of applying scaling before train/test split in order to ensure an equal proportion of classes in the training data. In order to assess it's efficacy the another attempt was made to split the model testing and results method in order to test the impact of adjusting rounding thresholds on model predictions (Brownlee, 2021). To achieve this a for loop was implemented to iterate through thresholds between 0.1 and 0.9 at 0.1 intervals. This loop produced a dataframe with thresholds and impact on model f1-scores. Using this the model with the highest f1 score could be implemented. In the case of oversampled data, an update to a threshold of 0.6 resulted in a n increase in overall model accuracy to 85.8%, an increase of 2% over baseline, it should be noted however that this approach increased false positive rate and thus resulted in a decrease in f1-score.

	Test Loss	Test Accuracy	True Negative	True Positives	False Positives	False Negatives
Mean SMOTE	0.345409	0.858073	1203	1300	271	143

However, making adjustments to the model parameters described above - in various combinations - led to no noticeable increase in validation accuracy or decrease in training loss over the 85.8% validation accuracy initially noted.



```
def create_model(
    model_x_train,
    model_y_train,
    model_x_test,
    model_y_test,
    epochs,
    batch_size,
    activation,
):
    model = models.Sequential()
    model.add(layers.Dense(512, activation=activation))
    model.add(BatchNormalization())
    model.add(Dropout(0.2))
    model.add(layers.Dense(512, activation=activation))
    model.add(BatchNormalization())
    model.add(Dropout(0.2))
    model.add(layers.Dense(1, activation="sigmoid"))
    model.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["accuracy"])
    callbacks_list = [
        callbacks.ReduceLROnPlateau(
            monitor='val_loss',
            factor=0.1,
            patience=3
        )]
    model.fit(
        model_x_train,
        model_y_train,
        epochs=epochs,
        batch_size=batch_size,
        callbacks=callbacks_list,
        validation_data=(model_x_test, model_y_test),
    )
    return model
```

Code 3: Model creation function after optimisation steps

Convolutional Neural Network

Model Construction

The challenge with applying a convolutional model to tabular data is reshaping the data to be interpreted by the convolutional layers. An attempt was made to utilise a convolutional layer to assess whether this resulted in any level of increased accuracy over a typical ANN. The TensorFlow backend was utilised to apply an expanded dimension to each row of the dataset by utilising a lambda function. This reshaped data was then passed to a 1-Dimensional convolutional layer with a kernel size of 3 and filter size of 5. Both larger and smaller kernel and filter sizes were attempted to reduce validation accuracy.

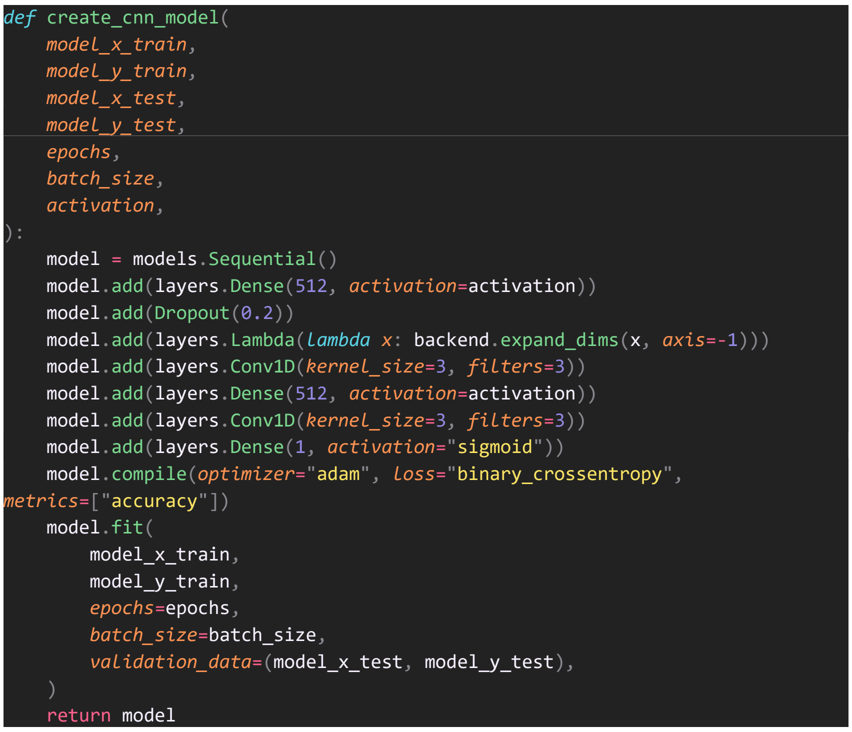
Other than the described reshape and convolutional layers, the CNN architecture followed in a similar vein to the previously described Fully Connected Neural Network, with a Dense initial layer with 512 neurons, followed by a dropout layer with a 20% probability to address the potential for model overfit. In addition, there was a subsequent dense layer between the two convolutional layers. The second convolutional layer's output is passed through a single neuron dense layer with a sigmoid activation function. An issue that arose from this method of data preparation prevention of model predictions being castable into confusion matrices, we will have to rely on accuracy metrics alone to measure model accuracy.

