

Using machine learning classifiers to predict shot outcomes in association football

Author: Florian Neziri

Supervisor: Alessandro Proveti

MSc Data Science project report

Department of Computer Science and Information Systems

Birkbeck College, University of London, 2021

This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I have read and understood the sections on plagiarism in the Programme Handbook and the College web site. I give my permission for it to be submitted to the JISC Plagiarism Detection Service. The report may be freely copied and distributed provided the source is explicitly acknowledged.

## **Abstract**

This paper aims to apply machine learning classifiers to predict shot outcomes in association football. I will use open data from the football analytics company Statsbomb to form the basis of these models, cleaning the data and creating extra features, before applying three different feature selection techniques to eliminate any irrelevant features from the dataset: Feature Importance using Extra Trees Classifier, Mutual Information, and Recursive Feature Elimination. I will then use three different machine learning classifiers on each of the datasets produced after feature selection techniques have been applied, to create the predictions of shot outcomes: Logistic Regression, Classification and Regression Trees (CART), and K-Nearest Neighbours. Each model and dataset combination will be tuned, using both grid search and randomised search techniques, to find the optimal hyperparameters for each model, and then these models will be evaluated using metrics such: ROC AUC score, accuracy, precision, recall, and F1 score.

# Contents

<b>Abstract.....</b>	<b>2</b>
<b>1. Introduction.....</b>	<b>4</b>
<b>2. Data Pre-processing and Feature Selection .....</b>	<b>7</b>
<b>2.1 Data.....</b>	<b>7</b>
<b>2.2 Creating additional features .....</b>	<b>9</b>
<b>2.3 Data Pre-processing .....</b>	<b>11</b>
<b>2.4 Feature Selection .....</b>	<b>14</b>
<b>3. Project design and methodology .....</b>	<b>18</b>
<b>3.1 Machine learning classifiers.....</b>	<b>18</b>
<b>3.2 Hyperparameter tuning.....</b>	<b>21</b>
<b>4. Results and Analysis .....</b>	<b>25</b>
<b>4.1 Evaluation Metrics .....</b>	<b>25</b>
<b>4.2 Results without Tuning .....</b>	<b>26</b>
<b>4.3 Optimal Hyperparameters.....</b>	<b>30</b>
<b>4.4 Results using Optimal Hyperparameters from Tuning .....</b>	<b>34</b>
<b>5. Summary and Conclusion .....</b>	<b>39</b>
<b>Appendix A – Results of applying different feature selection techniques to the dataset .....</b>	<b>42</b>
<b>Appendix B – Hyperparameter tuning results and evaluation metrics pre and post tuning .....</b>	<b>45</b>
<b>Appendix C – Full Project Code.....</b>	<b>50</b>

## 1. Introduction

Over the last decade, football has slowly started embraced technological advances into the game - from the introduction of goal-line technology to determine whether the entire ball has crossed the goal-line, to the introduction of a Video Assistant Referee (VAR) to go over video replays of key incidents and determine whether or not there are grounds to potentially overrule the original decision of the on-field referee and, lastly, the increasing use of analytics and statistics to evaluate different aspects of player and team performance.

The overall aim of the game, however, has remained the same: two teams of 11 players aim to score more goals than the opposition across two halves of 45 minutes each. Football is generally a low-scoring sport with games across the top 5 leagues across Europe - in no particular order: English Premier League, French Ligue 1, German Bundesliga, Italian Serie A, and Spanish La Liga – averaging between 2.5 – 3.5 goals per game between the 2012/13 – 2019/20 seasons:

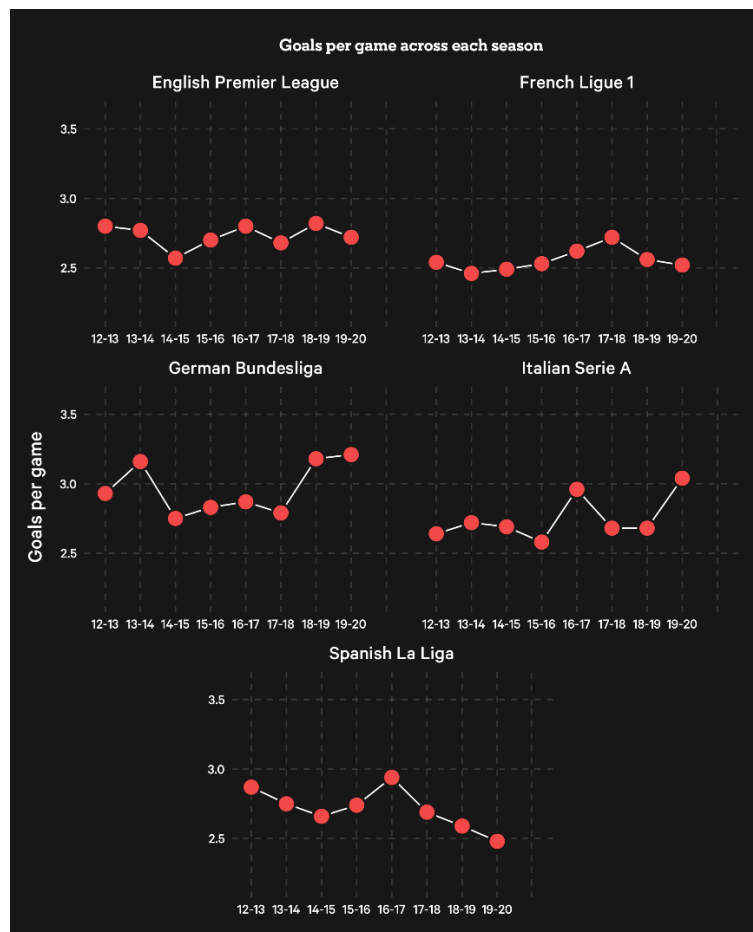


Figure 1: Goals per game in Europe's top 5 league [1]

Given the low-scoring nature of the sport, a lot of work is put into analysing the shots a team is taking to try and score, and how better to maximise the chance of a shot resulting in a goal. Expected goals (xG) is a measure that can give us an indication of how many goals a player or team should have scored on average, given the shots they have taken [2].

Expected goals measures the probability of a shot resulting in a goal based on numerous different variables such as: assist type (did the player taking the shot receive a pass on the ground or was it a lofted cross?, etc) shot angle, shot distance, and whether it was a headed shot (at the same distance, shots with feet are far more likely to become goals than shots with the head [3]). The xG model computes for every shot taken the probability to score based on these variables, with a higher xG denoting a higher probability of scoring, with probabilities ranging from 0 (minimum) to 1 (maximum) [4].



Figure 1: a visual example of the application of xG when analysing shots taken by Dominic Calvert-Lewin, striker for Everton FC and England [2].

In terms of its application, it is generally not too insightful to look at xG on a shot-by-shot basis. What is far more useful is adding up a player or team's xG during a game, a longer period, or a whole season (or seasons) to give a better idea of how many goals they should have scored, based on the quality of shots they had. When you have a sample to work from, spanning a longer period of time, we can use xG to deduce whether a certain player or team is under or over-performing, in terms of scoring goals, compared with their expectation [5]. Hence this makes xG a powerful long-term measure of effectiveness of players when it comes to taking shots and scoring goals from these shots.

This project aims to build a machine learning classifier that, given certain variables about a shot, will be able to reasonably predict whether or not this shot is likely to result in a goal. We will apply different feature selection techniques on an initial dataset to select those features that will contribute the most to the likelihood of a shot resulting in a goal, before applying different machine learning classifiers to the new datasets and evaluating the relative performance of each model on the data. These models will be tuned to find the optimal hyperparameters for each classifier.

## **2. Data Pre-processing and Feature Selection**

### **2.1 Data**

Technological advances have allowed data companies to capture up to 2,000 events per games [6] to pass on to match commentators and viewers. However, despite these vast improvements in technology and data capture, it is incredibly difficult to obtain datasets that are useful for analysis as these detailed data are owned by specialized companies and hence are rarely publicly available for scientific research [7].

The data that I will be using in this project is open data that comes from the analytics company Statsbomb. This dataset contains event data for specific matches played in the men's Spanish La Liga between the 2004/2005 and 2019/2020 seasons, with additional competitions including: the men and women's FIFA World Cups in 2018 and 2019 respectively, the 2018 USA National Women's Soccer League season, the 2018/2019 and 2019/2020 seasons from the English Women's Super League, the 2003/2004 men's English Premier League and, lastly, various fixtures from the men's Champions League between the 1999/2000 and 2019/2020 seasons.

To control for the difference in the quality of different leagues across Europe between men and women's football, I shall be using just the data from the La Liga seasons. Statsbomb does not provide a full season's worth of data for each of the 16 La Liga seasons it covers, instead selecting just the matches that Barcelona's former Argentinian forward and captain Lionel Messi featured in. During these 16 seasons, Barcelona won 10 championships, with Messi personally winning the prestigious Ballon d'Or (Golden Ball) award in 6 of these seasons - a record for any player – which is an award given for the men's football player perceived to have performed the best among his peers across world football, voted for by football journalists, coaches, and captains of national teams [8].

There is a risk of overfitting the models as the Statsbombs data covers a team and players – specifically Messi – at the very peak of their powers. This means that the quality of shooting could be higher than you would expect when comparing to fellow footballers across the 20 teams in the Spanish division (and further afield in Europe), which could inflate the expected

goals value of a shot and the likelihood of our classifiers deciding that a shot will result in a goal. However, the total number of shots that were taken by Messi in this dataset is 2,126 which, despite being 1,583 more shots than the second-highest number of shots taken by a player in the dataset (Uruguay's Luis Suarez with 579), make up just 18% of all shots taken. Whilst this is still a much larger total than any other player, any effects of Messi's superior technical ability should not be significant.

I did consider an alternative data source for this project, which was part of a public dataset of spatio-temporal events from football matches, collated by Pappalardo et al. in 2019 [9] from Wyscout; a platform providing various videos, data, statistics and tools to analyse teams, matches and players. As well as spatio-temporal data such as shots, passes, and fouls, this dataset contained match events such as position, time, match outcome, player, and characteristics from the 2017/18 season in the top 5 leagues of European club football - Premier League (England), Ligue 1 (France), Bundesliga (Germany), La Liga (Spain), Serie A (Italy) - as well as 2 international tournaments in the 2018 FIFA World Cup and 2016 European Championship.

Ultimately I decided to use the Statsbomb dataset, owing to the far wider range of variables available in this dataset compared to the Pappalardo dataset. Variables such as: technique used for the shot, body part the shot was taken with, whether the shot was taken in open play or from a range of different set pieces, offer a far greater level of contextual information in constructing an accurate description of a shot and therefore its likelihood in resulting in a goal, which the Pappalardo dataset does not do. Whilst the Statsbomb data doesn't offer the full data in a season like Pappalardo does, the greater level of detail the Statsbomb data offers is far more valuable for the purposes of creating an accurate machine learning classifier. We will have 12,000 shots to analyse across 485 different games (which is greater than the 380 games across just one season of La Liga).



## 2.2 Creating additional features

Before using the data to train our machine learning models, we must first create some new features in the dataset: distance to goal and angle of shot. Intuitively, the closer you are to the goal, the higher your chance of scoring and, similarly, the wider your angle to the goal, the more of the goal you can see and aim for, subsequently increasing the probability that your shot will result in a goal.

The Statsbomb data provides us with pitch coordinates (in yards) for the location of each shot. However, the effect of location on the likelihood of a shot resulting in a goal can be broken down into a combined effect of angle and distance; without separating these two effects, it is difficult to interpret the relationship between location and expected goals (other than being able to state that location has *some* effect).

The dimensions of the pitch we are using in this model: 120 yards long by 80 yards wide.

