Stockholms universitet

Machine Learning for Football Betting

Alexander Kostavelis

Kandidatuppsats i matematisk statistik
Bachelor Thesis in Mathematical Statistics

Kandidatuppsats 2025:11
Matematisk statistik
Juni 2025

www.math.su.se

Matematisk statistik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm

Matematiska institutionen

Mathematical Statistics
Stockholm University
Bachelor Thesis **2025:11**
http://www.math.su.se

Machine Learning for Football Betting

Alexander Kostavelis*

June 2025

Abstract

This thesis investigates the performance of two supervised machine learning models, XGBoost and a feedforward neural network, to predict whether a football match will have more or fewer than 2.5 goals. Before the analysis, we present the mathematical background of the models. Using historical data from the top five European leagues downloaded from Football-Data.co.uk, relevant features are engineered and used to train the two models. Their performance is evaluated in terms of prediction accuracy as well as profitability when simulating bets on the over/under 2.5 goals market in the current 2024/2025 season. Although the results are not entirely satisfactory in terms of classification performance, the models show signs of profitability and should be investigated further. We propose several directions for further improvement, with the main suggestion being to incorporate more representative features that better capture the different aspects of match tactics and play styles, as a model is only as good as the data it learns from.

*Postal address: Mathematical Statistics, Stockholm University, SE-106 91, Sweden. E-mail: akostavelis3@gmail.com. Supervisor: Ola Hössjer and Johannes Heiny.

Acknowledgements

This is a Bachelor's thesis of 15 ECTS in mathematical statistics at Stockholm University. I would like to thank my supervisors Ola Hössjer and Johannes Heiny for their valuable feedback and continued support throughout the process. I would also like to thank Chun-Biu Li for his feedback and help during the analysis part of this thesis.

AI has been used for LaTeX formatting and correction of programming code. It has also occasionally been used for synonym suggestions.

Contents

1 Introduction

1.1 Background

Like many other sports, football is a source of entertainment, filled with emotion and unexpected moments. However, as football has grown into a massive global industry, the rise of data science and data generation has created opportunities for all parties involved in the football industry to optimize their decisions based on actual data, leaving as little as possible to chance. Teams now use data for everything from evaluating match performance and analyzing opponents' strengths and weaknesses to scouting players (a concept popularized in the movie Moneyball). This availability of data has also given betting companies (also called bookmakers) the ability to provide increasingly accurate odds (increasing their revenue) not only for match results but also for a variety of in-game outcomes such as total number of goals, corners and more.

This raises an interesting question: since data is publicly available, is it also possible for individuals outside the professional football industry like bettors and spectators to use it to their advantage? In this thesis, we explore that question by employing two machine learning models: XGBoost and a feedforward neural network. We focus on the over/under 2.5 goals market (predicting whether a match will have more or fewer than 2.5 goals) and frame the task as a binary classification problem. We evaluate the models based on predictive accuracy and cumulative profit by simulating bets on the 2024/2025 season using the models. We consider the top five European leagues.

1.2 Disposition

We start by presenting the underlying theory behind neural networks as well as XGBoost in Section 2. In Section 2 we also present some ideas of betting used when evaluating our models. In Section 3 we do an exploratory analysis of the data used as well as discuss the model design and hyperparameter tuning done. Section 4 contains the results of the modeling and in Section 5, discussion of the results and possible improvements can be found.

2 Background theory

This section will cover the underlying theory for the models and methods used in this thesis. Unless stated otherwise, Sections 2.1, 2.2, 2.3 and 2.4 are based on Hastie et al. (2017) Chapter 2.

2.1 Supervised learning

In machine learning, usually we have a set of input data (also called predictors or features) and we want to make predictions based on that data. There are two main categories in machine learning: supervised and unsupervised learning. The main difference between the two categories is the nature of the input data, where we have labeled data when applying supervised learning and unlabeled data for unsupervised learning. Labeled input data is data that has a corresponding output variable (also called target or response variable). In other words, in the unsupervised setting we are trying to learn useful properties in the dataset and describe the relations between the input variables, whereas in the supervised setting we are trying to predict the outcome of the output variable, given a realization of the input variables.

There are two categories of supervised learning: regression and classification problems. In the regression task, the output variable is quantitative whilst classification tasks handle qualitative output variables.

In this thesis we study a classification problem and employ two models that will be described in Sections 2.5 and 2.8.