Model Results

Due to their overwhelming accuracy in the previous rounds, in addition to the significant increase in training time of the convolutional model, only over and undersampled data were used in model training. Accuracy vs traditional ANNs was significantly degraded with a peak test accuracy of 79% for the model trained on the oversampled data. The undersampled model was significantly reduced vs ANN with a test accuracy of 47.7%.

	Test Loss	Test Accuracy
SMOTE Oversampled	0.447929	0.791334
Undersampled	2.923244	0.477165

Table 5: Test accuracy metrics for convolutional neural network



```
def create_cnn_model(
    model_x_train,
    model_y_train,
    model_x_test,
    model_y_test,
    epochs,
    batch_size,
    activation,
):
    model = models.Sequential()
    model.add(layers.Dense(512, activation=activation))
    model.add(Dropout(0.2))
    model.add(layers.Lambda(lambda x: backend.expand_dims(x, axis=-1)))
    model.add(layers.Conv1D(kernel_size=3, filters=3))
    model.add(layers.Dense(512, activation=activation))
    model.add(layers.Conv1D(kernel_size=3, filters=3))
    model.add(layers.Dense(1, activation="sigmoid"))
    model.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["accuracy"])
    model.fit(
        model_x_train,
        model_y_train,
        epochs=epochs,
        batch_size=batch_size,
        validation_data=(model_x_test, model_y_test),
    )
    return model
```

Code 4: CNN creation function definition

Model Optimisation

In a repetition of the work completed in attempting to optimise the previously created Dense Neural Network, several different epoch and batch sizes were trialled, in addition to the inclusion of both Batch Normalisation layers into the model architecture after Dense and Convolutional layers, in addition, to use of the ReduceLROnPlateau callback function. The Dense Network saw no notable increases in model accuracy over the first iteration of the model's architecture by applying these steps.

Compared to the DFNN, these additional model layers and steps significantly increased overall model training time. However, in contrast to the optimisation performance on the DFNN, accuracy metrics were marginally improved for the oversampled data to 80.37%. On the undersampled data, validation accuracy increased to 48.37%. Accuracy statistics, bearing in mind the previously outlined limitations on converting predictions into confusion matrices, can be seen below.

	Test Loss	Test Accuracy
SMOTE Oversampled	0.438557	0.8034838
Undersampled	2.418018	0.4837401

Code 5: Post optimisation CNN accuracy metrics

Random Forest

Model Construction

When creating a Random Forest Model, an important consideration is the number of trees included in the forest. While Scikit-Learn defaults to 100 trees, it is possible to improve the model's overall accuracy. With this in mind finding a way to iterate through several different estimator quantities and return the estimator number with the greatest f1-score, a measure that considers the number of true positives, false positives and false negatives; illustrated by the equation below.

