ACML Project 2022

Yaseen Haffejee (1827555)
Ziyaad Ballim (1828251)
Jeremy Crouch (1598024)

1 Introduction

The purpose of the project was to find a dataset and based on the various properties of the dataset, select an appropriate machine learning algorithm to solve the problem at hand. Numerous optimisation techniques needed to be implored in order to optimise the chosen machine learning algorithm. Thereafter a brief analysis of the results was required. The dataset chosen was that of the Ultimate Fighting Championship (UFC), which recorded various fighter details from fights dating between 1993 and 2021. Based on the characteristics of the dataset, two algorithms were implored and studied comparatively, namely Extreme Gradient Boosting (XGBoost) and a Neural Network. The results showed that the both models achieved a similar accuracy, however the Neural Network provided more consistency in its predictions. The paper will follow the structure with section 2 discussing the dataset, section 3 will give an in depth explanation of the machine learning models, section 4 will provide an analysis of the results and section will conclude the report.

2 Dataset

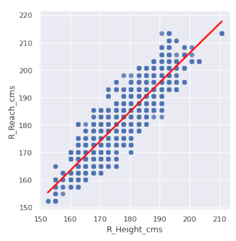
The dataset chosen was that of the UFC which captured fighters details for all the fights which occurred between 1993 and 2021. The details of each fight were scraped from the UFC stats website. Each row is a compilation of both fighter stats. Fighters are represented by 'red' and 'blue' which indicates which corner they were in. Within each instance, the red fighter has the complied average stats of all the fights except the current one. The stats include damage done by the red fighter on the opponent and the damage done by the opponent on the fighter, in their fights prior to the current fight. The same information exists for the blue fighter. The target variable is the column 'Winner' which indicates the outcome of the fight.

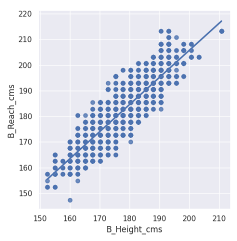
2.1 Features exploration

The dataset contained 6012 rows with 144 features. However, certain features such as the fighters names, the referee, the date of the fight and the location were dropped since these features have no predictive power in the outcome of the fight.

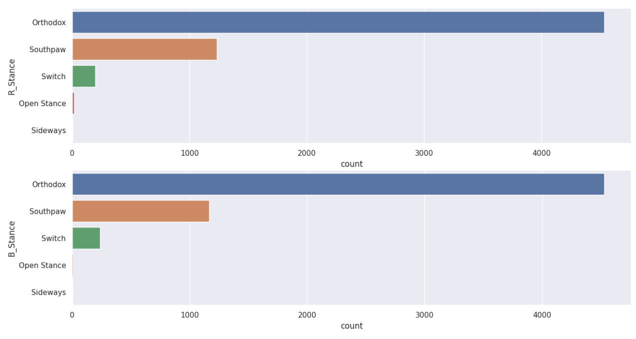
2.1.1 NaN values

The dataset initially contained 6012 rows, however 2098 of these contained NaN values. We explored the columns which contained these NaN values and tried to fill them in when possible. One of the variables which we managed to fill in was the fighters height and reach. We found that these two features were correlated, and consequently used them to fill in any missing values in these columns. The figures below denote the correlation between these features.





Another feature that the NaN values were filled in for was the stance column. We looked at the mode of each corner, and assigned that value to the missing values since it is extremely likely that the fighter will use this stance. The plot below shows that the most frequent stance was orthodox and thus it was used to fill in the missing values.



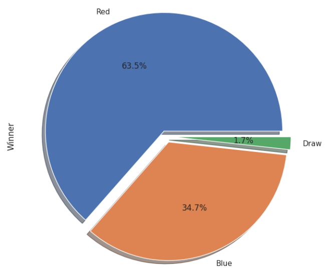
Since the other features that were missing values were fighter dependent, we could not fill these values in. Consequently, we dropped these rows and ended up with 4012 instances.

2.1.2 Encoding

Features such as the weight class, stance, and winner were all categorical. In order to utilise these in most machine learning algorithms, we need to encode them. Since these categorical variables are not ordinal, we utilised one-hot encoding for them.
The final categorical feature was title bout. This represented whether or not a fight was for a championship or not. Since fights for championships are more significant, we utilised label encoding for this feature.

2.2 Target Variable

The target variable is denoted by the column "Winner". The column indicates which corner won the fight, and in the rare instance, it is possible for the fight to end as a draw. The proportion of the possible results is indicated in the pie-chart below.



We can see that we have quite a large class imbalance which may hinder any models ability to predict the minority class "Draw". Since the majority target variable belongs to the red corner, most classification algorithms should be able to learn to predict this class quite easily.

2.3 Exploratory Data Analysis

Based on the features selected by the Variance Inflation Factor (which will be further expanded on in 2.4.1) we performed various data exploratory tasks, to gain a better understanding on the impact and meaning of the features.

2.3.1 Understanding Weight Classes by Combat-related features

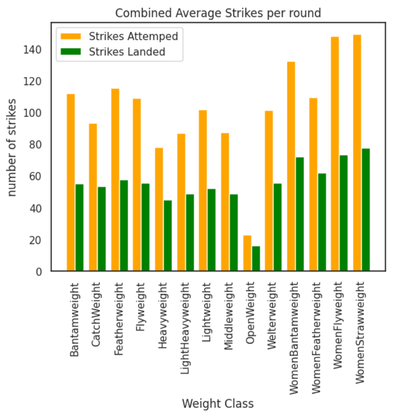


Figure 1: Weight-classes by average strikes per round

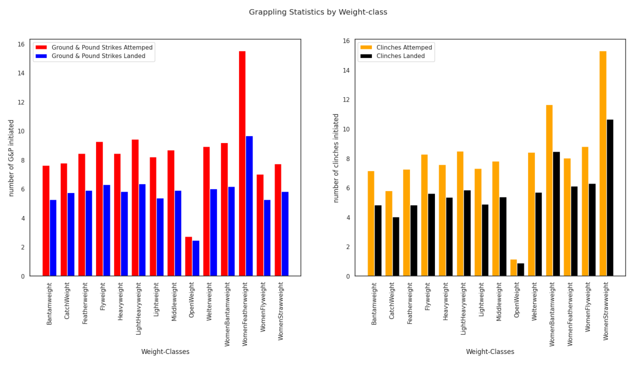


Figure 2: Weight-classes by average grappling related actions per round

Looking at the above figures - some trends become obvious.