2.2 Loss function

At the core of supervised learning, the objective is to find the relationship between the features and the response variable that gives us the best prediction. More formally, we are trying to approximate the true regression function $f(\mathbf{X})$ in $Y = f(\mathbf{X}) + \varepsilon$, where ε is a centered error term with $\text{Var}(\varepsilon|X = \boldsymbol{x}) = \sigma^2(\boldsymbol{x})$. Given a method of fitting our model to training data, we obtain an estimate $\hat{f}$ of the regression function f. This gives rise to a prediction $\hat{y} = \hat{f}(\boldsymbol{X})$ of the observed value y of the outcome variable Y for a new observation, with predictor vector $\boldsymbol{X}$ that is not part of training data. In order to find the best approximation $\hat{f}$ of f, we need a measure of the prediction error. This error is determined by a loss function (also called cost function) $L(y, \hat{y})$. The loss function is used when choosing a method for fitting the model to the data and determine the optimal fit $\hat{f}$, which is the estimate of f that *minimizes* the loss function. In regression problems with a training dataset $\{(\boldsymbol{X}_i, Y_i)\}_{i=1}^N$ of size N, a common loss function is the mean squared error (MSE) $\frac{1}{N}\sum_{i=1}^N (y_i - \hat{y}_i)^2$, which is referred to as the mean squared error of prediction (MSEP) if the dataset $\{(\boldsymbol{X}_i, Y_i)\}_{i=1}^N$

is a test dataset. However, MSE is not well-suited for classification tasks. Although MSE is possible to use for classification problems, it is more often used for regression problems. For this reason, other loss functions like rates of misclassification can also be used. The chosen loss function for the XGBoost model and the neural network used in this thesis will be presented in their respective model sections.

2.3 The bias-variance tradeoff

Before the approximated function $\hat{f}(\mathbf{X})$ is being modeled, the dataset is split into two parts: a training and a test set. If the dataset is large enough, it can also be split into three sets where the third set is used for validation. When we approximate the function $f(\boldsymbol{X})$ by $\hat{f}(\mathbf{X})$, it is done using the training set and referred to as training the model. The test set is then used to assess the predictive accuracy of the model.

When evaluating the performance of the model, the training and test sets are used. The optimal model will have good performance on both sets but if it performs well only on the training data, the model is said to be overfitting. This happens when the model is overadapting to the patterns of the training set and not able to generalize well to new data. In other words the fitted model is too *complex*. On the other hand, if the model performs poorly on the training and the test sets, it is said to be underfitting. This occurs when the model fails to learn the relationship between the predictors and the response variable sufficiently well (the fitted model is not *complex* enough).

The expected prediction error, given a specific predictor vector $\boldsymbol{x}_0$, can be decomposed as:

$$\begin{aligned} Err(\boldsymbol{x}_0) &= \mathbb{E}[(Y - \hat{f}(\boldsymbol{x}_0))^2] \\ &= (\mathbb{E}[\hat{f}(\boldsymbol{x}_0)] - f(\boldsymbol{x}_0))^2 + \mathbb{E}[(\hat{f}(\boldsymbol{x}_0) - \mathbb{E}[\hat{f}(\boldsymbol{x}_0)])^2] + \sigma^2(\boldsymbol{x}_0) \\ &= \text{Bias}^2(\boldsymbol{x}_0) + \text{Var}(\hat{f}(\boldsymbol{x}_0)) + \sigma^2(\boldsymbol{x}_0). \end{aligned}$$

There is always an irreducible error $\sigma^2(\boldsymbol{x}_0)$ present in the expected prediction error but the bias and variance behave differently depending on the fit of the model. If overfitting occurs, the expected prediction error will be dominated by the variance since the model is too complex and will be sensitive to changes in the data, meaning variance will be high and bias low. On the contrary, when a model is underfitting, the expected prediction error will be dominated by the bias since the model is not complex enough and the difference between datasets is therefore not significant, meaning high bias and low variance. The bias-variance tradeoff refers to the search of a model that balances between these two scenarios and is illustrated in Fig-

ure 1. In most cases, we are more concerned about overfitting rather than underfitting.