$$f1 = 2.\frac{precision\,.recall}{precision + recall}$$

The code used to produce this search can be seen below. In summary, the model would iterate through random forest estimators between 50 and 225 at 25 estimator intervals. This produced a Pandas DataFrame identifying the estimators queried, in addition to the f1-score of the model with those parameters. Once the most accurate number of estimators had been established for each dataset, an adapted model could be run using the identified most accurate n_estimators parameter.

```
def create_forest_classifier_grid(rf_x_train, rf_y_train):
    estimator = []
    f1_score_list = []
    for num in range(50, 250, 25):
        rf_model = RandomForestClassifier(
            n_estimators=num,
            criterion="gini",
            class_weight={0: 1, 1: 100},
            min_samples_leaf=6,
            max_features=None,
            max_depth=3,
        )
        rf_model.fit(rf_x_train, np.ravel(rf_y_train))

        rf_y_pred = rf_model.predict_proba(rf_x_train)
        print("Score on the training set:")
        print(classification_report(rf_y_train, np.around(rf_y_pred[:, 1])))
        print("roc_auc score: ", end="")
        print(roc_auc_score(rf_y_train, rf_y_pred[:, 1]))
        print("f1 score:", f1_score(rf_y_train, np.around(rf_y_pred[:, 1])),
end="\n\n")
        new_f1 = f1_score(rf_y_train, np.around(rf_y_pred[:, 1]))
        estimator.append(num)
        f1_score_list.append(new_f1)
    f1scores = pd.DataFrame(f1_score_list, columns=["f1_score"])
    estimators = pd.DataFrame(estimator, columns=["estimator"])
    output = pd.concat([estimators, f1scores], axis=1)
    return output
```

Model Results

Macro precision was good in the case of the model trained on oversample data with 99% accuracy in predicting negative outcomes and 62% accuracy predicting positive outcomes, resulting in a macro average precision score of 81%. All models suffered from a lower level of accuracy in predicting positive values. Perhaps because of the lack of variability in predictor variable suffered in most cases from low accuracy when predicting negative values, with a significant proportion of false negatives. The table below displays the accuracy statistics for the three processed Random Forest models.

Generalised Linear Models - GLM (Linear and Logistic Regression)

Model Construction

Considering the dataset focuses on predicting a binary target variable, Logistic Regression is the most suitable machine learning application. However, a Linear Regression Model will also be outlined to enable comparison.

When applying Logistic Regression to the dataset, two separate avenues were explored to assess the overall impact on the model first. In the first case, these models were only trained on the dataset with NaN BMI values dropped, and the second set of models on NaN values replaced with column mean, and SMOTE oversampled to address dataset imbalance. For the default model using all predictors, an initial model was established using model defaults.

Model Results

Logistic Regression on the dataset with dropped values was 73.9%, with an f1-Score of 0.15. In line with previously developed models, accuracy for SMOTE oversampled data is moderately higher with an overall accuracy of 80.9% but with an f1-Score of 0.811. The improvement of positive case identification likely brings about this significant increase in f1-Score. Model accuracy for Logistic Regression can be seen below.

	Accuracy	**Precision**	**Recall**	**F1-Score**	**TN**	**TP**	**FP**	**FN**
NaN Dropped	0.739308	0.088889	0.705882	0.157895	702	24	246	10
SMOTE Oversampled	0.809393	0.804027	0.819425	0.811653	1163	1198	292	264
RFE Dropped	0.965377	0	0	0	948	0	0	34
RFE Oversampled	0.615358	0.624633	0.582763	0.602972	943	852	512	610

Table 6: Logistic regression accuracy metrics

Model Optimisation

A further investigation was completed using predictor variable selection using Recursive Feature Elimination (RFE) to investigate the impact of multicollinearity on model accuracy. RFE functions as a feature selection algorithm that creates a model for all predictors and computes an importance score for each predictor. It removes minor essential predictors and iterates through the process until the correct predictor features for a given dataset are identified (Kuhn & Johnson, 2019). Essential parameters for RFE are both the step, outlining how many predictor variables are moved per iteration, and the n_estimators parameter, which functions at the ideal number of predictor variables to remain after the RFE process completes. Initially, these were set to a step size of 1 and the number of features to 4, with RFE suggesting a model using 'hypertension', 'heart_disease', 'ever_married' and 'bmi'.

When reviewing the models implemented using RFE, accuracy was markedly decreased over the SMOTE oversampled model with an overall accuracy score of 61.5% with an f1-Score of .602. Interrogation of the confusion matrix saw a significant increase in the false positive and false negative classes (Type I and Type II errors, respectively).

Table 7: Logistic regression accuracy metrics (RFE Optimised)

Linear Model

When creating the Linear Model, the implementation of the system itself was broadly similar to the logistic regression model described previously, with a requirement to use both the Scikit-Learn Linear Regression package. Unlike logistic Regression, which produces discrete outputs fitting predictions against a sigmoid curve, linear Regression makes continuous predictions against a line of best fit. The implication for this model is a question of accuracy alongside how best to interpret predictions. For this study, linear regression predictions were made and then rounded to the closest integer using Pythons built-in round function. The rounded figures can then serve as the models' predictions, facilitating accurate model fit analysis.