- Open-weight Category can be treated as an outlier; looking at the history UFC - this was a short-lived category that was featured in the first 6 UFC tournaments and allowed fighters of any weights to go against each other. These fights were done for the sole purpose of curiosity and entertainment.
- Striking: We can see a general trend with more average strikes occurring in the lighter weight classes and lessening going up. This could be due to the fact that generally lighter people have a lot more energy and fatigue less easily than heavier people. The lighter a fighter is, the weaker their punches tend to be and so can unload a lot more punches in a fight. With a stronger punch, comes a lot more knockout-power and increase the chance of cutting a fight short by knocking-out an opponent, so will have less average punches overall.

- Grappling: In the grappling statistics we see a similar trend amongst all the weight-classes with similar attempts and successful actions. We would assume this would be due to the fact that initiating grappling is quite different to striking. Often times, in a fight in a specific round a fighter will initiate a grappling technique, land it successfully, and then control them in the position till the end of the round. There are fighters out there who do not grapple at all (such as current Middleweight champion Israel Adesanya), so the data associated with grappling will have a lot more 0 values.

- Women's statistics: The women's division was introduced into the UFC in 2013 and still to this day does not have as large a roster or frequent fights as the Men's division, so the data is skewed. However despite being skewed, we still see the same trend amongst the women's divisions in terms of lighter divisions have more average strikes thrown per fight.

2.3.2 Significant Strikes by Winner

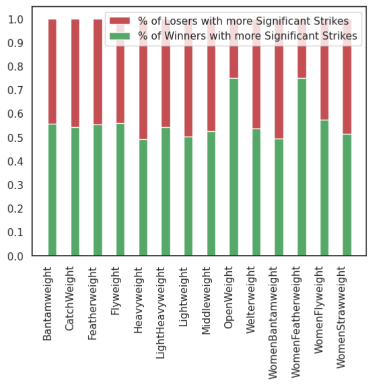


Figure 3: Winners in each Weight-Class who attained more significant strikes in a fight

Looking at the figure above, and based on the conclusions deduced about the Open-Weight and Women's weight divisions - we can see that having the most strikes in a fight is not a strong signifier for the winner.

2.3.3 Strikes by stance

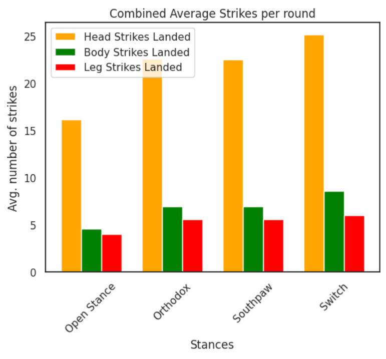


Figure 4: Average frequency for each type of strike by fighter's preferred stance.

Through this analysis - we can see the preferred type strike amongst all stances is a head strike. This makes sense as hitting someone in the head as it is the most open part of their body, as well as the fact that it is the area that is most likely to hit to result in a knock-out. Orthodox and Southpaw have near identical characteristics, this is due to the fact a fighter will generally pick this as their preferred stance if they have a dominant punching hand (right hand for Orthodox, left hand for southpaw). The switch stance is most utilized by kick-boxers and Tae-kwando practitioners which explains which it has a higher average number of overall strikes. The Open stance is specifically utilized for counter-striking, so it would make sense that it has a lower overall average that compared to the other stances.

2.3.4 Prior Wins and Losses by Weight Class

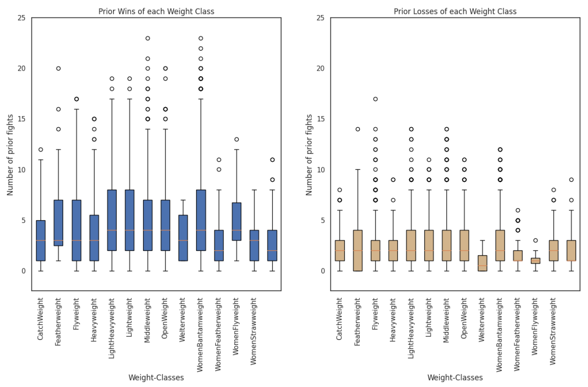


Figure 5: Average frequency for each type of strike by fighter's preferred stance.

A general pattern can't be deduced from the Prior Wins graph, however prior losses has a similar number of Lower and Upper Quartiles, as well as means - this is could possibly be explained by the UFC's ruthless approach it takes to fighter selection. Often times a fighter will be dropped from the roster should they go on a 3 fight losing streak; or won't be signed initially should they have more than 4 career losses.

2.4 Feature Elimination

Considering the dataset had 144 features, we needed to shrink the feature space since this will increase the complexity of a machine learning model, which will make the model prone to overfitting. We used multiple methods in order to shrink the feature space.