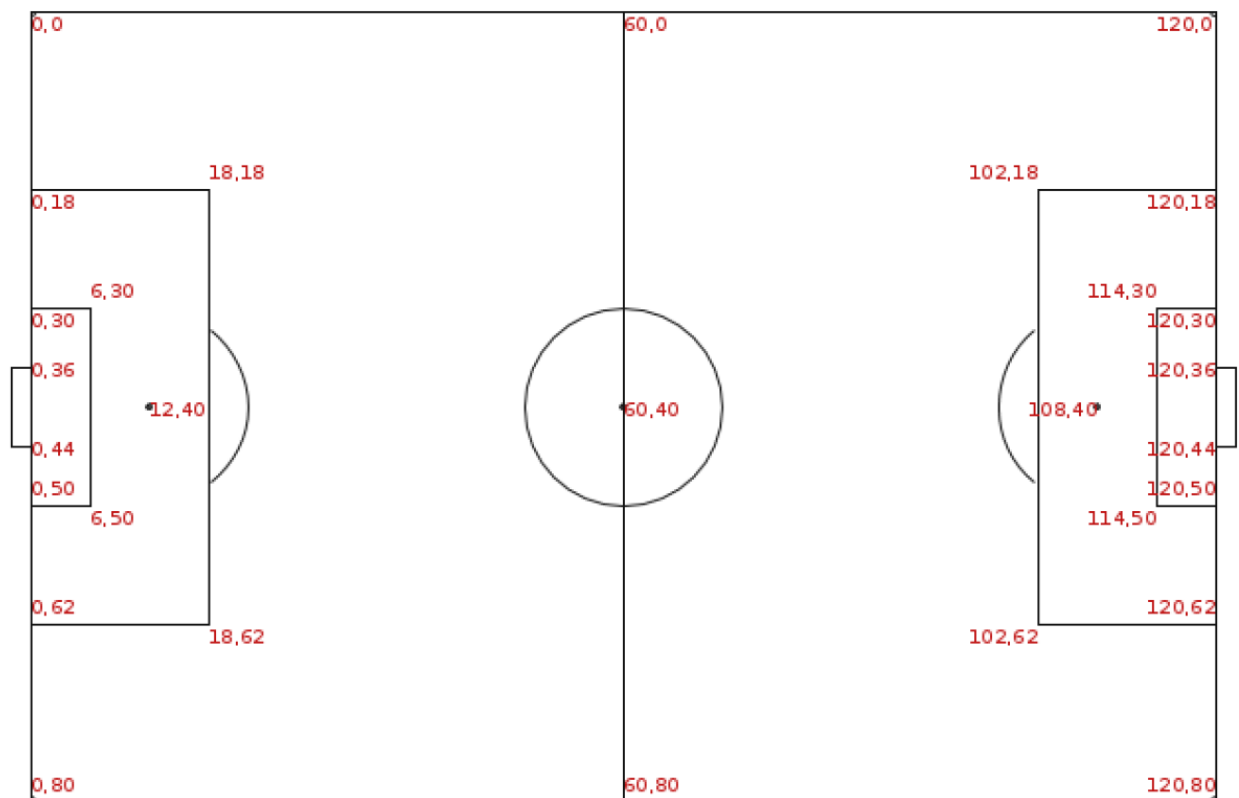


Figure 1: dimensions of a football pitch based on Statsbomb data [10]

An x-coordinate of 120 is the closest a player can be to the opposing goal - the goal that they are trying to score in. Therefore, the higher the x-coordinate of a player's location, the closer they are to the opposing goal. Y-coordinates of 0 or 80 are the furthest the player can be from the centre of the pitch, with an y-coordinate of 40 being the midpoint of the width of the pitch and the most central position a player can take up. We can use these coordinates to calculate the Euclidean distance between the shot location and the middle of the goal (120,40).

$$\text{Shot distance (yards)} = \sqrt{(120 - x)^2 + (40 - y)^2}$$

Shot angle is calculated by drawing a triangle with vertices at either post on the goal and a vertex from the coordinates the shot was taken from.

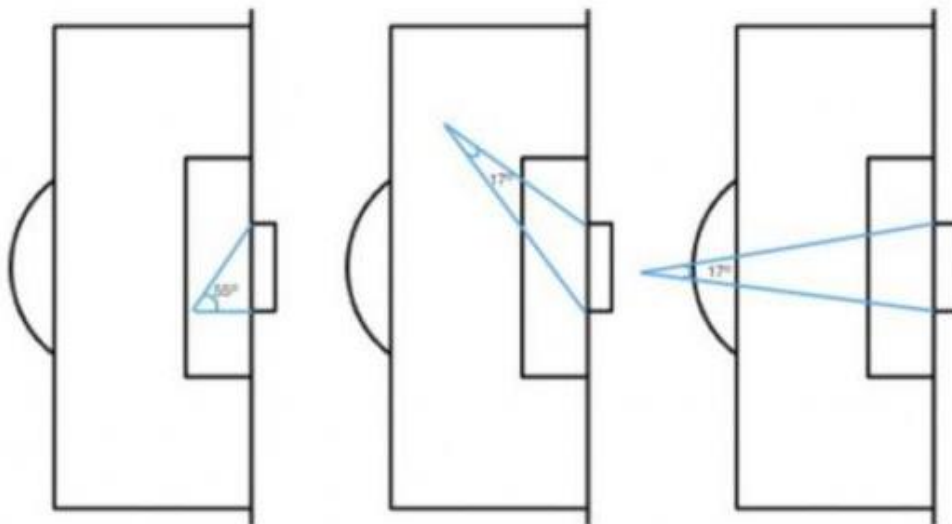


Figure 2: examples different shooting angles [11]

We can use some straightforward trigonometry (cosine rule) to calculate the angle of interest - the latter of the vertices mentioned – from where the shot was taken.

$$\text{Shot angle} = \frac{8(120-x)}{(120-x)^2 + (y^2 - 80y + 1584)}$$

Where x and y are the coordinates of the shot location.

### 2.3 Data Pre-processing

Before applying feature selection techniques to reduce number of variables in the dataset, we can firstly manually remove features that we know intuitively will have no bearing on the shot outcome. Firstly, we can drop any variables that are purely used as identifiers (different types of IDs such as match id, competition id, season id, etc). Secondly, we can drop any variables that are related to another type of event in the game (e.g. pass end location, carry (dribble) end location, possession, etc). Lastly, we can remove shot variables that won't have an impact on the expected goals value as the variables are related to events happening after the shot is taken (e.g. shot end location, shot saved off target, shot saved to post, etc).

In addition to the variables above, we can also remove variables that would be highly correlated with another. For example, we can now drop the *location* variable as we have split this into distance and angle variables, and the *play pattern name* variable as the values are very similar to the *shot type name* variable. We can also remove variables that essentially tell us the same information (e.g. removing one of shot technique id and shot technique name or body part id and body part name).

Once we have done this, we can change the values of some the variables from descriptors to Boolean variables 0 and 1. We can do this using a combination of one-hot encoding – which creates a new binary column for each value in a variable – and custom binary transformation based on knowledge of the wider context of the dataset.

The *shot technique name* variable contains the values:

Shot Technique	Description
Backheel	A shot that was taken with the heel
Diving Header	A shot attempted with a header and the player diving to reach the ball
Half Volley	Contact was made off the ground and after a bounce
Lob	A shot with a high arc trajectory to pass over the opposition player
Normal	A shot that does not fall into any other technique
Overhead Kick	Player's back was to goal when taking the shot
Volley	The ball never touched the ground prior to the shot

Table 1: Description of values for the *shot technique name* variable [12]

We used one-hot encoding to convert this from one variable with 7 different values, to 7 separate variables in separate columns, each with the value 1 or 0 denoting whether or not that specific shot technique was used or not. For example, if the shot was a volley, the shot technique name volley column would have 1 in it and all the other shot technique name columns that were produced through one-hot encoding will have 0.

The *shot body part name* variable contains the values:

Body Part	Description
Head	Shot attempted with head
Left Foot	Shot attempted with the left foot
Other	Other body parts (i.e knee, chest, etc)
Right Foot	Shot attempted with right foot

Table 2: Description of values for the *body part name* variable [12]

For this variable we used a combination of custom binary encoding and one-hot encoding. Instead of one-hot encoding so that each variable has its own column, we first combined the left foot and right foot columns into a new column simply called *shot body part foot*. We did this because players predominantly have one foot they use regularly – their ‘stronger’ foot – with the other considered their ‘weaker’ foot. Given that the majority of footballers are right-footed (similar to how the majority of the general population is right-handed), the majority of shots taken with the left foot will be shots taken with the weaker foot, so the model may wrongly make the assumption that, for *all* players, shooting with your left foot decreases the expected goals value, even if the player is left-footed.

Ideally, we would like extra data on the player to understand which foot is their strongest, in addition to which foot their shot was taken with – the logic being that a shot for a player with their stronger foot would have a higher expected goals values than the same shot taken with their weaker foot. We could then replace the left foot and right foot values with ‘strong’ foot. Instead, we will focus on the different effects shooting with your foot compared to other body parts have, so after combining left foot and right foot into one variable *shot body part foot*, we one-hot encode the *shot body part* variable to give us 3 new columns: *shot body part foot*, *shot body part head*, and *shot body part other*.

The *shot type name* variable contains the values:

Shot Type	Description
Corner	Shot direct from a corner kick
Free Kick	Shot is from a direct free kick
Open Play	Shot is not directly from a set-piece
Penalty	Shot is a penalty kick
Kick Off	Shot directly from kick off

Table 3: Description of values for the *shot type name* variable [12]

This variable has been transformed using one-hot encoding, so each value for this variable now has its own column, with 1 denoting this variable was true for the shot in question and all other shot type variable columns containing 0. Note: when applying this technique, there was no column for kick off – no shots in this dataset were taken directly from kick off – so we can discard this value.

Lastly, the *shot outcome name* variable will be our y variable in the dataset - the variable we are measuring - and it contains the values:

Shot Outcome	Description
Blocked	A shot that was stopped from continuing by a defender
Goal	A shot that was deemed to cross the goal-line by officials
Off T	A shot that's initial trajectory ended outside the posts
Post	A shot that hit one of the three posts
Saved	A shot that was saved by the opposing team's keeper
Wayward	An unthreatening shot that was way off target or did not have enough power to reach the goal line (or a miskick where the player didn't make contact with the ball)
Saved Off T	A shot that was saved by the goalkeeper but was not on target.
Saved To Post	If the keeper saves the shot and it bounces off the goal frame

Table 4: Description of values for the *shot outcome name* variable [12]

This variable will be renamed to *goal*, with a custom binary encoding telling us whether a shot resulted in a goal (1) or not (0), as this is the only shot outcome we are interested in.

After we have removed these values and our dataset is ready to analyse, we must also standardise the variables in the dataset so that they all have mean 0 and standard deviation 1. We can do this by using the *StandardScaler* class of Scikit-Learn's preprocessing library. Once we have standardised the features, we can split the dataset into train and test datasets – we will be using a split of 0.33 for our test dataset.

## **2.4 Feature Selection**

Once we have reduced the dataset to just variables that we know will have some sort of impact on shot outcome, our next aim is to discern which features in the data contribute the most to the probability of a shot resulting in a goal, and therefore the xG number attached to that shot, through applying different feature selection techniques. The main benefits to using feature selection techniques on the dataset prior to training the classifiers are threefold:

1. Training time decreases: fewer variables in the dataset means there is less data to train our chosen classifiers on, meaning the chosen algorithms train faster
2. Overfitting decreases: less redundant data means there are fewer opportunities for the classifiers to make decisions based on noise
3. Accuracy increases: modelling accuracy increases due to less misleading data

Features selection techniques can generally be split into 3 different categories: filters, wrappers, and embedded methods. Filters select subsets of variables as a pre-processing step, independently of the chosen predictor, and select features based on some classifier criterion (in our case this is Mutual Information which is explained later); wrappers assess subsets of variables according to their usefulness to a given predictor; and with embedded methods, feature selection is integrated into a given learning algorithm and performs in the training process [11][12].