As with many other models, dataset imbalance leads to a significant accuracy metric for the model trained on data with no additional sampling and NaN values dropped. However, viewing model accuracy within the context of a -0.042 R^2 value, which serves as a measurement of the variance seen in the target variable that is explained by variations in the predictor variables, it is possible to state that the model is ineffective in making reliable predictions of stroke risk for patients.

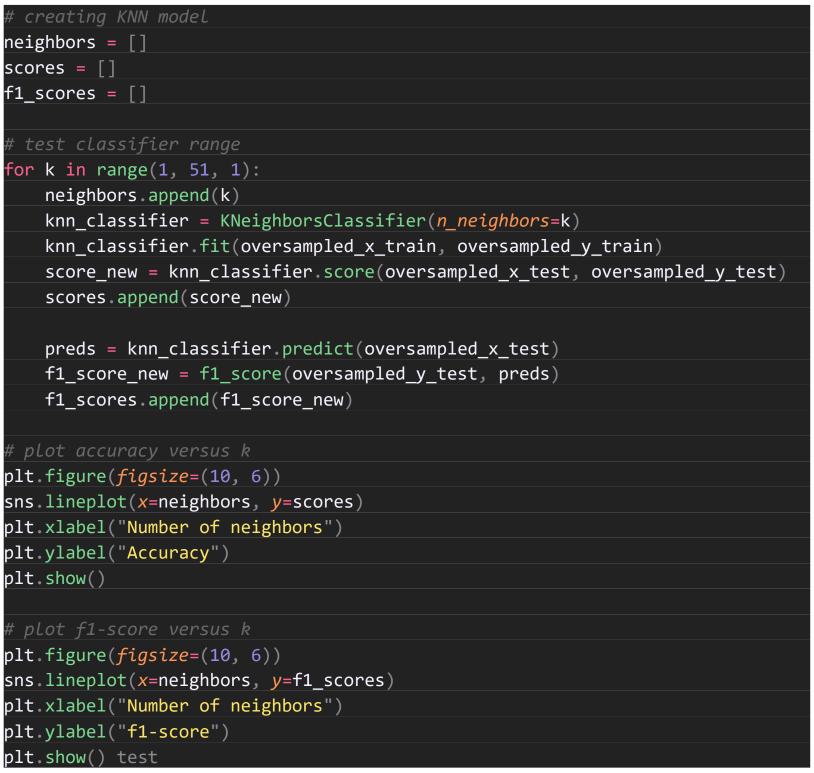
	R2	**RMSE**	**MAE**	**Acc.**	**Prec.**	**Rec.**	**F1**	**TN**	**TP**	**FP**	**FN**
NaN Dropped	-0.042	0.202	0.041	0.969	0	0	0	942	0	0	40
SMOTE Oversampled	0.240	0.436	0.189	0.810	0.798	0.831	0.814	1148	1215	307	247

Table 8: Linear regression accuracy metrics

K-Nearest Neighbour

A final model established to compare various machine learning methods is K-Nearest Neighbour, an algorithm that estimates unknown values based on the k-nearest number of most similar known values (Burkov, 2019). The most vital consideration is to ensure that a value is selected for K that is the most suitable for a given dataset. With this requirement in mind, the speed and flexibility of machine learning comes into its own. With a few lines of code, it is possible to iterate through a large number of iterations of a given model, enabling a user to see how K values within a given range compare both in terms of accuracy and f1-score, given that as outlined already, using accuracy as a sole measure of success rarely gives the complete picture of a given models performance in real-world predictions.

	Macro Precisi-on	**Weighted f1-score**	**Mean AUC**	**True Negative**	**True Positives**	**False Positives**	**False Negatives**
NaN Dropped	0.54	0.66	0.8296	503	38	2	439
SMOTE Oversampled	0.81	0.65	0.7882	541	1464	2	910
Undersampled	0.75	0.61	0.8393	28	70	2	50



```python
# creating KNN model
neighbors = []
scores = []
f1_scores = []

# test classifier range
for k in range(1, 51, 1):
    neighbors.append(k)
    knn_classifier = KNeighborsClassifier(n_neighbors=k)
    knn_classifier.fit(oversampled_x_train, oversampled_y_train)
    score_new = knn_classifier.score(oversampled_x_test, oversampled_y_test)
    scores.append(score_new)

    preds = knn_classifier.predict(oversampled_x_test)
    f1_score_new = f1_score(oversampled_y_test, preds)
    f1_scores.append(f1_score_new)

# plot accuracy versus k
plt.figure(figsize=(10, 6))
sns.lineplot(x=neighbors, y=scores)
plt.xlabel("Number of neighbors")
plt.ylabel("Accuracy")
plt.show()

# plot f1-score versus k
plt.figure(figsize=(10, 6))
sns.lineplot(x=neighbors, y=f1_scores)
plt.xlabel("Number of neighbors")
plt.ylabel("f1-score")
plt.show() test
```