2.4.1 Variance inflation factor

VIF is a number that determines whether a variable has multicollinearity or not. That number also represents how much a variable is inflated because of the linear dependence with other variables. The VIF value starts from 1, and it has no upper limit. If the number gets larger, it means the variable has huge multicollinearity on it. For calculating the VIF, we performed a linear regression process for each variable, where that variable is the target variable. After we do that process, we calculate the R^2 from it. And finally, we calculate the VIF value with this formula: $VIF = \frac{1}{(1-R^2)}$.
We considered any feature with a VIF greater than 10 as highly correlated with other features and removed it. Consequently we removed multicollinearity within the dataset since it will affect the statistical analysis of the importances of features which are conducted below. Upon completion of this process, we were left with 53 features from which we further reduced the feature space.

2.4.2 Using L1 regularization

Regularisation consists in adding a penalty to the different parameters of the machine learning model to reduce the freedom of the model and in other words to avoid overfitting. In linear model regularisation, the penalty is applied over the coefficients that multiply each of the predictors. From the different types of regularisation, Lasso or l1 has the property that is able to shrink some of the coefficients to zero. Therefore, that feature with a 0 coeffecient makes no contribution to the predictive power of the model and can be removed from the model.
We fitted two linear models with regularization, namely logistic regression and a linear support vector component. We then stored the features which did not have 0 coeffecients from both models which we used later on to select the final set of features.

2.4.3 Random Forest

Random forests are one the most popular machine learning algorithms. They are so successful because they provide in general a good predictive performance, low overfitting and easy interpretability. This interpretability is given by the fact that it is straightforward to derive the importance of each variable on the tree decision. In other words, it is easy to compute how much each variable is contributing to the decision. Random forests consist of 4-12 hundred decision trees, each of them built over a random extraction of the observations from the dataset and a random extraction of the features. Not every tree sees all the features or all the observations, and this guarantees that the trees are de-correlated and therefore less prone to over-fitting. Each tree is also a sequence of yes-no questions based on a single or combination of features. At each node (this is at each question), the three divides the dataset into 2 buckets, each of them hosting observations that are more similar among themselves and different from the ones in the other bucket. Therefore, the importance of each feature is derived by how "pure" each of the buckets is.
For classification, the measure of impurity is either the Gini impurity or the information gain/entropy. For regression the measure of impurity is variance. Therefore, when training a tree, it is possible to compute how much each feature decreases the impurity. The more a feature decreases the impurity, the more important the feature is. In random forests, the impurity decrease from each feature can be averaged across trees to determine the final importance of the variable. To give a better intuition, features that are selected at the top of the trees are in general more important than features that are selected at the end nodes of the trees.
We fitted a random forest to the data and then extracted the importances of all the features and stored it. We will use this to determine the optimal features later.

2.4.4 Information Gain

Information gain or mutual information measures how much information the presence/absence of a feature contributes to making the correct prediction on the target. It estimates mutual information for a discrete target variable. Mutual information (MI) between two random variables is a non-negative value, which measures the dependency between the variables. It is equal to zero if and only if two random variables are independent, and higher values mean higher dependency. Since we calculated the information gain between each feature and the targets, the higher the information gain the more influence that feature has in predicting the targets.
We stored the features with the information gain that is not 0, which was 29 of the 53 features, and will use it later to find the optimal features.

2.4.5 Recursive feature elimination with cross validation

Recursive Feature Elimination with Cross Validation (RFECV) selects the best subset of features for the estimator by removing 0 to N features iteratively using recursive feature elimination. Thereafter it selects the best subset based on the accuracy or cross-validation score or roc-auc of the model. Recursive feature elimination technique eliminates n features from a model by fitting the model multiple times and at each step, removing the weakest features.
This method selected 46 features from the available 53, which we store and use to find the optimal set of features.

2.4.6 Selecting the optimal features

In order to select the optimal features, we simply chose the features that were chosen as important by all of the afore mentioned techniques. This enabled us to reduce our feature space from 53 features all the way down to 16 features. We then used these features to train our model.

2.5 Feature Scaling

Since the scales of the selected features are not the same, we performed standardisation to ensure that the features have values in a similar range. From the 2 models we use, XGboost and a Neural Network, the Neural network utilises gradient descent or some variation to minimise the cost function. These minimisation techniques require the features to be on a similar scale to ensure we converge faster, and uniformly to the minima. Consequently, we performed feature scaling as this had no effect on the performance of XGBoost, but ensured the robustness of the Neural Network.

2.6 Splitting the data

Conventionally, the data should be split into 3 sets, the training, validation and testing sets. However, since we utilised a search method to find the optimal hyperparameters, the search method utilises cross-validation to find the optimal values. Thus, it takes the training set and splits it into 2 sets, one for training and another for cross validation. The set used for cross-validation is the validation set. Consequently, the presence of a validation set will serve no purpose and simply further reduce the training set. Hence, we split the data into a training and testing set using 70% of the data for training and 30% for testing.

3 Machine learning algorithms

3.1 Extreme Gradient Boosting

XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework. When it comes to small-to-medium structured/tabular data, decision tree based algorithms are considered best-in-class right now.
Boosting algorithms are an ensemble learning technique. Ensemble learning offers a systematic solution to combine the predictive power of multiple learners. The resultant is a single model that gives the aggregated output from several models. The models that form the ensemble, also known as base learners, could be either from the same learning algorithm or different learning algorithms.

3.1.1 General working of the algorithm

Boosting algorithms seek to improve the prediction power by training a sequence of weak models, each compensating the weaknesses of its predecessors. A weak learner is a model that classifies data slightly better than random guessing.
There are 2 different boosting algorithms:

- Adaptive boosting: This method operates iteratively, identifying misclassified data points and adjusting their weights to minimize the training error. The model continues optimize in a sequential fashion until it yields the strongest predictor.
- Gradient boosting: works by sequentially adding predictors to an ensemble with each one correcting for the errors of its predecessor. However, instead of changing weights of data points like AdaBoost, the gradient boosting trains on the residual errors of the previous predictor.