A main characteristic of wrapper methods is that they tend to be more computationally expensive than filter and embedded methods because they have to train and test the classifier for each feature subset candidate (filters tend to be the least intensive with embedded methods somewhere in the middle of the two). Additionally, the performance of wrapper and

embedded methods are optimised directly by the selected learning method(s) [12]. Filters are generally less computationally expensive than wrappers, but at the expense of producing feature sets which are not tuned specific predictive models.

To control for the different ways in which these 3 types of feature selection techniques select features, we shall use techniques from all 3 categories: Mutual Information (filter), Feature Importance using Extra Trees Classifier (embedded), and Recursive Feature Elimination (wrapper). We will then train and tune our different models on the three datasets produced by the respective feature selection techniques, and evaluate each model and feature selection technique pairing to see which has the best performance in predicting shot outcomes.

Our first feature selection is the filter method Mutual Information. Mutual Information is a quantity that measures the mutual dependence between two random variables. That is, how much information one random variable has about another [13]. Formally, the mutual information between two random variables X and Y can be stated as:

$$I(X ; Y) = H(X) - H(X | Y)$$

Where  $I(X ; Y)$  is the mutual information for X and Y,  $H(X)$  is the entropy for X and  $H(X | Y)$  is the conditional entropy for X given Y, and the measure is symmetrical:  $I(X ; Y) = I(Y ; X)$  [14]. Mutual information measures the average reduction in uncertainty about x that results from learning the value of y; or vice versa, the average amount of information that x conveys about y [15].

When we apply Mutual Information feature selection to our ‘clean’ dataset (after I have removed variables that I have deemed to have no bearing on shot outcomes or are strongly related to other variables), the variables that are deemed to be unimportant to a goal being scored (based on having an MI score of 0) are: *shot.follows\_dribble*, *shot.technique.name\_Half Volley*, *shot.technique.name\_Normal*, *shot.technique.name\_Volley*, *shot.body\_part.name\_Foot*, *shot.type.name\_Corner*, *shot.type.name\_Open Play* (see table 1 in Annex A for full list of MI scores). This leaves us with 18 variables from the original 25 variables in the dataset before we applied feature selection.

Another feature selection technique we will be using is Feature Importance, using an Extra Trees classifier, which is an embedded method. The Extra-Trees algorithm builds an ensemble of unpruned decision trees, using the whole learning sample to grow the trees rather than a bootstrap replica as in other tree-based ensemble methods (e.g. random forests) in order to minimise bias. Another difference with other tree-based ensemble methods is that nodes are split by choosing cut-points fully at random, instead of looking for the most discriminative thresholds to split nodes [16].

Extra Trees has two parameters:  $K$ , the number of attributes randomly selected at each node and  $n_{min}$ , the minimum sample size for splitting a node. It is used several times with the (full) original learning sample to generate an ensemble model. The predictions of the trees are aggregated to yield the final prediction, by majority vote (in classification problems such as ours). The randomisation of the cut-point and attribute, combined with ensemble averaging, is used to reduce variance more strongly than the randomisation used by other methods [16].

When we apply Feature Importance feature selection using the Extra Trees Classifier to our clean dataset, the variables that are deemed to be unimportant to a goal being scored (based on having the lowest FI scores across the 25 original variables) are: *shot.aerial\_won*, *shot.technique.name\_Diving Header*, *shot.technique.name\_Overhead Kick*, *shot.technique.name\_Backheel*, *shot.follows\_dribble*, *shot.body\_part.name\_Other*, *shot.type.name\_Corner* (see table 2 in Annex A for a full list of FI scores). This again leaves us with 18 variables from the original 25 variables in the dataset before we applied feature selection.

The last feature selection technique we will apply to the dataset is Recursive Feature Elimination (RFE) – a wrapper method. Given an external estimator that assigns weights to features (in this scenario we are using a Random Forests Classifier estimator), RFE aims to select features by recursively considering smaller and smaller sets of features [17].

Backwards selection techniques such as RFE are commonly used with random forest models for two reasons: random forest tends not to exclude variables from the prediction equation and random forest has a well-known internal method for measuring feature importance [18].



RFE begins with the estimator being trained on the full set of features and computing an importance score for each predictor. The least important predictor(s) are then removed, the model is re-built on the smaller dataset without the least important predictors, and importance scores are computed again. This is repeated until we reach the desired number of features which we can specify - RFE uses the model accuracy to identify which attributes (or combination of attributes) contribute the most to predicting the target attribute. In this feature selection, however, I have decided to use the RFECV version of recursive feature selection in Scikit Learn's feature selection module, which implements RFE with cross-validation to determine the best number of features to use.

When we apply RFE to our clean dataset, cross-validation reduces the dataset from 25 variables to 18 variables. The variables deemed not to be important to a goal being scored are: *shot.type.name\_Free Kick*, *shot.type.name\_Corner*, *shot.technique.name\_Backheel*, *shot.technique.name\_Diving Header*, *shot.follows\_dribble*, *shot.technique.name\_Overhead Kick*, *shot.body\_part.name\_Other*. (See table 3 in Annex A)

### 3. Project design and methodology

#### 3.1 Machine learning classifiers

The three machine learning classifiers we will be using to predict the outcomes of shots are: Logistic Regression, CART (Classification and Regression Trees – classification in this case), and K-Nearest Neighbours.

We will use logistic regression as it is commonly used to estimate the probability that an instance belongs to a particular class and in our model we are interested in finding out, based on our input data for a shot, whether a shot belongs to the *positive* class of goal (labelled 1) or *negative class* of no goal class (labelled 0) [19]. If the estimated probability of a shot resulting in a goal is greater than some threshold probability value (by default 0.5), then the model predicts that this shot belongs to the positive class, and otherwise it predicts that it belongs to the negative class. A logistic regression model does this by calculating a weighted sum of input features (plus bias) and outputting the logistic of this result (as opposed to, for example, linear regression which outputs this value directly). The logistic regression model estimated probability in vectorised form is:

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\mathbf{x}^T \boldsymbol{\theta})$$

The logistic is an s-shaped sigmoid function which outputs a number between 0 and 1:

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

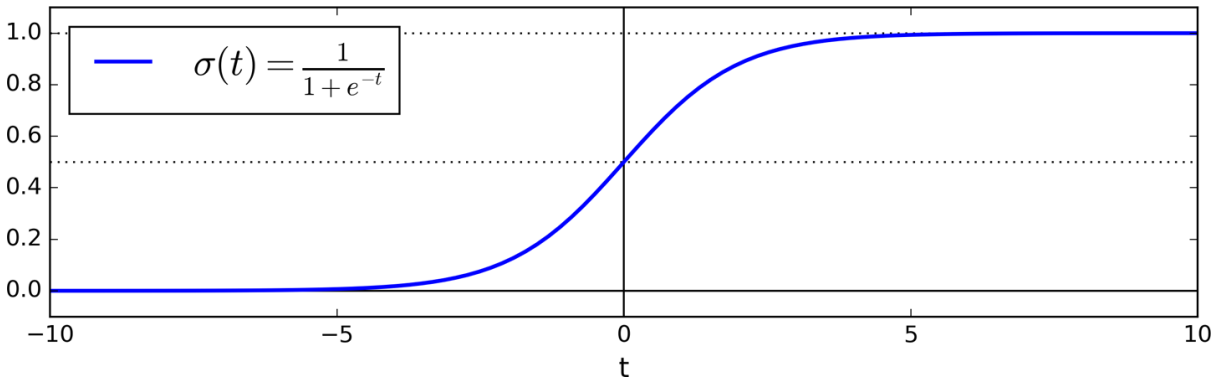


Figure 3: Logistic function [19]

The score  $t$  in the logistic equation above is often called the *logit* which comes from the logit function, which is the inverse of the logistic function, defined as:

$$\text{logit}(p) = \log \frac{p}{(1-p)}$$

Once the probability  $\hat{p} = h_{\theta}(\mathbf{x})$  that an instance  $\mathbf{x}$  belongs to our positive class (goal scored) has been estimated by the logistic regression model, the model can then easily predict  $\hat{y}$ :

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

The next model we will be looking at is Classification and Regression Trees (CART) which is the algorithm the Scikit-Learn package in Python that we are using uses to train Decision Trees. The CART algorithm works by first splitting our training set into two subsets using a single feature  $k$  in our dataset and a specific threshold value  $t_k$  that determines into which one of the two binary classes produced that specific shot will fall into. The CART algorithm produces only binary trees – non-leaf nodes always have two children: one corresponding to the ‘yes’ branch of the condition specified node, the other corresponding to the ‘no’ branch.

The algorithm chooses  $k$  and  $t_k$  such that the pair  $(k, t_k)$  produces the purest subsets (weighted by size) – a node is pure when all training instances it applies to belong to the same class - and that minimises the following cost function [20]:

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where

$$\begin{cases} G_{\text{left/right}} & \text{measures the impurity of the left/right subset} \\ m_{\text{left/right}} & \text{is the number of instances in the left/right dataset} \end{cases}$$

As the CART algorithm successfully splits the training data into two subsets, it repeats this process recursively in the same way as it did initially; the next subsets produced are then also split into two and so on. The algorithm stops recursing once it has reached the maximum depth (a hyperparameter of the algorithm) or if the algorithm can no longer produce a split that will reduce impurity.

This makes CART a greedy algorithm: it searches for the optimum split at each level without checking whether this specific split will or will not contribute to producing the lowest level of impurity several levels further down. This will lead to a reasonably good solution but not one that is guaranteed to be optimal. Despite this, however, there are several benefits to the CART algorithm and using decision trees that contribute to choosing this model for analysis in this project. Possibly the most important benefit of CART in the context of this project is the ease of understanding of the model [21]. In an industry such as football that has long been resistant to statistics and analysis (and has only recently been adopting the use of both), having a model that is simple to explain – a series of yes/no decisions to produce the final separation of classes – is invaluable. In addition, decision tree methods are well suited for data in instances where we do not know ahead of time which variables are important predictors – it may help us uncover relationships which might be masked by other more computationally intensive methods [21].

The last machine learning classifier we will be looking at is k-nearest neighbours. Neighbours-based classification is calculated through a simple majority vote of the nearest neighbours of each point: a query point is assigned the data class which has the most representatives within the nearest neighbours of the point. Since the neighbours are nearby, they are likely to be similar to the point being classified and so are likely to be the same class as that point. The ‘closeness’ of the neighbours is defined by the difference (distance) along the scale of each variable, which is converted to a similarity measure [22]. The distance measure traditionally used is Euclidean distance but, as distance is a hyperparameter of the k-nearest neighbour classifier, we can tune this later on to see whether other measures of distance (Manhattan or Minkowski) are better to use.

K-nearest neighbours implements learning based on the k-nearest neighbours of each query point, where k is an integer value specified by the user. The best choice of k depends largely on the data. In general, larger values of k tend to create larger classes in terms of the range of values included in the class, which has the effect of reducing the noise in the classification but makes the classification boundaries less distinct [22][23]. Like the CART classifier, the simplicity of this model is a main reason for using it as one of our classifiers, which is very useful for applying it to a football context. In addition, KNN’s ability to adapt to new data introduced into the dataset is a significant advantage: it will make applying the classifier to new shots we want to analyse in the future very easy.