Code 6: KNN range search and plotting definition

The code outlined above illustrates how, using a for loop, it is possible to iterate for K values between one and 50, with a step of one—training a model, generating accuracy and f1-scores and adding these to a list. Once models have been built for the fifty K-values, each of these values was appended to a list before having these lists graphically represented using a Seaborn line plot. To further aid in identifying the most accurate K-value, these lists could be converted to a Pandas DataFrame with the Python max() function to identify the line with the highest f1-score.

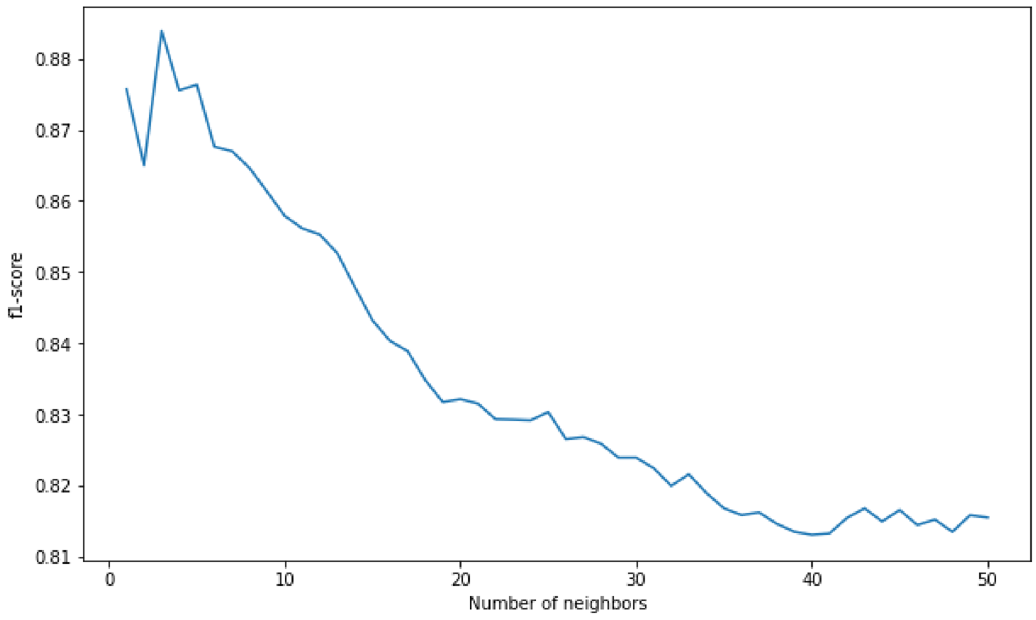


Figure 16: Plotting f1-score for K Neighbours between a range 1-50 (inclusive)

As has been seen consistently with previous work, while accuracy with NaN values dropped is high, the models' efficacy in making predictions is extremely low due to model bias towards negative predictions. Once the most accurate K-value has been identified, 3 in the case of the oversampled dataset, a model could be retrained using the known value before final accuracy measures could be established and a confidence matrix produced. The output of this work can be seen in the table below.

	Accuracy	F1-Score	TN	TP	FP	FN
NaN Dropped (K=1)	0.937882	0.164384	915	6	34	27
SMOTE Oversampled (K=3)	0.877957	0.883888	1134	1325	127	331

Table 9: KNN Accuracy metrics, K noted.

Conclusion

This work serves as a case study of the dangers of accepting accuracy as a single measure of accuracy when considering the effectiveness of a given model. In this example, based on the parameters of the model given, the most accurate model was established through applying K-Nearest Neighbour, given its accuracy metrics approaching 90%. A vital consideration in all models was that the imbalance of the dataset had a significant impact on the ability to produce accurate models without using some form of scaling to allow the ML algorithms to avoid biasing negative predictions.