The general boosting procedure is as follows:

- The first weak learner is created and used to classify the data. The results are shown in the image below. We can clearly see that the boundary is incorrect. There are 3 positives that are incorrectly classified. The residual of this model is calculated, and the next predictor will focus on minimising this residual by learning to classify these incorrectly classified points correctly.

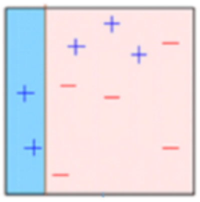


Figure 6: Weak Learner 1

- The next weak learner is created and used to classify the data, but the previously misclassified points are focused on. We can now see that those previously misclassified points are classified correctly, but there are 3 negative values that are classified incorrectly. Consequently, the next predictor will focus on minimising this predictors residual by learning to classify these incorrectly classified points correctly.

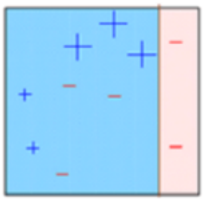


Figure 7: Weak Learner 2

- The next weak learner is created and used to classify the data, but the previously misclassified points are focused on. We can now see that those previously misclassified negative points are classified correctly, but there are 2 negative values that are classified incorrectly. However, those points are classified correctly by the first learner. Thus we know we can stop training.

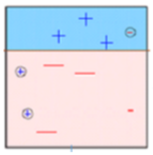


Figure 8: Weak Learner 3

- If we combine the above weak learners, we end up with a strong learner denoted below.

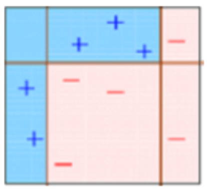


Figure 9: Combined weak learners

3.1.2 Initial model without optimisation

The initial model was extremely simple. We fit a general XGBoost model with 100 estimators. This implies that the maximum number of estimators allowed in the ensemble was 100.
First looking at the training results:

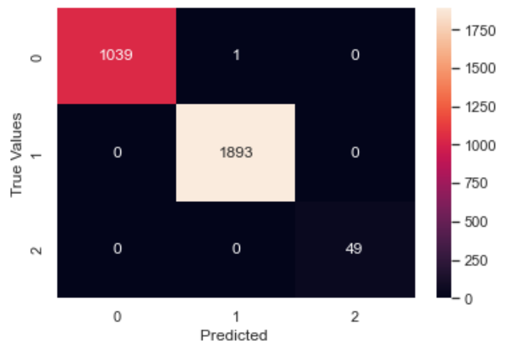


Figure 10: XGBoost Training Confusion Matrix

We see that the model managed to achieve a training accuracy of 99.97%. This was extremely impressive. However, if we look at the figure below, which is a plot of the of the cross-validation performance plot, we see that the training and validation accuracies vary largely. This is indicative of overfitting and a high variance model.

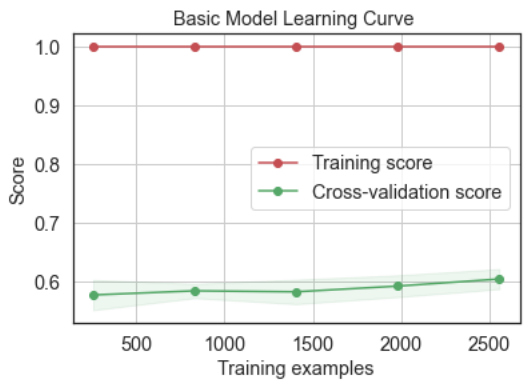


Figure 11: Cross-Validation performance plot

The suspicion of overfitting was confirmed when we used the model to predict on the test data. The confusion matrix is shown below.

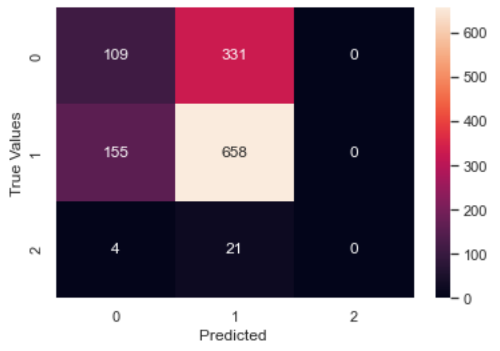


Figure 12: XGBoost Testing Confusion Matrix

From this , we see that the model achieved a testing accuracy of 60% which is a significantly lower than the 99% training accuracy, consequently the model definitely did overfit.

3.1.3 Optimising the model

The goal of the optimization procedure is to find a vector that results in the best performance of the model after learning, such as maximum accuracy or minimum error. The XGBoost model has numerous hyperparameters in order to maximise its performance. These hyperparameters are as follows:

- eta/learning rate: Step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
- max_depth: Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree.
- min_child_weight: It defines the minimum sum of weights of all observations required in a child. It is used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree. Too high values can lead to under-fitting.
- subsample: It denotes the fraction of observations to be randomly samples for each tree. Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. This will prevent overfitting. Subsampling will occur once in every boosting iteration. Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.
- lambda: L2 regularization term on weights (analogous to Ridge regression). This is used to handle the regularization part of XGBoost. Increasing this value will make model more conservative.
- alpha: L1 regularization term on weights (analogous to Lasso regression).It can be used in case of very high dimensionality so that the algorithm runs faster when implemented.Increasing this value will make model more conservative.
- scale_pos_weight : It controls the balance of positive and negative weights. It is useful for imbalanced classes. A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence. A typical value to consider: sum(negative instances) / sum(positive instances).
- N_estimators: It determines the number of trees which we use in the ensemble.