### *3.2 Hyperparameter tuning*

The machine learning classifiers described in the previous section will be tuned using two different techniques: grid search and randomised search. Hyperparameter tuning using the grid search technique methodically builds and evaluates a model for every single combination of algorithm parameters that is specified in a grid. Hyperparameter tuning using a randomised search technique, however, does not build and evaluate a model on all combinations of algorithm parameters specified. Instead, a randomised search will sample algorithm parameters from a random distribution for a fixed number of iterations – in the randomised search function I have created, this has been set at 10. Then, for each combination of parameters that are sampled from the random distribution, a model is constructed and evaluated [24].

For logistic regression, the hyperparameters we will be tuning using both grid search and randomised search are: penalty – which is used to regularise the model; C – which controls the regularisation strength (penalty) of the model - a higher value of C means a less regularised model); and solver - the algorithm used in the optimisation problem.

The grid search for our logistic regression classifier will be set up with five different values of C: 0.01, 0.1, 1, 10, 100, two different penalties: l1 and l2, and three different solver algorithms: saga, liblinear, and lbfgs. The penalty specifies the regularisation term to be added to the cost function in the regression. For l1, this is the absolute value of the l1 norm of the vector of feature weights but, for l2, this is half of the square of the l2 norm of the weight vector [19]. For the solver hyperparameter, saga is a variant of the ‘sag’ solver which uses stochastic average gradient descent but, unlike sag, is able to use the l1 penalty. Liblinear uses a coordinate descent algorithm, and relies on the LIBLINEAR library. Lbfgs is an optimization algorithm that approximates the Broyden–Fletcher–Goldfarb–Shanno algorithm [25].

Our randomised search will be randomly selecting a value of C between 0.01 and 100. We will be using the log uniform distribution between 0.01 – 100 to provide a continuous probability distribution, which means that sampling with replacement will be used when running the randomised search. The randomised search will also choose random values for penalty and solver, selecting from the same list for each variable as in the grid search.

For the CART classifier, the hyperparameters we will be tuning using both grid search and randomised search are: criterion – the function that measures the quality of a split, the maximum number of features that are evaluated for splitting at each node, the minimum number of samples a node must have before it can be split, and the minimum number of samples a leaf node must have [20].

The grid search for our CART classifier will be set up with two different values for criterion: entropy and gini, two different values for max\_features: auto and log2, eight different values for min\_samples\_split: ranging from 2 to 9, and nine different values for min\_samples\_leaf: ranging from 1 to 9. For criterion, we have gini and entropy: gini impurity is the probability of incorrectly classifying a randomly chosen element in the dataset if it were randomly labelled according to the class distribution in the dataset [26]. It's calculated as:

$$G = \sum_{i=1}^C p(i)(1 - p(i))$$

where  $C$  is the number of classes and  $p(i)$  is the probability of randomly picking an element of class  $i$ . Entropy can also be used as an impurity measure: a set's entropy is zero when it contains instances of only one class. The equations for entropy is [27]:

$$E = - \sum_{i=1}^C p(i) \log_2 p(i)$$

For the max\_feature hyperparameter in the CART algorithm: auto means that max\_features is set to  $\sqrt{n\_features}$  and log2 means that max\_features is set to  $\log_2(n\_features)$ . Our randomised search will be choosing random values for criterion and max\_features, selecting from the same list for each variable as in the grid search. It will also be choosing random values for min\_features\_split and min\_features\_leaf, ranging from 2 to 9 and 1 to 9 respectively, however, unlike the randomised search for logistic regression, we will not be using a continuous distribution as these two hyperparameters only take integer values. The randomised search function will choose integers at random between the respective ranges for the hyperparameters.

For our k-nearest neighbours classifier, the hyperparameters we will be tuning using both grid search and randomised search are: the number of neighbours to use, the weight function used in prediction, and the metric (distance) metric used for the tree.

The grid search for our k-nearest neighbours classifier will be set up with nine different values for number of neighbours: ranging from 1 to 9, two different weights: uniform and distance, and three different distance metrics: Euclidean, Manhattan, and Minkowski. For the weight hyperparameter, uniform weight means all points in each neighbourhood are weighted equally, whereas distance weights points by the inverse of their distance (closer neighbours of a query point will have a greater influence than neighbours which are further away). For our metrics (distances), Euclidean distance is defined in an n-dimensional space as:

$$D_E = \sqrt{\left(\sum_{i=1}^n (p_i - q_i)^2\right)}$$

Manhattan distance is defined in an n-dimensional space as:

$$D_{MA} = \sum_{i=1}^n |p_i - q_i|$$

Where, n = number of dimensions and  $p_i, q_i$  = data points. Lastly, Minkowski distance is defined in an n-dimensional space as:

$$D_{MI} = \left(\sum_{i=1}^n |p_i - q_i|^p\right)^{\frac{1}{p}}$$

Where p represents the order of the norm. The Minkowski distance can be considered a generalisation of both the Euclidean and Manhattan distances and is equal to the Euclidean distance when p=2 and equal to the Manhattan distance when p=1.

Our randomised search will be choosing random values for number of neighbours ranging from 1 to 9 and, like with our CART tuning, we will not be using a continuous distribution as number of neighbours must be an integer value. The randomised search function will choose integers at random between 1 and 9. It will also be choosing values for metric and distance, selecting from the same list for each variable as in the grid search.

The aim of using these tuning techniques on each machine learning classifier model is to find the set of hyperparameters that will produce the optimal performance for each model. This will, in turn, contribute to this project's aim of creating an accurate model of predicting shot outcomes.

Based on the descriptions above, it is clear that grid search can be seen as a far more exhaustive search than randomised search, as it encompasses every combination of parameters specified. Randomised search takes a fraction of the time and resources required by grid search but there is the possibility that the hyperparameters chosen will produce a model with lower accuracy, due to randomised search not evaluating every single possible combination of parameters or as many combinations as grid search. This can be seen in the different number of iterations between grid search and randomised search for each model. The randomised search function has been set to run for 10 iterations on each model whereas for grid search the combinations of hyperparameters are: 30 for our logistic regression classifier (5x2x3), 288 for our CART classifier (2x2x8x9), and 54 for our k-nearest neighbours classifier (9x3x2). It will be interesting to see how much of a difference the extra runs for each model in our grid search will make to the accuracy score produced.



## 4. Results and Analysis

### 4.1 Evaluation Metrics

We will be using the Area Under the Receiver Operating Characteristic Curve (ROC AUC) metric to evaluate our different models. The ROC curve plots the true positive rate (recall or sensitivity) against the false positive rate (FPR): the ratio of negative instances that are incorrectly classed positive. Intuitively, the FPR is equal to:  $1 - \text{the true negative rate (TNR)}$ , which is the ratio of negative instances that are correctly classified as negative, and is also known as the specificity. Hence, the ROC curve is a plot of sensitivity (recall) versus  $1 - \text{specificity}$  [28].

$$TPR \text{ or recall} = \frac{TP}{TP + FN} \quad FPR = \frac{FP}{FP + TN}$$

We can then calculate the area under the curve (AUC) of the ROC curve to get the ROC AUC score. The AUC can be used to evaluate the performance of classifiers and represents a model's ability to distinguish between the two binary classes – in this case: goal scored (1), and no goal (0). An area of 1.0 is a perfect classifier, getting all of its predictions in the right class, whereas an area of 0.5 is a classifier that is as good as random. Therefore, the closer the ROC curve to the upper left hand corner of its plot and therefore the higher the area under the ROC curve, the better the classifier performs.

In addition to the ROC AUC and the true positive rate (recall), we will also use some additional metrics, based on the calculation of a confusion matrix, to evaluate the performance of the machine learning classifiers we are using: accuracy; precision, which measures the number of correctly classified goals; and the F1 Score, which can be interpreted as a weighted average of recall and precision.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$precision = \frac{TP}{TP + FP}$$

$$F1 = \frac{2TP}{2TP + FP + FN}$$

## 4.2 Results without Tuning

We initially ran the three classifiers – logistic regression, CART, and k-nearest neighbours – on the three different datasets produced using different feature selection techniques: feature importance using an extra trees classifier (FI), mutual information (MI), and recursive feature elimination (RFE).

For logistic regression, highest ROC AUC score of 0.8444 was recorded on the FI dataset, marginally higher than the scores for the RFE (0.8431) and MI (0.8355) datasets. However, the RFE dataset recorded the highest accuracy with 0.8836, compared to 0.8826 for the FI dataset and 0.8811 for the MI dataset; the highest recall with a score of 0.2454, marginally higher than the FI dataset's score of 0.2436 and much higher than the MI dataset's recall of 0.2271; the highest F1 score with 0.3676 - again, the FI dataset was marginally lower with 0.3639 and the MI dataset was further away with an F1 score of 0.3449; and the highest precision score of 0.7322, with the FI and MI datasets having precision scores of 0.7189 and 0.7168 respectively.

	FI	MI	RFE
AUC Score	0.8444	0.8355	0.8431
Accuracy	0.8826	0.8811	0.8836
Recall	0.2436	0.2271	0.2454
F1	0.3639	0.3449	0.3676
Precision	0.7189	0.7168	0.7322

Adapted from table 4c in Annex B: Evaluation Metrics for Logistic Regression without Tuning

Looking at the confusion matrices produced by the logistic regression classifier using the different datasets, confirms that the RFE dataset was the best performing for our logistic regression classifier, as it recorded the highest number of true positives: 134, as opposed to 133 for the FI dataset and 124 for the MI dataset; the lowest amount of false negatives: 412, as opposed to 413 for the FI dataset and 422 for the MI dataset; the joint lowest amount of false positives along with the MI dataset with 49, as opposed to 52 for the FI dataset; and the joint highest amount of true negatives along with the MI dataset with 3366, as opposed to 3363 for the FI dataset.

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3363	52	3366	49	3366	49
	Goal	413	133	422	124	412	134

Table 4a from Annex B: Logistic Regression Confusion Matrices without Tuning

For our CART classifier, the RFE dataset produced the highest scores across every metric used, with marginally higher scores for ROC AUC: 0.6584, as opposed to 0.6548 for the MI dataset and 0.6532 for the FI dataset, and for accuracy: 0.8382, as opposed to 0.8372 for the FI dataset and 0.8359 for the MI dataset. For the remaining metrics, the RFE dataset produced scores with a larger gap to the FI and MI datasets. For precision, the RFE dataset scored 0.4103, as opposed to 0.4048 for the MI dataset and 0.3993 for the FI dataset; for recall, the RFE dataset scored 0.4114, as opposed to 0.4048 for the MI dataset and 0.4033 for the FI dataset; for F1, the RFE dataset scored 0.4125, as opposed to 0.4075 for the FI dataset and 0.4048 for the MI dataset.

	FI	MI	RFE
AUC Score	0.6532	0.6548	0.6584
Accuracy	0.8372	0.8359	0.8382
Recall	0.3993	0.4048	0.4103
F1	0.4033	0.4048	0.4114
Precision	0.4075	0.4048	0.4125

Adapted from table 5c in Annex B: Evaluation Metrics for CART without Tuning

Looking at the confusion matrices produced by the CART classifier using the different datasets confirms that the RFE dataset was the best performing for our CART classifier, recording the highest number of true positives: 224, as opposed to 221 for the MI dataset and 217 for the FI dataset, and the lowest amount of false negatives: 322, as opposed to 325 for the MI dataset and 328 for the FI dataset. However, it was the FI dataset which produced the highest number of true negatives: 3098, as opposed to 3096 for the RFE dataset and 3090 for the MI dataset, and the FI dataset also produced the lowest number of false negatives: 317, as opposed to 319 for the RFE dataset and 325 for the MI dataset.