While some SMOTE scaled models' noted accuracy was positive, they did note a large number of False Negative results. Given the healthcare application of this work, it is crucial to consider the potential for life-changing impacts suffered by patients as a matter of misdiagnosis. As such, it would be worth conducting additional research to increase the number of positive results and enable more accurte training of predictions, given that a sample size of 5,110 is relatively small given the Big Data approaches being implemented.

Furthermore, this study only includes five models, with a limited number of techniques applied within each, due to limitations in the capacity of the project. Further investigation could be conducted into additional ML techniques and optimisation approaches to ensure that accuracy was increased while False Negative results were controlled.

References:

Brownlee, J. (2021, January 5). A Gentle Introduction to Threshold-Moving for Imbalanced Classification. Machine Learning Mastery. https://machinelearningmastery.com/threshold-moving-for-imbalanced-classification/

Burkov, A. (2019). *The Hundred-Page Machine Learning Book*. Andriy Burkov (self-published).

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16, 321–357. https://doi.org/10.1613/jair.953

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning* (Adaptive Computation and Machine Learning series) (Illustrated ed.). The MIT Press.

Chen, R., Ovbiagele, B., & Feng, W. (2016). Diabetes and Stroke: Epidemiology, Pathophysiology, Pharmaceuticals and Outcomes. *The American Journal of the Medical Sciences*, 351(4), 380–386. https://doi.org/10.1016/j.amjms.2016.01.011

Kuhn, M., & Johnson, K. (2019). *Feature Engineering and Selection: A Practical Approach for Predictive Models* (Chapman & Hall/CRC Data Science Series) (1st ed.). Chapman and Hall/CRC.

Müller, A. C., & Guido, S. (2016). *Introduction to Machine Learning with Python: A Guide for Data Scientists* (1st ed.). O'Reilly Media.

Weiran, S. (2021, December 12). *Avoid Data Leakage — Split Your Data Before Processing*. Medium. https://towardsdatascience.com/avoid-data-leakage-split-your-data-before-processing-a7f172632b00

Appendix 1 : Updated model results function showing iteration code through thresholds.

```
def model_results(model, results_x_test, results_y_test, title):
    results = model.evaluate(results_x_test, results_y_test, batch_size=20)
    print("test loss, test acc:", results)

    predictions = model.predict(results_x_test)
    score=[]
    thresholds = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
    for num in thresholds:
        rounded_preds = np.where(predictions > num, 1, 0)
        f1_score_new = f1_score(results_y_test, rounded_preds)
        score.append(f1_score_new)

    num_df = pd.DataFrame(thresholds)
    num_df.rename(columns={0:"Threshold"}, inplace=True)
    score_df = pd.DataFrame(score)
    score_df.rename(columns={0:"f1-Score"}, inplace=True)
    output = pd.concat([num_df, score_df], axis=1)
    return output
```

Appendix 2: Updated model output results function, producing model accuracy results for updated threshold quantity.

```
def model_output(model, results_x_test, results_y_test, title, threshold):
    predictions = model.predict(results_x_test)
    threshold_rounded_preds = np.where(predictions > threshold, 1, 0)

    conf_matrix = confusion_matrix(results_y_test, threshold_rounded_preds)

    accuracy = accuracy_score(results_y_test, threshold_rounded_preds)
    precision = precision_score(results_y_test, threshold_rounded_preds)
    recall = recall_score(results_y_test, threshold_rounded_preds)
    score = f1_score(results_y_test, threshold_rounded_preds)

    base_rounded_preds = np.where(predictions > 0.5, 1, 0)
    base_accuracy = accuracy_score(results_y_test, base_rounded_preds)
    base_precision = precision_score(results_y_test, base_rounded_preds)
    base_recall = recall_score(results_y_test, base_rounded_preds)
    base_score = f1_score(results_y_test, base_rounded_preds)

    plt.figure(figsize=(10, 8))
    plt.title(title)
    plt.xlabel("Predicted Value")
    sns.heatmap(conf_matrix, annot=True, fmt="d")

    print(f"New model accuracy is: ",accuracy)
    print(f"Change in model accuracy with updated threshold: ", accuracy-
base_accuracy)
    print(f"New model precision is: ",precision)
    print(f"Change in model precision with updated threshold: ", precision-
base_precision)
    print(f"New model recall is: ",recall)
    print(f"Change in model recall with updated threshold: ", recall-
base_recall)
    print(f"New model f1-Score is: ",score)
    print(f"Change in model f1-score with updated threshold: ", score-
base_score)
```