In order to optimise these hyperparameters, a randomised search was utilised. A random search simply defines a search space bounded by the domain of hyperparamter values, and then randomly selects a combination of these values to use in the model. We then use these hyperparameter values to build a model, we split the training data into 2 sets, training and testing. We train the model with the hyperparameter values on the training data and validate its performance using the validation set. We enable the random search to do run this process for 25 iterations, and picked the set of hyperparameter values which produced the best cross-validation results.

Upon completion of this process, the following values were selected as optimal:

- eta/learning rate: 0.01
- max_depth: 20
- min_child_weight: 1
- subsample: 0.79
- lambda: 7
- alpha: 3
- scale_pos_weight : 7
- N_estimators: 100

We then trained an XGBoost model using these hyperparameters and the results were as follows:

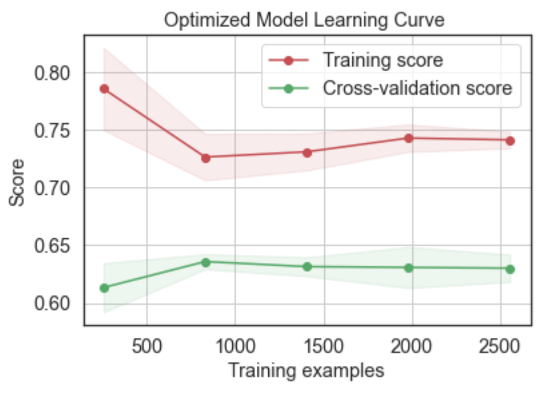


Figure 13: Optimised XGBoost Cross-Validation performance plot

We can clearly see from the figure above that once again the model has overfit for the optimal hyperparameter values. However, we do note that the degree of overfitting has decreased significantly compared to the basic model.

Confirmation that the optimised model has overfit to some degree again is shown in the test confusion matrix shown below. The model achieved a test accuracy of 62.2% which is much lower than the training accuracy. Another indicator is the fact that majority of the predictions are Class 1, which indicates poor generalisability from the mode.

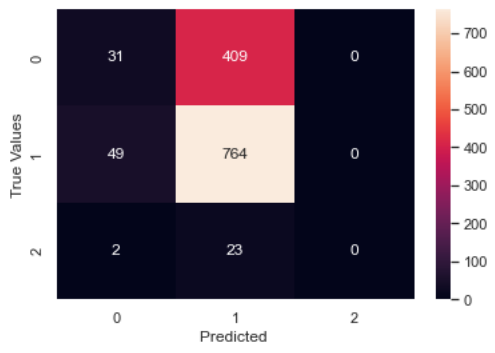


Figure 14: Optimised XGBoost Testing Confusion Matrix

However, given that we had a class imbalance, we need to consider the AUC metric in order to evaluate the true performance of the model.

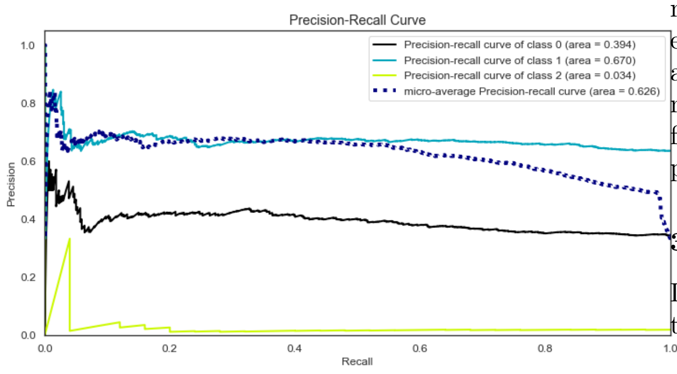


Figure 15: Optimised XGBoost Precision-Recall curve

From the plot above, if we look at the area underneath the curves, we see Class 1 (light blue) has the largest area since it was the class with the most data instances and the model thus can predict them relatively well. If we look at Class 0 (dark blue), we see that the curve decreases at the bottom and the area is lower than that of Class 1. This means that the model is not as good at predicting Class 0. Finally if we look at Class 2, we see the curve has an area of approximately 0 underneath. This is because the model predicted all instances incorrectly, but this is due to the fact that the model had not seen sufficient instances of this during training.

3.2 Neural Network

Neural networks are the most widely implemented machine learning models in real world applications. They entail a series of calculations which are utilised in order to discover trends within feature set, and ultimately find a relationship between these trends and the output. One of the strengths of Neural networks is their ability to make extremely accurate predictions given sufficient data. However, these models are complex and thus require a great deal of optimisation in order to achieve this improved performance.

3.2.1 General working of the algorithm

A simple neural network includes an input layer, an output (or target) layer and, in between, a hidden layer. The layers are connected via nodes, and these connections form a “network” of interconnected nodes. The nodes between layers are connected via weights, these weights dictate whether or not a node will be utilised or not. The main aim of neural networks is to find the weights between nodes such that the error in the predictions is minimised. This is achieved through a process called back-propagation. Essentially back-propagation computes the gradient of the loss function with respect to the weights of the network. In order to minimize the loss function various methods can be used, but the most utilised method is gradient descent. Gradient descent is an iterative first-order optimisation algorithm used to find a local minimum/maximum of a given function. In neural networks, we utilise it to minimise the cost function and thus maximise the models predictive power.

3.2.2 Terminology

In order to understand the following sections, certain terminologies which are utilised are clarified.

- Epoch: indicates the number of passes of the entire training dataset the machine learning algorithm has completed.
- Activation Function: defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network. It is used to add non-linearity to the data.
- Batch size: controls the number of training samples to work through before the model’s weights are updated.
- Neurons: refers to the nodes in a layer which are computation units.
- Optimiser: The algorithm which will be used to minimise the loss/error function.