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3098	317	3090	325	3096	319
	Goal	328	218	325	221	322	224

Table 5a from Annex B: CART Confusion Matrices without Tuning

Lastly, for our k-nearest neighbours classifier, the RFE and FI datasets produced identical scores for every metric other than ROC AUC score, which was marginally higher for the RFE dataset: 0.8023, than for the FI dataset: 0.8022. For the RFE and FI datasets, accuracy was: 0.8836, as opposed to 0.8755 for the MI dataset; the recall was 0.3864, as opposed to 0.3535 for the MI dataset, the F1 score was: 0.4779, as opposed to 0.4391 for the MI dataset; and the precision score was: 0.6261, as opposed to 0.5796 for the MI dataset.

	FI	MI	RFE
AUC Score	0.8022	0.7984	0.8023
Accuracy	0.8836	0.8755	0.8836
Recall	0.3864	0.3535	0.3864
F1	0.4779	0.4391	0.4779
Precision	0.6261	0.5796	0.6261

Adapted from table 6c in Annex B: Evaluation Metrics for KNN without Tuning

Similar to the evaluation metrics above, the confusion matrices produced for the RFE and FI datasets were identical and, again, better performing when compared to the MI dataset. The RFE and FI datasets scored highest on true positives: 211, as opposed to 193 for the MI dataset; highest on true negatives: 3289, as opposed to 3275 for the MI dataset; lowest for false positives: 126, as opposed to 140 for the MI dataset; and lowest for false negatives: 335, as opposed to 353 for the MI dataset.

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3289	126	3275	140	3289	126
	Goal	335	211	353	193	335	211

Table 6a from Annex B: KNN Confusion Matrices without Tuning

Overall, across the 3 different classifiers, the RFE dataset performed the best without tuning the models. This is unsurprising as recursive feature elimination was the most computationally expensive of the three feature selection techniques, building a model with all the features included, removing the lowest performing feature, and repeating this recursively until we reach our reduced number of variables that are deemed useful to the dataset.

Of the classifiers using the RFE dataset, our logistic regression classifier achieved the highest ROC AUC score, accuracy, and precision with scores of 0.843, 0.8836 (joint with KNN), and 0.7322 respectively, compared to 0.8023, 0.8836 (joint with logistic regression) and 0.6261 for KNN, and 0.6584, 0.838, and 0.4125 for CART. However, logistic regression had the lowest recall and F1 scores of all classifiers with scores of 0.2454 and 0.3676 respectively. The CART classifier scored the highest on recall with a score 0.4103, compared to 0.3864 for KNN, whereas KNN performed highest on F1 with a score of 0.4779, compared to 0.4114 for the CART classifier.

	LR/RFE	CART/RFE	KNN/RFE
AUC Score	0.8431	0.6584	0.8023
Accuracy	0.8836	0.8382	0.8836
Recall	0.2454	0.4103	0.3864
F1	0.3676	0.4114	0.4779
Precision	0.7322	0.4125	0.6261

Adapted from table 7 in Annex B: best combinations of dataset and classifier without tuning

### 4.3 Optimal Hyperparameters

After running the machine learning classifiers without tuning the hyperparameters, they were rerun using both grid search and randomised search, with the grid and distributions described in section 3.2. The accuracy measure used for the best performing hyperparameters (best scores in the tables) was changed from accuracy to ROC AUC.

For logistic regression classifier, we looked at three hyperparameters: penalty, C, and solver. The results are shown in the table below for each dataset and search method:

	Grid Search			Random Search		
	FI	MI	RFE	FI	MI	RFE
C	0.1	0.1	0.1	0.1706	0.0202	0.1706
Penalty	l2	l2	l2	l2	l2	l2
Solver	liblinear	liblinear	liblinear	liblinear	lbfgs	liblinear
Best Score	0.8225	0.8251	0.8248	0.8224	0.8249	0.8246

Table 1 from Annex B: Logistic Regression hyperparameter tuning on three different datasets, using grid search and randomised search methods

When using each different dataset, grid search produced a marginally higher ROC AUC best score than the randomised search for the logistic regression hyperparameters, and these hyperparameters will be used when the models are rerun after tuning. There was no difference in penalty, when using each dataset and search method: l2 was selected for all, and only when using the MI dataset in the randomised search was a different optimal solver (lbfgs) chosen than the other combinations of dataset and search method, which all found liblinear to be the optimal value the solver hyperparameter of the classifier.

The optimal value of C chosen, using each dataset, was the same when using grid search: 0.1. For the continuous distribution used in the randomised search to find the optimal value of C, 0.1706 was found to be the optimal value of C when using both the FI and RFE datasets - not too different to the optimal value of C found through grid search - however when using the

MI dataset, the optimal value of C was 0.0202. Overall, the highest best scores were produced when using the MI dataset in both tuning techniques.

For the CART classifier, we looked at four hyperparameters: criterion, the maximum number of features that are evaluated for splitting at each node, the minimum number of samples a node must have before it can be split, and the minimum number of samples a leaf node must have. The results are shown in the table below for each dataset and search method:

	Grid Search			Random Search		
	FI	MI	RFE	FI	MI	RFE
Criterion	entropy	entropy	gini	entropy	entropy	entropy
max_features	auto	auto	auto	log2	auto	auto
min_samples_leaf	9	8	9	9	8	8
min_samples_split	2	2	2	9	6	6
Best Score	0.7886	0.7952	0.7959	0.7886	0.7952	0.7863

Table 2 from Annex B: CART hyperparameter tuning on three different datasets, using grid search and randomised search methods

Interestingly, when using the FI and MI datasets, grid search and randomised search produced identical ROC AUC best scores for the CART classifier, with scores of 0.7886 and 0.7952 when using the FI and MI datasets respectively, and these were all achieved with entropy as the criterion. When using the RFE dataset, entropy was also chosen as the optimal value for criterion in the randomised search, but gini was chosen as the optimal value using grid search.

Despite producing identical scores in the grid search and randomised search, when using the FI dataset, different optimal values were found for max\_features: auto (square root of the number of features) was the optimal method found through grid search, and log2 (log2 of the number of features) was the optimal method found through randomised search. Similarly, different optimal values were found the min\_samples\_split hyperparameter: 2 in the grid

search and 9 in the randomised search. The `min_samples_leaf` hyperparameter, however, had the same optimal value across both tuning techniques: 9.

There was no difference when using the MI or RFE datasets to find the optimal value for the `max_features` hyperparameter across the two tuning techniques: both datasets used the grid search and randomised search. When using the MI dataset, the classifier also chose the same value for `min_samples_leaf`: 8, across the two tuning techniques, however had a different value for `min_samples_split`: 2 in the grid search and 6 in the randomised search. When using the RFE dataset, the optimal values for the classifier were different for both `min_samples_leaf`: 9 in the grid search and 8 in the randomised search, and `min_samples_split`: 2 in the grid search and 6 in the randomised search, across the two tuning techniques. Overall, using the RFE dataset produced the best score in the grid search and using the MI dataset produced the highest best score in the randomised grid search.

Lastly, for the k-nearest neighbours classifier, we tuned three hyperparameters: the number of neighbours, the weight function, and the (distance) metric. The results are shown in the table below for each dataset and search method:

	Grid Search			Random Search		
	FI	MI	RFE	FI	MI	RFE
Metric	manhattan	manhattan	euclidean	manhattan	manhattan	euclidean
N_Neighbours	9	9	9	9	9	9
Weights	uniform	uniform	uniform	distance	distance	distance
Best Score	0.8112	0.8098	0.8114	0.8110	0.8094	0.8114

Table 3 from Annex B – K-Nearest Neighbours hyperparameter tuning on three different datasets, using grid search and randomised search methods

Each dataset selected the highest possible number of neighbours across both grid search and randomised search: 9. When using the FI and MI datasets, the optimal value for the (distance) metric hyperparameter was Manhattan distance, across both tuning techniques. When using the RFE dataset however, the optimal value for this hyperparameter was Euclidean distance.



Interestingly, when using each dataset and performing a grid search to find the optimal weight hyperparameter for the k-nearest neighbours classifier, uniform is chosen as the optimal weight. However, when the datasets are used for the classifier in a randomised search, distance is chosen as the optimal weight hyperparameter. Again, the highest best scores are found when we use grid search, for all three datasets (though the best score is identical when using the RFE dataset across the two search techniques). For both grid search and randomised search, it is when using the RFE dataset that the highest best scores are achieved.

Overall, we can see that grid search performs at least as well and usually better than randomised search. This is not a surprising result, given that the number of iterations the randomised search searched through was limited to 10 – much lower than the number of combinations the grid search explored. Our logistic regression model produced the highest best scores when hyperparameter tuning was applied, across all datasets and search methods. Interestingly, the classifier (CART) in which the grid search had to go through the most combinations of hyperparameters (288), and therefore had the biggest difference between grid search and randomised search iterations, produced identical results across both search methods when using the FI and MI datasets.

#### 4.4 Results using Optimal Hyperparameters from Tuning

Using the optimal hyperparameters produced in section 4.3, the machine learning classifiers were rerun and re-evaluated using the same metrics as before. For our logistic regression classifier, the ROC AUC score increases with tuning, when using all three datasets. However, when tuning is applied to the classifier using the RFE dataset, every other score decreases: accuracy falls from 0.8836 to 0.8821, recall falls from 0.2454 to 0.2344, F1 score falls from 0.3676 to 0.3541, and precision falls from 0.7322 to 0.7232. These slight falls, mean that the classifier no longer performs best when using the RFE dataset: the best overall performance for our classifier after tuning occurs when using the FI dataset, which has the highest accuracy: 0.8824, compared to 0.8821 when using the RFE dataset and 0.8806 when using the MI dataset; highest recall: 0.2363, compared to 0.2344 when using the RFE dataset and 0.2253 when using the MI dataset; highest F1 score: 0.3546, compared to 0.3541 when using the RFE dataset and 0.3436 when using the MI dataset; and the highest precision: 0.7247, compared to 0.7235 when using the MI dataset and 0.7232 when using the RFE dataset.

	Without Tuning			With Tuning		
	FI	MI	RFE	FI	MI	RFE
AUC Score	0.8444	0.8355	0.8431	0.8446	0.8813	0.8433
Accuracy	0.8826	0.8811	0.8836	0.8824	0.8806	0.8821
Recall	0.2436	0.2271	0.2454	0.2363	0.2253	0.2344
F1	0.3639	0.3449	0.3676	0.3564	0.3436	0.3541
Precision	0.7189	0.7168	0.7322	0.7247	0.7235	0.7232

Table 4c in Annex B: Evaluation Metrics for Logistic Regression

Looking at the confusion matrices produced for logistic regression using the three different datasets, using the FI dataset produces the highest number of true negatives: 129, compared to 128 when using the RFE dataset and 123 when using the MI dataset, and the lowest number of false negatives: 417, compared to 418 when using the RFE dataset and 423 when using the MI dataset. Using the MI dataset however, produces the highest number of true negatives: 3368, as opposed to 3366 when using both the FI and RFE datasets, and the lowest number of false positives: 47, as opposed to 49 when using both the FI and RFE datasets.

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3366	49	3368	47	3366	49
	Goal	417	129	423	123	418	128

Table 4b from Annex B: Logistic Regression Confusion Matrices with Tuning

For our CART classifier, ROC AUC score and precision see large increases across the three datasets once tuning is applied: when using the FI dataset, ROC AUC increases to 0.8034 from 0.6532 and precision increases to 0.5911 from 0.4075; when using the MI dataset, ROC AUC increases to 0.8095 from 0.6548, and precision increases to 0.6304 from 0.4048; when using the RFE dataset, ROC AUC score increases to 0.7883 from 0.6584, and precision increases to 0.6174 from 0.4125. Accuracy also increase after tuning when using all three datasets: to 0.8755 from 0.8372 when using the FI dataset; to 0.8791 from 0.8359 when using the MI dataset; and to 0.8798 from 0.8382 when using the RFE dataset.