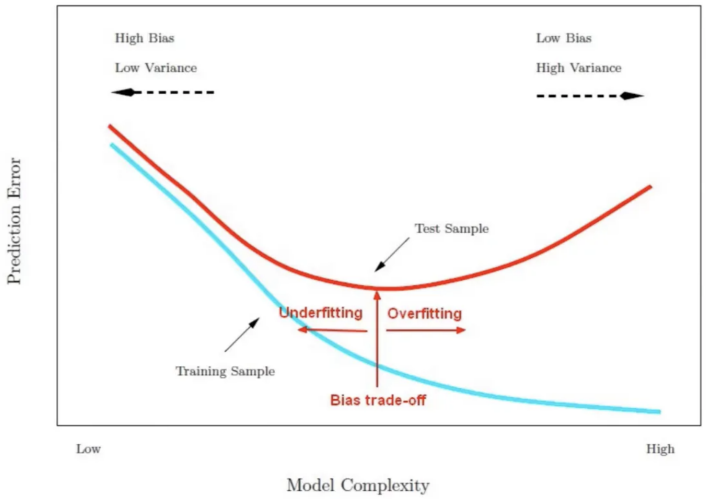

Figure 1: Visual representation of the bias-variance tradeoff. The goal of fitting a supervised machine learning model is to find the middle ground between under- and overfitting, indicated by the vertical uppward pointing arrow. Taken from Hastie et al. (2017).

There are different ways of preventing overfitting, using regularization techniques. Further discussion of regularization is provided in the corresponding subsection for XGBoost and neural network.

2.4 Hyperparameters

In addition to the loss function, machine learning models often include hyperparameters, which control various aspects of the learning process. Hyperparameters are not learned from the data but are set before training and can significantly affect model performance. Common hyperparameters include learning rate, number of hidden layers in neural networks, and maximum tree depth in decision trees or boosting algorithms. The specific hyperparameters used for each model in this thesis will be explained in later sections.

2.5 Neural network

In this section, we will present the theoretical background of the neural network used in this thesis. Unless stated otherwise, the content is based on Goodfellow et al. (2016). Exceptions include Sections 2.5.3 Batch Normalization and 2.5.5 Regularization, which are based on Ioffe & Szegedy (2015) and Srivastava et al. (2014), respectively.

Some readers might be familiar with the term neural network but now we will take a closer look at one of the simpler neural networks called *feedforward* neural network. The name comes from the fact that the information flows explicitly forward through the network from one layer to the next. Each layer consists of a set of *neurons* or *units* that are connected to the units in the previous and next layer. When the units are connected to all units in the previous and next layer the network is said to be *fully connected* (see the left plot of Figure 2 for a simple illustration). Neural networks are composed of three types of *layers*: input, hidden, and output layers. The input layer is the first layer in the network and contains the same number of units as there are features in the input data. Following the input layer are one or more hidden layers, whose number and size (number of units per layer) are not fixed and must be chosen prior to training. The last layer of the network is called output layer and the number of units depends on the problem at hand. For multiclass classification with K classes, the output layer consists of K units where each unit outputs the estimated conditional probability of the input vector of features belonging to its respective class.

Nonlinear functions called *activation functions* (described in more detail in Section 2.5.4) are applied to the output of each layer. The activation function applied to each layer does not have to be the same, however, the activation function for the output layer is related to the loss function used and needs some deliberation. The most common loss function used in neural networks is the *cross-entropy*

$$L(\hat{\boldsymbol{\theta}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k(\hat{\boldsymbol{\theta}}), \tag{1}$$

where $\hat{\boldsymbol{\theta}}$ are the model parameters, K is the total number of classes, y_k is the indicator of the target being of class k and $\hat{y}_k(\hat{\boldsymbol{\theta}})$ is the estimated conditional probability of $Y = (Y_1, \ldots, Y_K)$ being of class k given $\boldsymbol{x}$, i.e. $\hat{y}_k = P(Y_k = 1|\boldsymbol{x}; \hat{\boldsymbol{\theta}})$. In the case of binary classification, we have a single target y that is either one or zero. The target variable is then Bernoulli distributed and has the estimated conditional probability

$$\hat{P}(Y = y \mid \boldsymbol{x}) = \hat{y}^y (1 - \hat{y})^{1-y}. \tag{2}$$

By substituting $\hat{y}_k$ in equation (1) with equation (2) and setting the number of classes K to two, the *cross-entropy* for binary classification can then be written as

$$L(\hat{\boldsymbol{\theta}}) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y})). \tag{3}$$

When we are looking at a training set with multiple independent observations, the total *cross-entropy loss* on the training set is then

$$L(\hat{\boldsymbol{\theta}}) = -\sum_{i=1}^{N} \left[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \right], \tag{4}$$

where N is the total number of observations in the training data. This is what we are trying to minimize when training the neural network.