3.2.3 Initial model without optimisation

The simple Neural Network architecture utilised consisted of 1 hidden layer with 10 neurons in it. The output layer consists of 3 neurons since we have 3 output classes, and a softmax activation function was used. The structure is denoted below.

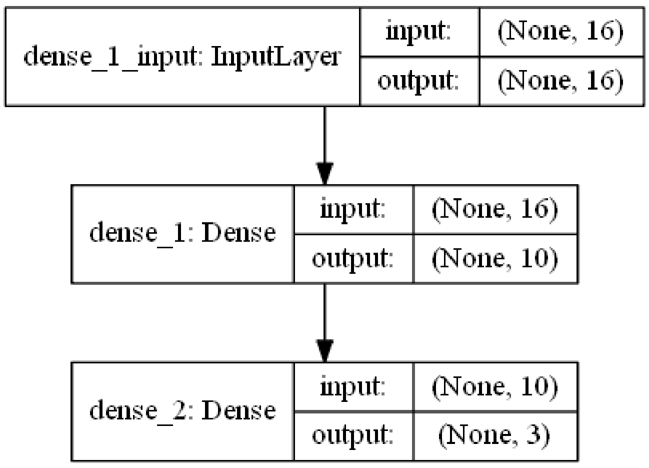


Figure 16: Simple Neural Network Architecture

For the activation function of the hidden layer, a sigmoid function was utilised and the weights were initialised using a random distribution. The optimiser used was Stochastic Gradient Descent which is explained in the next section. We allowed a maximum of 50 epochs and use a batch size of 100. These are relatively generic values for the hyperparameters.
The results of the basic Neural Network are as follows:

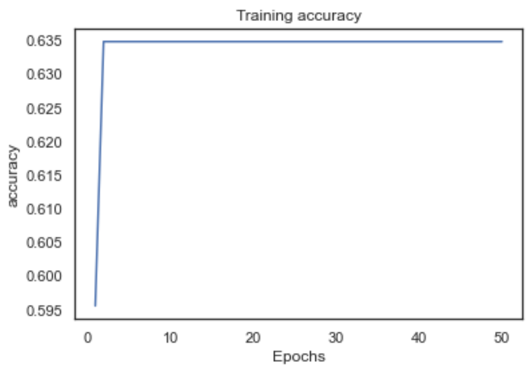


Figure 17: Simple Neural Network training accuracy graph

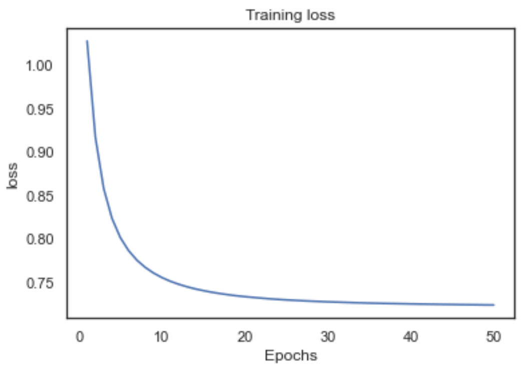


Figure 18: Simple Neural Network training loss graph

Looking at figure 12, we see that the model reaches an accuracy of approximately 63.5% and thereafter stops improving. From figure 13 we see that the loss function stops decreasing once it reaches a value of approximately 0.75. Both of these indicate that the model eventually stopped learning. Considering we utilised a sigmoid function and just randomly initialised the weights, learning could have stopped due to vanishing gradients. Since we utilise gradients in order to update our weights, when the gradients become extremely small such that they are approximately 0, no weight adjustment occurs and thus learning stops. This is the problem of vanishing gradients. If we look at the test accuracy, we see that the model achieved 63.62% accuracy. The confusion matrix is denoted below.

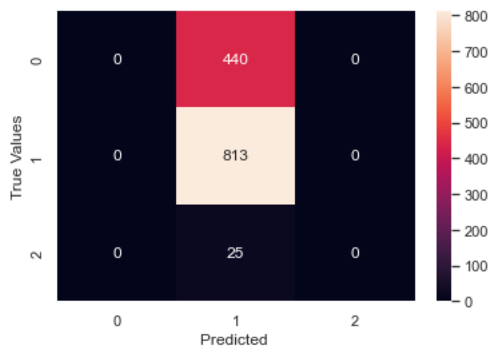


Figure 19: Simple Neural Network testing confusion matrix

From the confusion matrix we see that the model simply predicted class 1 for every instance, and since class 1 is the majority class, it achieves a decent accuracy. However, the fact that the model can only predict a single class is indicative that learning stopped early. In order to change this, various optimisations will have to be employed.

3.2.4 Optimising the model

In order to optimised the Neural Network, two sets of optimisations were ran both using a random search as explained in section 3.1.3.
The first set of hyperparameters which were optimised are:

- The initialization method to use.
- The optimization technique to use.
- The number of epochs to use.
- The best batch-size to use.
- The best activation function for the hidden layer.