Unfortunately, the recall score falls dramatically when using all three datasets: to 0.3150 from 0.3993 when using the FI dataset; to 0.2967 from 0.4048 when using the MI dataset; and to 0.3370 from 0.4103 when using the RFE dataset. This fall in recall negates the improvements in precision, resulting in marginal differences in the F1 score after tuning: F1 increases slightly to 0.4110 from 0.4033 when using the FI dataset; falls very slightly to 0.4035 from 0.4048 when using the MI dataset; and also increases to 0.4360 from 0.4414 for the RFE dataset. The CART classifier still performs better across the majority of categories after tuning when using the RFE dataset, as was the case without tuning, although the MI dataset performs better in two categories: ROC AUC and precision (and performs almost as well in accuracy).

	Without Tuning			With Tuning		
	FI	MI	RFE	FI	MI	RFE
AUC Score	0.6532	0.6548	0.6584	0.8034	0.8095	0.7883
Accuracy	0.8372	0.8359	0.8382	0.8755	0.8791	0.8798
Recall	0.3993	0.4048	0.4103	0.3150	0.2967	0.3370
F1	0.4033	0.4048	0.4114	0.4110	0.4035	0.4360
Precision	0.4075	0.4048	0.4125	0.5911	0.6304	0.6174

Table 5c in Annex B: Evaluation Metrics for CART

Looking at the confusion matrices produced for the CART classifier using the three different datasets, using the RFE dataset produces the highest number of true negatives: 184, compared to 172 when using the FI dataset, and 162 when using the MI dataset; and also produces the lowest number of false negatives: 362, compared to 372 when using the FI dataset, and 384 when using the MI dataset. However, using the MI dataset produces the largest number of true negatives: 3320, as opposed to 3301 when using the RFE dataset, and 3296 when using the FI dataset; and also produces the lowest number of false positives: 95, compared to 114 when using the RFE dataset, and 119 when using the FI dataset. Overall, though, using the RFE dataset edges using the MI dataset with the CART classifier after tuning, but it should be noted that the performance of the classifier when using the MI dataset showed a dramatic increase in performance after tuning.

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3296	119	3320	95	3301	114
	Goal	374	172	384	162	362	184

Table 5b from Annex B: CART Confusion Matrices with Tuning

Lastly, we will look at the performance of our k-nearest neighbours classifier across the three different datasets after tuning has been applied. ROC AUC score and precision showed good increases after tuning, across all three datasets: ROC AUC score increased to 0.8309 from 0.8022 when using the FI dataset, to 0.8293 from 0.7984 when using the MI dataset, and from 0.831; precision increased to 0.6818 from 0.6261 when using the FI dataset, to 0.6903 from 0.5796 when using the MI dataset, and to 0.6823 from 0.6261 when using the RFE dataset. Accuracy showed negligible improvements: increasing to 0.8884 from 0.8836 when using the FI dataset, to 0.8879 from 0.8755 when using the MI dataset, and to 0.8877 from 0.8836 when using the RFE dataset.

Like the CART classifier, however, the recall score fell for our KNN classifier across the three datasets: to 0.3571 from 0.3864 when using the FI dataset, to 0.3388 from 0.3535 when using the MI dataset, and to 0.3462 from 0.3864 when using the RFE dataset. This led to the F1 score falling when using both the FI and RFE datasets: to 0.4688 from 0.4779 and to 0.4593 from 0.4779 respectively, however the F1 score when using the MI dataset: to 0.4545 from 0.4391 (the increase in precision when using the MI dataset offset the fall in recall). It is much harder to see which dataset produces the best results overall when used with the KNN classifier after tuning: using the RFE produces a marginally higher ROC AUC score than the other two datasets, but using the FI dataset produces a marginally higher accuracy score, recall score, and F1 score than the other two datasets, and using the MI dataset produces the best precision.

	Without Tuning			With Tuning		
	FI	MI	RFE	FI	MI	RFE
ROC AUC	0.8022	0.7984	0.8023	0.8309	0.8293	0.8319
Accuracy	0.8836	0.8755	0.8836	0.8884	0.8879	0.8877
Recall	0.3864	0.3535	0.3864	0.3571	0.3388	0.3462
F1	0.4779	0.4391	0.4779	0.4688	0.4545	0.4593
Precision	0.6261	0.5796	0.6261	0.6818	0.6903	0.6823

Table 6c in Annex B: Evaluation Metrics for KNN

Looking at the confusion matrices produced for the KNN classifier using the three different datasets backs up this point: using the FI dataset produces the highest number of true negatives: 195, as opposed to 189 when using the RFE dataset and 185 when using the MI dataset; and also produces the lowest number of false negative: 351, as opposed to 357 when using the RFE dataset and 361 when using the MI dataset. However, the highest number of true negatives are produced when using the MI dataset: 3332, as opposed to 3327 when using the RFE dataset and 3324 when using the FI dataset; and the lowest number of false positives are produced when using the MI dataset: 83, as opposed to 88 when using the RFE dataset and 91 when using the FI dataset. Overall, though, the KNN classifier performs best after tuning when using the FI dataset.

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3324	91	3332	83	3327	88
	Goal	351	195	361	185	357	189

Table 6b from Annex B: KNN Confusion Matrices with Tuning

From the results discussed above, it is clear that tuning the hyperparameters has an impact on the different classifiers when using the different datasets. Pre tuning, using the RFE dataset produced the best results for all three classifiers but, whilst this is still true of the CART classifier after tuning, the logistic regression and KNN classifier perform best when using the FI dataset.

	LR/FI	CART/RFE	KNN/FI
AUC Score	0.8446	0.7883	0.8309
Accuracy	0.8824	0.8798	0.8884
Recall	0.2363	0.3370	0.3571
F1	0.3564	0.4360	0.4688
Precision	0.7247	0.6174	0.6818

Adapted from table 7 in Annex B: best combinations of dataset and classifier without tuning

## 5. Summary and Conclusion

The best combinations of dataset and classifier produced fairly similar results to one another both pre tuning and post tuning, with trade-offs in performance between different evaluation metrics. As such, there isn't one clear classifier and dataset combination that produces the best results.

The classifiers we trained achieved a high accuracy and ROC AUC score. However, given the imbalanced nature of the dataset – 1645 out of the total of 12,003 shots were goals – it is perhaps more instructive to look at the recall, precision, and F1 scores. Generally, most of the combinations of datasets and classifiers performed well on precision, however the recall score was pretty low (with and without tuning) which, in turn, contributed to a relatively low F1 score. This could potentially be down to the fact that 0.5 is incorrect as a probability threshold for determining whether a shot will result in a goal in for our logistic regression and knn classifiers, or as a threshold for splitting nodes for our decision trees in the CART classifier. This would be something to further investigate to improve the classifiers in this project.

We can also further develop our xG model to produce another metric called: expected goals on target (xGOT). This measures the expected goals value of a shot after a shot has been taken and is going on target. The Statsbomb data provides use with coordinates on the goal (as seen in figure 4 below), which we can use to calculate where a shot is going to end up (provided it is on target) to produce an xGOT number that can help us explore further the likelihood of a shot becoming a goal.

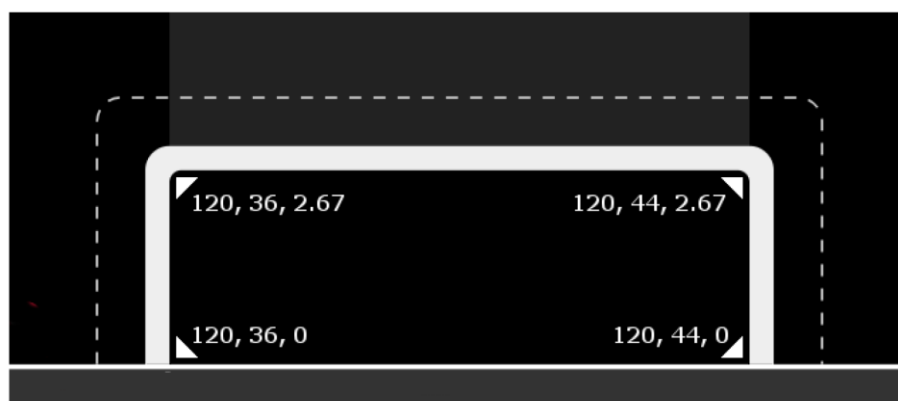


Figure 4: coordinates of the goal from Statsbomb data [10]

Like xG, the number produced for xGOT is a value between 0 (certain no goal) and 1 (certain goal), but we have further context: a shot going that is going towards a top corner of the goal is far more likely to result in a goal than one that is aimed to the middle.

The main thing xGOT tells us is how well a player is executing their shots – if a player is managing an xGOT value that is consistently higher than their xG value for a shot, this can indicate that the player is executing their shots better than the quality of the chance they are being provided with.

### Which players added the most value to their on-target shots last season?

Non-penalty shooting goals added in the Premier League in 2020-21.











		XG	XGOT	SHOOTING GOALS ADDED
	Son Heung-min	8.9	12.7	3.8
	J. Ward-Prowse	2.6	5.6	3.0
	A. Lacazette	9.3	11.8	2.5
	D. Ings	7.0	9.4	2.4
	J. Rodriguez	2.8	5.2	2.4
	J. Lingard	3.4	5.7	2.3
	S. Dallas	4.2	6.3	2.1
	Raphinha	5.7	7.7	2.0
	J. Willock	3.7	5.6	1.9
	I. Gundogan	7.4	9.2	1.8

Figure 5: table showing the biggest overperformance between xGOT – xG in the 2020-21 Premier League season [5]



Analogously, for goalkeepers, xGOT tells us how many goals they should have conceded based on the quality of the shots on target they have faced and, when taken with the xG value of those shots, we can calculate the difference to see whether a goalkeeper is preventing more or less goals than our model would expect.

### Who were the best shot-stoppers in the Premier League last season?

All non-penalty shots faced in the Premier League in 2020-21, excluding own goals. Includes all players with 900+ mins played.

			SHOTS FACED	XGOT CONCEDED	GOALS CONCEDED	GOALS PREVENTED
1	H. Lloris		487	47.1	42	5.1
2	E. Martinez		539	49.7	45	4.7
3	N. Pope		477	40.4	36	4.4
4	A. Areola		407	51.3	47	4.3
5	Alisson		274	35.9	32	3.9
6	Ederson		263	29.3	27	2.3
7	L. Fabianski		435	41.7	40	1.7
8	J. Pickford		397	38.2	38	0.2
9	B. Leno		393	32.5	33	-0.5
10	D. Henderson		151	11.5	12	-0.5
11	M. Dubravka		211	16.6	18	-1.4
12	D. de Gea		278	28.1	30	-1.9
13	E. Mendy		256	22.0	24	-2.0
14	I. Meslier		521	48.0	50	-2.0
15	R. Patricio		426	45.7	48	-2.3
16	R. Sanchez		265	23.5	26	-2.5
17	S. Johnstone		579	67.9	71	-3.1
18	K. Darlow		366	37.4	41	-3.6
19	M. Ryan		114	16.3	20	-3.7
20	A. Ramsdale		544	56.3	60	-3.7
21	K. Schmeichel		370	42.9	48	-5.1
22	A. McCarthy		343	47.0	56	-9.0
23	V. Guaita		521	51.5	64	-12.5

Figure 6: table showing the differences between xGOT – xG for goalkeepers in the 2020-21 Premier League season [5]

## Appendix A – Results of applying different feature selection techniques to the dataset

Table 1: results of applying Mutual Information feature selection

Feature	MI Score
shot_angle	0.05352091
shot_distance	0.04917578
duration	0.02499913
shot.type.name_Penalty	0.01151956
shot.open_goal	0.01151573
shot.one_on_one	0.00701462
shot.first_time	0.00644591
shot.technique.name_Overhead Kick	0.00506777
minute	0.00348888
shot.technique.name_Diving Header	0.00340726
shot.deflected	0.00293958
shot.type.name_Free Kick	0.00264556
shot.body_part.name_Head	0.00201606
under_pressure	0.00167723
shot.technique.name_Backheel	0.00135272
shot.aerial_won	0.00108384
shot.body_part.name_Other	0.00099228
shot.technique.name_Lob	0.00010032
shot.follows_dribble	0
shot.technique.name_Half Volley	0
shot.technique.name_Normal	0
shot.technique.name_Volley	0
shot.body_part.name_Foot	0
shot.type.name_Corner	0
shot.type.name_Open Play	0

Table 2: results of applying Feature Importance using Extra Trees Classifier

Feature	FI Score
shot_angle	0.227013
duration	0.222049
shot distance	0.213603
minute	0.164886
shot.open_goal	0.045632
shot.type.name_Penalty	0.037030
shot.deflected	0.013867
shot.one_on_one	0.013475
shot.first_time	0.011191
under_pressure	0.008043
shot.technique.name_Normal	0.006439
shot.technique.name_Lob	0.005986
shot.body_part.name_Foot	0.005503
shot.body_part.name_Head	0.004847
shot.type.name_Free Kick	0.004349
shot.type.name_Open Play	0.004146
shot.technique.name_Half Volley	0.003464
shot.technique.name_Volley	0.002486
shot.aerial_won	0.001640
shot.technique.name_Diving Header	0.001121
shot.technique.name_Overhead Kick	0.001082
shot.technique.name_Backheel	0.000788
shot.follows_dribble	0.000718
shot.body_part.name_Other	0.000625
shot.type.name_Corner	0.000017

Table 3: results of applying Recursive Feature Selection

Feature	RFE Ranking	Important
minute	1	True
duration	1	True
shot.first_time	1	True
under_pressure	1	True
shot.one_on_one	1	True
shot.aerial_won	1	True
shot.open_goal	1	True
shot.deflected	1	True
shot_distance	1	True
shot_angle	1	True
shot.technique.name_Half Volley	1	True
shot.technique.name_Lob	1	True
shot.technique.name_Normal	1	True
shot.technique.name_Volley	1	True
shot.body_part.name_Foot	1	True
shot.body_part.name_Head	1	True
shot.type.name_Open Play	1	True
shot.type.name_Penalty	1	True
shot.type.name_Free Kick	2	False
shot.technique.name_Overhead Kick	3	False
shot.technique.name_Diving Header	4	False
shot.follows_dribble	5	False
shot.body_part.name_Other	6	False
shot.technique.name_Backheel	7	False
shot.type.name_Corner	8	False

## Appendix B – Hyperparameter tuning results and evaluation metrics pre and post tuning

Table 1 - Logistic Regression hyperparameter tuning on 3 different datasets, each using a different feature selection technique: Feature Importance, Mutual Information, Recursive Feature Elimination

	Grid Search			Random Search		
	FI	MI	RFE	FI	MI	RFE
C	0.1	0.1	0.1	0.1706	0.0202	0.1706
Penalty	l2	l2	l2	l2	lbfgs	l2
Solver	liblinear	liblinear	liblinear	liblinear	liblinear	liblinear
Best Score	0.8225	0.8251	0.8248	0.8224	0.8249	0.8246

Table 2 - CART hyperparameter tuning on 3 different datasets, each using a different feature selection technique: Feature Importance, Mutual Information, Recursive Feature Elimination

	Grid Search			Random Search		
	FI	MI	RFE	FI	MI	RFE
Criterion	entropy	entropy	gini	entropy	entropy	entropy
max_features	auto	auto	auto	log2	auto	auto
min_samples_leaf	9	8	9	9	8	8
min_samples_split	2	2	2	9	6	6
Best Score	0.7886	0.7952	0.7959	0.7886	0.7952	0.7863

Table 3 – K-Nearest Neighbours hyperparameter tuning on 3 different datasets, each using a different feature selection technique: Feature Importance, Mutual Information, Recursive Feature Elimination

	Grid Search			Random Search		
	FI	MI	RFE	FI	MI	RFE
Metric	manhattan	manhattan	euclidean	manhattan	manhattan	euclidean
N_Neighbours	9	9	9	9	9	9
Weights	uniform	uniform	uniform	distance	distance	distance
Best Score	0.8112	0.8098	0.8114	0.8110	0.8094	0.8114

The following tables display evaluation metrics and confusion matrices on the 3 machine learning classifiers used – Logistic Regression, CART, KNN - on 3 different datasets created through different selection techniques: Feature Importance, Mutual Information, and Recursive Feature Elimination.

Table 4 – Logistic Regression

4a: Confusion Matrices – Without Tuning

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3363	52	3366	49	3366	49
	Goal	413	133	422	124	412	134

4b: Confusion Matrices – With Tuning

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3366	49	3368	47	3366	49
	Goal	417	129	423	123	418	128

4c: Evaluation Metrics

	Without Tuning			With Tuning		
	FI	MI	RFE	FI	MI	RFE
ROC AUC	0.8444	0.8355	0.8431	0.8446	0.8813	0.8433
Accuracy	0.8826	0.8811	0.8836	0.8824	0.8806	0.8821
Recall	0.2436	0.2271	0.2454	0.2363	0.2253	0.2344
F1	0.3639	0.3449	0.3676	0.3564	0.3436	0.3541
Precision	0.7189	0.7168	0.7322	0.7247	0.7235	0.7232

Table 5 – CART

5a: Confusion Matrices – Without Tuning

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3098	317	3090	325	3096	319
	Goal	328	218	325	221	322	224

5b: Confusion Matrices – With Tuning

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3296	119	3320	95	3301	114
	Goal	374	172	384	162	362	184

5c: Evaluation Metrics without and with Tuning

	Without Tuning			With Tuning		
	FI	MI	RFE	FI	MI	RFE
ROC AUC	0.6532	0.6548	0.6584	0.8034	0.8095	0.7883
Accuracy	0.8372	0.8359	0.8382	0.8755	0.8791	0.8798
Recall	0.3993	0.4048	0.4103	0.3150	0.2967	0.3370
F1	0.4033	0.4048	0.4114	0.4110	0.4035	0.4360
Precision	0.4075	0.4048	0.4125	0.5911	0.6304	0.6174

Table 6 – KNN

6a: Confusion Matrices – Without Tuning

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3289	126	3275	140	3289	126
	Goal	335	211	353	193	335	211

6b: Confusion Matrices – With Tuning

		Predicted Class					
		FI		MI		RFE	
		No Goal	Goal	No Goal	Goal	No Goal	Goal
Actual Class	No Goal	3324	91	3332	83	3327	88
	Goal	351	195	361	185	357	189

6c: Evaluation Metrics

	Without Tuning			With Tuning		
	FI	MI	RFE	FI	MI	RFE
ROC AUC	0.8022	0.7984	0.8023	0.8309	0.8293	0.8319
Accuracy	0.8836	0.8755	0.8836	0.8884	0.8879	0.8877
Recall	0.3864	0.3535	0.3864	0.3571	0.3388	0.3462
F1	0.4779	0.4391	0.4779	0.4688	0.4545	0.4593
Precision	0.6261	0.5796	0.6261	0.6818	0.6903	0.6823



Table 7 – Overall Results

Table 7: Best combinations of dataset and classifier

	Without Tuning			With Tuning		
	LR/RFE	CART/RFE	KNN/RFE	LR/FI	CART/RFE	KNN/FI
AUC Score	0.8431	0.6584	0.8023	0.8446	0.7883	0.8309
Accuracy	0.8836	0.8382	0.8836	0.8824	0.8798	0.8884
Recall	0.2454	0.4103	0.3864	0.2363	0.3370	0.3571
F1	0.3676	0.4114	0.4779	0.3564	0.4360	0.4688
Precision	0.7322	0.4125	0.6261	0.7247	0.6174	0.6818

## Appendix C – Full Project Code

```
"""MSc Project"""

import pandas as pd
import numpy as np
import math

df = pd.read_json(path_or_buf='shots.json')

df = df.fillna(0)

df.shape
df.head()

#create additional variable: shot distance

shot_distance = []

for i in range(0, df.shape[0]):
    x = df['location'][i][0]
    y = df['location'][i][1]
    distance = math.sqrt((120-x)**2 + (40-y)**2)
    shot_distance.append(distance)

df['shot_distance'] = shot_distance

#create additional variable: shot angle

shot_angle = []

for i in range(0, df.shape[0]):
    x = df['location'][i][0]
    y = df['location'][i][1]
    angle = math.degrees(math.atan((8*(120-x))/((120-x)**2 + ((y**2) - (80*y)
+ 1584))))
    shot_angle.append(angle)

df['shot_angle'] = shot_angle

#drop variables that we know aren't useful

df_trim = df.drop(['id',
'index', 'period', 'second', 'related_events', 'type.id', 'type.name',
'possession_team.id', 'possession_team.name', 'team.id',
'team.name',
'tactics.lineup', 'player.id', 'player.name',
'position.id', 'position.name',
'pass.end_location', 'carry.end_location',
'shot.statsbomb_xg', 'shot.end_location',
'shot.key_pass_id', 'shot.freeze_frame',
'goalkeeper.end_location',
'match_id', 'competition_id', 'season_id',
'out', 'off_camera', 'shot.saved_off_target',
'shot.saved_to_post', 'shot.redirect'], axis=1)

df_trim.head()
```

```

#remove related variables columns e.g. id columns corresponding to name
columns

df_clean =
df_trim.drop(['timestamp', 'possession', 'location', 'play_pattern.id',
'play_pattern.name',

'shot.technique.id', 'shot.body_part.id', 'shot.type.id', 'shot.outcome.id'],
axis =1)

#custom encoding for variables

df_clean['shot.body_part.name'] = ['Foot' if x in ['Right Foot', 'Left
Foot'] else x for x in df_clean['shot.body_part.name']]
df_clean['shot.outcome.name'] = [1 if x == 'Goal' else 0 for x in
df_clean['shot.outcome.name']]
df_clean = df_clean.rename(columns={'shot.outcome.name': 'goal'})

#one-hot encode variables

df_clean = pd.get_dummies(df_clean, columns=['shot.technique.name'])
df_clean = pd.get_dummies(df_clean, columns=['shot.body_part.name'])
df_clean = pd.get_dummies(df_clean, columns=['shot.type.name'])

df_clean.head()

#create X and Y dataframes

from sklearn.model_selection import train_test_split
from sklearn import preprocessing

X = df_clean.drop(['goal'], axis=1)
Y = df_clean['goal']

test_size = 0.33
seed = 7
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=test_size, random_state=seed)

scaler = preprocessing.StandardScaler().fit(X_train)
x_train_scaled = scaler.transform(X_train)
x_test_scaled = scaler.transform(X_test)

columns = list(X)

X.head()