Their are 2 main *initialisation* methods used:

- Glorot initialization:
  - This is the preferred method of initialization when the activation function is a Tanh or Sigmoid.
  - The weights are sampled using a uniform distribution or a normal distribution.
  - Using a uniform distribution, the weights are generated using the following formula: $W = U[-\sqrt{\frac{6}{N_{in}+N_{out}}}, \sqrt{\frac{6}{N_{in}+N_{out}}}]$, where N_{in} represents the number of neurons from the current layer and N_{out} represents the number of neurons in the layer the weights connect to.
  - Using a noram distribution, the weights are generated using the following formula: $W = G(0, \sqrt{\frac{6}{N_{in}+N_{out}}})$.
  - Note that $\sqrt{\frac{6}{N_{in}+N_{out}}}$ represents the Variance.
  - This ensures that the variance between weights is not large and relatively similar throughout the layers.
  - By keeping the variance the same across the various layers, we are able to combat the problem of vanishing and exploding gradients which hinders learning.
- He initialization:
  - This is the preferred method of initialization when the activation function is a ReLU.
  - The formulae is exactly the same as Glorot, the only different is that we multiply the variance by 2.
  - The reason we do this is because the ReLU turns half of the Z-values (the negative ones) into zeros, effectively removing about half of the variance. So, we need to double the variance of the weights to compensate for it.

There are 5 main optimisers that are commonly used:

- Gradient Descent:
  - Gradient Descent is the most basic but most used optimization algorithm. It's used heavily in linear regression and classification algorithms. Backpropagation in neural networks also uses a gradient descent algorithm.
  - Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function.
  - It calculates that which way the weights should be altered so that the function can reach a minima. Through backpropagation, the loss is transferred from one layer to another and the model's parameters also known as weights are modified depending on the losses so that the loss can be minimized.
- Stochastic Gradient Descent:
  - It's a variant of Gradient Descent that tries to update the model's parameters more frequently by making an update after each point is propagated through the network.
  - As the model parameters are frequently updated parameters have high variance and fluctuations in loss functions at different intensities. Thus we take steps in non-optimal directions. This leads to high variance amongst the model parameters which leads to a non-uniform convergence path to the minima.

  - However since we update the weights more frequently, convergence is faster. But the caveat is that we may overshoot the minima and diverge.
  - This method is preferred if the dataset is large since we don't require the entire dataset to be in RAM.
  - The better alternative is called ¡b¿Mini-Batch Gradient Descent¡/b¿. Which propagates a smaller batch through the network before making an update to the weights.
  - This combines the speed of SGD with the Uniformity of general GD.

- Adagrad:
  - In most other optimizers, the learning rate is kept constant for all the weights at every epoch.
  - This optimizer changes the learning rate. It changes the learning rate α for each parameter and at every time step 't'.
  - It works on the derivative of an error function and also works using the second order derivative.
  - α is a learning rate which is modified for given parameter θ_i at a given time based on previous gradients calculated for given parameter θ_i.
  - The update formula is as follow: $\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i}+\epsilon}}.g_{t,i}$, where $g_{t,i} = \nabla_\theta J(\theta_t, i)$.
  - We store the sum of the squares of the gradients w.r.t. θ_i up to time step t, while ϵ is a smoothing term that avoids division by zero.
  - It makes big updates for less frequent parameters and a small step for frequent parameters.
  - A problem with AdaGrad is that it can slow the search down too much, resulting in very small learning rates for each parameter or dimension of the search by the end of the run. This has the effect of stopping the search too soon, before the minimal can be located.

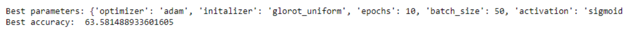
- RMSprop:
  - This is an extension of Adagrad.
  - In AdaGrad we take the cumulative summation of squared gradients but, in RMSProp we take the 'exponential average'.
  - An optimizer which uses a decaying average or moving average of the partial derivative in the calculation of the learning rate for each parameter.
  - Using a decaying moving average of the partial derivative allows the search to forget early partial derivative values and focus on the most recently seen shape of the search space.
  - The update formula is as follow: $\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{V_t+\epsilon}}.g_t$, where $V_t = \beta * V_{t-1} + (1-\beta) * g_t^2$, where $g_t = \nabla_\theta J(\theta_t)$.
  - Uses a moving average of squared gradients to normalize the gradient which is denoted by g_t^2.
  - This normalization balances the step size (momentum), decreasing the step for large gradients to avoid exploding and increasing the step for small gradients to avoid vanishing.

- Adam:
  - This optimizer is a combination of Gradient Descent with momentum and RMSprop.
  - Momentum is used to accelerate the gradient descent algorithm by taking into consideration the 'exponentially weighted average' of the gradients. Using averages makes the algorithm converge towards the minima in a faster pace.
  - By combining these two optimizations, Adam is able to converge much faster but is computationally expensive.
  - The update formula is given by: $\theta_{t+1} = \theta_t - \frac{\alpha * M_t}{\sqrt{V_t+\epsilon}}.g_t$, where $V_t = \beta_2 * V_{t-1} + (1-\beta_2) * g_t^2$ and $M_t = \beta_1 * M_{t-1} + (1-\beta_1) * g_t$, where $g_t = \nabla_\theta J(\theta_t)$.

Upon completion of the first round of optimisation, the following optimal values were found:

- The initialization method to use : Glorot.
- The optimization technique to use: Adam.
- The number of epochs to use: 10 .
- The best batch-size to use: 50.
- The best activation function for the hidden layer: Sigmoid.

The above is confirmed by the snippet below:



```
Best parameters: {'optimizer': 'adam', 'initalizer': 'glorot_uniform', 'epochs': 10, 'batch_size': 50, 'activation': 'sigmoid
Best accuracy:  63.581488933601605
```

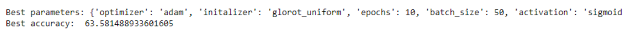
Figure 20: Optimised Snippet 1

Thereafter, another optimisation was run in order to optimise the architecture of the network. Since we are testing multiple hidden layers, we decided to set the activation function as a hyperparameter again

since the architecture will differ. We also know that the glorot initialiser works for the Tanh and sigmoid function, whilst He works for the ReLU and leaky ReLu, thus we took this into account when creating a layer.
The results of the second round are as follows:

- The number of hidden layers: 3.
- The neurons per hidden layer: 10.
- The best activation function for the hidden layers: Tanh.