```

```

#feature selection using mutual information

from sklearn.feature_selection import mutual_info_classif

np.random.seed(7)

mi = mutual_info_classif(x_train_scaled,Y_train)
print("MI scores: ", mi)

mi_features = []
mi_index = []

for i in range(0,len(mi)):
    if mi[i] != 0:
        mi_features.append((columns[i]))
        mi_index.append(i)

x_train_mi = x_train_scaled[:, mi_index]
x_test_mi = x_test_scaled[:, mi_index]

print("Number of features after MI feature selection applied: ",
      len(mi_features))
print("Features selected after MI feature selection applied: ",
      mi_features)

df_clean_mi = df_clean[df_clean.columns.intersection(mi_features)]

df_clean_mi.head()


#feature selection using Recursive Feature Elimination (RFE)

from sklearn.feature_selection import RFECV
from sklearn.ensemble import RandomForestClassifier

np.random.seed(7)

rfecv = RFECV(estimator=RandomForestClassifier(),cv=5)
rfecv_fit = rfecv.fit(x_train_scaled,Y_train)

print("Number of features after RFE applied: ", rfecv_fit.n_features_)
print("RFE feature ranking: ", rfecv_fit.ranking_)
rfecv_fit.support_

rfe_features = []
rfe_index = []

for i in range(0,len(rfecv_fit.support_)):
    if rfecv_fit.support_[i] == True:
        rfe_features.append((columns[i]))
        rfe_index.append(i)

x_train_rfe = x_train_scaled[:, rfe_index]
x_test_rfe = x_test_scaled[:, rfe_index]

print("Features selected after RFE applied: ", rfe_features)

df_clean_rfe = df_clean[df_clean.columns.intersection(rfe_features)]

df_clean_rfe.head()

```

```

#feature selection using Feature Importance (Extra Trees Classifier)

from sklearn.ensemble import ExtraTreesClassifier

np.random.seed(7)

fi = ExtraTreesClassifier()
fi_fit = fi.fit(x_train_scaled,Y_train)

print("Feature importances: ", fi_fit.feature_importances_)

fi_features = []
fi_index = []

for i in range(0,fi_fit.n_features_):
    if fi_fit.feature_importances_[i] >= 0.003:
        fi_features.append((columns[i]))
        fi_index.append(i)

x_train-fi = x_train_scaled[:, fi_index]
x_test-fi = x_test_scaled[:, fi_index]

print("Number of features after FI feature selection applied: ",
len(fi_features))
print("Features selected after FI feature selection applied: ",
fi_features)

df_clean-fi = df_clean[df_clean.columns.intersection(fi_features)]

df_clean-fi.head()

#metrics to evaluate the different models

from sklearn.metrics import roc_auc_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score

def evaluation_metrics(y_true,y_pred):
    print('Confusion Matrix: ', '\n')
    print(confusion_matrix(y_true,y_pred), '\n')
    print('Accuracy: ', accuracy_score(y_true,y_pred))
    print('Recall: ', recall_score(y_true,y_pred))
    print('F1: ', f1_score(y_true,y_pred))
    print('Precision: ', precision_score(y_true,y_pred), '\n')

```

```

#models without tuning

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
np.random.seed(7)

def run_model(model, x_train, x_test, y_train, y_test):
    model.fit(x_train, y_train)
    y_pred = model.predict(x_test)
    print('ROC AUC: ', roc_auc_score(y_test,
y_score=model.predict_proba(x_test)[: , 1]))
    evaluation_metrics(y_test, y_pred)

models = []
models.append(('Logistic Regression', LogisticRegression()))
models.append(('CART', DecisionTreeClassifier(random_state=7)))
models.append(('K-Nearest Neighbours', KNeighborsClassifier()))

x_data = []
x_data.append(('Feature Importance Dataset', x_train_fi, x_test_fi))
x_data.append(('Mutual Information Dataset', x_train_mi, x_test_mi))
x_data.append(('RFE Dataset', x_train_rfe, x_test_rfe))

for name, model in models:
    print(name, '\n')
    for x_name, x_train, x_test in x_data:
        print(x_name)
        run_model(model, x_train, x_test, Y_train, Y_test)

#create functions to tune hyperparameters

from scipy.stats import loguniform, randint
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV

#grid search function

def grid_search(model,param_grid,x_train_data):
    gs = GridSearchCV(estimator=model,param_grid=param_grid,n_jobs=-
1,scoring='roc_auc')
    gs_fit = gs.fit(x_train_data,Y_train)
    print(gs.best_params_)
    print(gs.best_score_)
    return gs_fit

param_grid_lr = {'C': [0.01, 0.1, 1, 10, 100],
                  'penalty': ['l1', 'l2'],
                  'solver': ['liblinear', 'saga', 'lbfgs']}

param_grid_cart = {'criterion':['gini','entropy'],
                   'max_features': ['auto', 'log2'],
                   'min_samples_split':np.arange(2,10),
                   'min_samples_leaf':np.arange(1,10)}

param_grid_knn = {'n_neighbors': np.arange(1,10),
                  'metric': ['euclidean', 'manhattan', 'minkowski'],
                  'weights': ['uniform', 'distance']}

```

```

#random search function

def random_search(model,param_dist,x_train_data):
    rs = RandomizedSearchCV(estimator=model, param_distributions=param_dist,
n_iter=10, random_state=7,n_jobs=-1,scoring='roc_auc')
    rs_fit = rs.fit(x_train_data,Y_train)
    print(rs.best_params_)
    print(rs.best_score_)
    return rs_fit

param_dist_lr = {'C': loguniform(0.01,100),
                'penalty': ['l1', 'l2'],
                'solver': ['liblinear', 'saga', 'lbfgs']}

param_dist_cart = {'criterion':['gini','entropy'],
                  'max_features': ['auto', 'log2'],
                  'min_samples_split': randint(2,10),
                  'min_samples_leaf': randint(1,10)}

param_dist_knn = {'n_neighbors': randint(1,10),
                  'metric': ['euclidean', 'manhattan', 'minkowski'],
                  'weights': ['uniform', 'distance']}

param_all = []
param_all.append(('Logistic Regression', LogisticRegression(),
param_grid_lr, param_dist_lr))
param_all.append(('CART', DecisionTreeClassifier(random_state=7),
param_grid_cart, param_dist_cart))
param_all.append(('KNN', KNeighborsClassifier(), param_grid_knn,
param_dist_knn))

#tune hyperparemeters

np.random.seed(7)

params_best = []

for name, model, param_grid, param_dist in param_all:
    print(name,'\n')
    for x_name, x_train, x_test in x_data:
        print(x_name)
        gs = grid_search(model, param_grid, x_train)
        rs = random_search(model, param_dist, x_train)
        if gs.best_score_ >= rs.best_score_:
            params_best.append((name, x_name, x_train, x_test,
gs.best_estimator_))
        else:
            params_best.append((name, x_name, x_train, x_test,
rs.best_estimator_))

```

```
#rerun models with tuned hyperparameters

from sklearn.metrics import roc_curve, plot_roc_curve
from matplotlib import pyplot as plt

np.random.seed(7)

for name, x_name, x_train, x_test, best_estimator in params_best:
    print(name, '\n')
    print(x_name, '\n')
    run_model(best_estimator, x_train, x_test, Y_train, Y_test)
    plot_roc_curve(best_estimator, x_test, Y_test)
    plt.show()
```



## Bibliography

1. Tom Worville, 'Goals galore, more away wins and fewer draws: 2020-21 under the microscope', The Athletic, October 21, 2020 (accessed August 2, 2021)
2. Tom Worville, 'Why Gareth Southgate picked Ollie Watkins for England and not Patrick Bamford', The Athletic, March 18, 2021 (accessed August 2, 2021)
3. Stats Perform, Explaining and Training Shot Quality, optasports.com, <https://statsbomb.com/2016/04/explaining-and-training-shot-quality/> (accessed August 2, 2021)
4. Stats Perform, 'Advanced Metrics', optasports.com, <https://www.optasports.com/services/analytics/advanced-metrics/> (accessed March 27, 2021)
5. Mark Carey and Tom Worville, 'The Athletic's football analytics glossary: explaining xG, PPDA, field tilt and how to use them', The Athletic, Jul 28, 2021 (accessed August 2, 2021)
6. Carl Bialik, 'The People Tracking Every Touch, Pass And Tackle in the World Cup', FiveThirtyEight, June 12, 2014 (accessed April 11, 2021)
7. Pappalardo et al., (2019) A public data set of spatio-temporal match events in soccer competitions, Nature Scientific Data 6:236
8. Anderson, C., Arrondel, L., Blais, A., Daoust, J., Laslier, J., & Van der Straeten, K. (2020). 'Messi, Ronaldo, and the Politics of Celebrity Elections: Voting for the Best Soccer Player in the World'. *Perspectives on Politics*, 18(1), 91-110. doi:10.1017/S1537592719002391
9. Pappalardo et al., (2019) A public data set of spatio-temporal match events in soccer competitions, Nature Scientific Data 6:236
10. StatsBomb Open Events Structure and Data Specification v4.0.0
11. Guyon, Isabelle & Elisseeff, André. (2003). An Introduction of Variable and Feature Selection. J. Machine Learning Research Special Issue on Variable and Feature Selection. 3. 1157 - 1182. 10.1162/153244303322753616.
12. Yishi Zhang, Shujuan Li, Teng Wang, Zigang Zhang, Divergence-based feature selection for separate classes, Neurocomputing, Volume 101, 2013, Pages 32-42, ISSN 0925-2312, <https://doi.org/10.1016/j.neucom.2012.06.036>.

13. S. J. Lee et al., "IMPACT: Impersonation Attack Detection via Edge Computing Using Deep Autoencoder and Feature Abstraction," in IEEE Access, vol. 8, pp. 65520-65529, 2020, doi: 10.1109/ACCESS.2020.2985089.
14. J. Brownlee, 'Information Gain and Mutual Information for Machine Learning,' <https://machinelearningmastery.com/information-gain-and-mutual-information/> (accessed September 12, 2021)
15. David J. C. MacKay. 2002. Information Theory, Inference & Learning Algorithms. Cambridge University Press, USA, pp 130.
16. Geurts, Pierre & Ernst, Damien & Wehenkel, Louis. (2006). Extremely Randomized Trees. Machine Learning. 63. 3-42. 10.1007/s10994-006-6226-1.
17. Scikit Learn, 'sklearn.feature\_selection.RFE', scikitlearn.org, [https://scikit-learn.org/stable/modules/generated/sklearn.feature\\_selection.RFE.html#sklearn.feature\\_selection.R](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html#sklearn.feature_selection.R) (accessed April 4, 2021)
18. Kuhn, M., & Johnson, K. (2019). Feature Engineering and Selection: A Practical Approach for Predictive Models (1st ed.). Chapman and Hall/CRC. <https://doi.org/10.1201/9781315108230>
19. A Geron, 'Training Models' in Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow, 2nd Edition 2019, ch. 4 pp. 135-147
20. A Geron, 'Decision Trees' in Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow, 2nd Edition 2019, ch. 6 pp. 177-182
21. Nisbet, R., Elder, J., Miner, G. (2009). Handbook of Statistical Analysis and Data Mining Applications. Germany: Elsevier Science, ch 12, pp. 278
22. Nisbet, R., Elder, J., Miner, G. (2009). Handbook of Statistical Analysis and Data Mining Applications. Germany: Elsevier Science, ch 11, pp. 239-240
23. Scikit Learn, '1.6.2. Nearest Neighbors Classification', scikitlearn.org, <https://scikit-learn.org/stable/modules/neighbors.html#classification> (accessed September 13, 2021)
24. J. Brownlee, 'How to Tune Algorithm Parameters with Scikit-Learn', machinelearningmastery.com, <https://machinelearningmastery.com/how-to-tune-algorithm-parameters-with-scikit-learn/> (accessed April 11, 2021)
25. Scikit Learn, '1.1.11. Logistic Regression', scikitlearn.org, [https://scikit-learn.org/stable/modules/linear\\_model.html#logistic-regression](https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression) (accessed September 13, 2021)

26. V. Zhou, 'A Simple Explanation of Gini Impurity', victorzhou.com,  
<https://victorzhou.com/blog/gini-impurity/> (accessed September 14, 2021)
27. V. Zhou, 'A Simple Explanation of Information Gain and Entropy', victorzhou.com,  
<https://victorzhou.com/blog/information-gain/> (accessed September 14, 2021)
28. A Geron, 'Classification' in Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow, 2nd Edition 2019, ch. 3 pp. 90-99