The above results are shown in the snippet below:



```
Best parameters: {'optimizer': 'adam', 'initalizer': 'glorot_uniform', 'epochs': 10, 'batch_size': 50, 'activation': 'sigmoid
Best accuracy:  63.581488933601605
```

Figure 21: Optimised Snippet 2

Now that we have managed to optimise all the hyperparameters, we built a model utilising these hyperparameters.
The results of the optimised model are as follows:

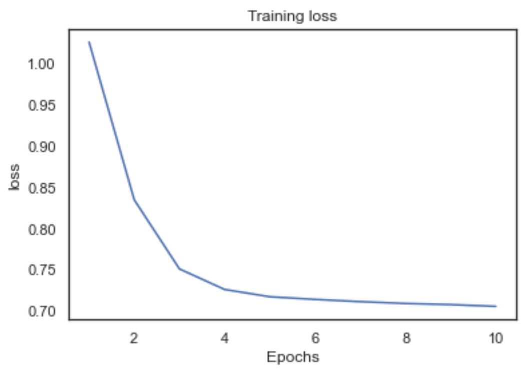


Figure 22: Optimal NN loss curve

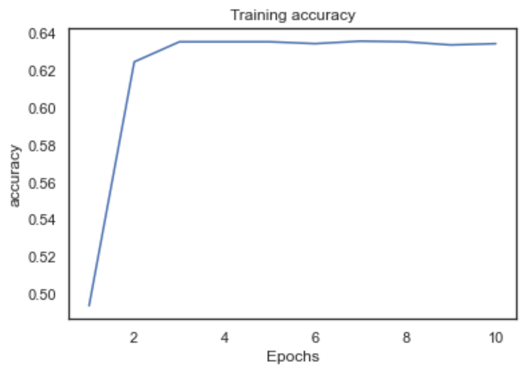


Figure 23: Optimal NN accuracy curve

From the curves above we can see again even though the model is optimised, we still end up reaching a threshold accuracy of approximately 64%. This lead us to believe that given the feature set used, this is the maximal accuracy achievable without overfitting the model. The cross-validation performance plot below,figure 19, is an indicator of the model reaching its maximum accuracy, since it shows that the model has yet to overfit since the training accuracy and cross-validation accuracy both increase at the same rate. However, at the end of the curve we notice that the curve begins to flatten which implies no further learning was occurring.

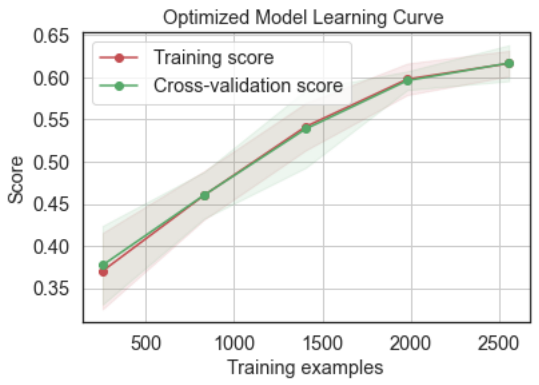


Figure 24: Optimal NN cross-validation performance plot

The test results of the optimal Neural Network are as follows:

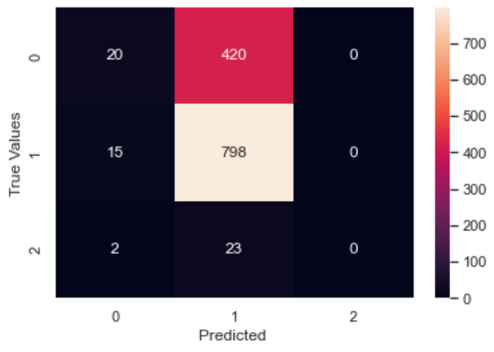


Figure 25: Optimal NN confusion matrix

From the confusion matrix above we can see that the model managed to achieve an accuracy of 64%, the highest we have managed to achieve without overfitting. We see that the model has learnt how to predict Class 0, however it still predicts Class 1 the best. This is simply because majority of the outcomes belong to Class 1. Proof is given by the precision-recall curve below, which further solidifies the sentiment that the class imbalances hinder the models performance.

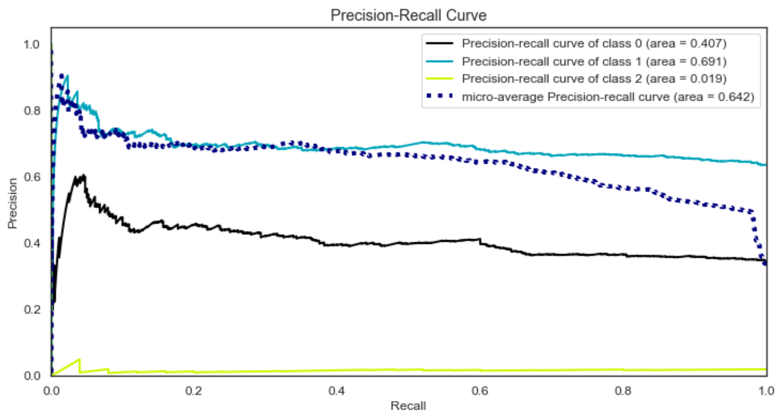


Figure 26: Optimal NN precision-recall plot

4 Conclusion

Given the reduced feature set and imbalanced classes, both XGBoost and the Neural Network managed to produced relatively good results. However, XGBoost had the tendency to overfit, whilst the Neural Network did not. Consequently, if we were to utilise a model to make predictions on unseen, new data, the Neural Network will be more reliable and provide more consistent classifications.