We now turn our attention to the parameters $\boldsymbol{\theta}$, which are optimized during the minimization of the cross-entropy loss function $L(\boldsymbol{\theta})$. The connections between the units in a layer and the outputs of the previous layer have an attached weight w to them. Each unit in the layer then gets a linear combination of the previous layer's output. There is also a bias b added to each unit. If we are looking at layer l with n_l units and n_{l-1} units in the previous layer, this operation can be written as a matrix multiplication in the following form

$$\boldsymbol{Z}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{X}^{(l-1)} + \boldsymbol{b}^{(l)}, \tag{5}$$

where $\boldsymbol{X}^{(l-1)}$ is the input vector to layer l, $\boldsymbol{Z}^{(l)}$ is the resulting transformed matrix by the layer l, $\boldsymbol{W}^{(l)}$ is a $n_l \times n_{l-1}$ matrix with all the layer l weights $w^{(l)}$ in the following configuration

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n_{l-1}}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n_{l-1}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l,1}^{(l)} & w_{n_l,2}^{(l)} & \cdots & w_{n_l,n_{l-1}}^{(l)} \end{bmatrix}$$

and $\boldsymbol{b}^{(l)}$ is the $n_l \times 1$ bias vector containing the biases b added to each unit transformation. The activation function $g(z)$ is applied element-wise to the vector $\boldsymbol{Z}^{(l)}$ before being passed to the next layer, so that the input to layer $l+1$ is $\boldsymbol{X}^{(l)} = g(\boldsymbol{Z}^{(l)})$. This process of propagating data from the input layer to the output layer is known as *forward propagation*. During training, information from the loss function $L(\boldsymbol{\theta})$ is propagated backward through the network in order to update the estimates of the model parameters $\boldsymbol{\theta}$.

2.5.1 Stochastic gradient descent

When it comes to training a neural network and minimizing the loss function, the most common algorithm is stochastic gradient descent (SGD). It takes advantage of the properties of gradients and iteratively minimizes the loss function by moving in the negative direction of the gradient which we know from calculus is the direction along which a function decreases the most. The gradient $\mathbf{g}$ is computed with respect to all model parameters $\boldsymbol{\theta}$ and has the form

$$\mathbf{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \tag{6}$$

where $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})$ is the gradient of the loss function for a single data point with index i and m is the size of the dataset.

In many machine learning applications, datasets are large, making it computationally unfeasible to use all data points for each update in the gradient descent. This is where the *stochastic* nature of SGD comes into play. Because the loss function is additive over the data points, the gradient can be interpreted as an expectation over the dataset. This expectation can be approximated using a small randomly sampled subset of the dataset known as a minibatch of size m'. The size of the minibatch is a *hyperparameter* that needs to be set before training and is usually chosen to be quite small since it is also offers regularization by introducing noise to the gradient updates (page 272 of Goodfellow et al. (2016)). The gradient in the SGD algorithm has the form described in equation (6) with m' replacing m,

$$\mathbf{g} = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}), \tag{7}$$

where $\{(\boldsymbol{x}^{(i)}, y^{(i)});\, i = 1, \ldots, m'\}$ is a randomly selected subset of data of size m'. The model parameters $\boldsymbol{\theta}$ (the weights w and biases b) are then updated by subtracting a fraction of the gradient in the form

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \mathbf{g},$$

where η is the *learning rate* and it will be decribed in the next subsection. If we look at the updates for weights w and biases b separately they have the form

$$w_{jk}^{new} \leftarrow w_{jk}^{old} - \frac{\eta}{m'} \sum_{i=1}^{m'} \frac{\partial L(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})}{\partial w_{jk}},$$

$$b_{j}^{new} \leftarrow b_{j}^{old} - \frac{\eta}{m'} \sum_{i=1}^{m'} \frac{\partial L(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})}{\partial b_{j}}.$$

2.5.2 Learning rate

In the case of neural networks, the learning rate η is a *hyperparameter* that scales the gradient when updating the weights in the gradient descent algorithm. In other words, it dictates how far along the surface of the loss function we move in each update. The gradient only tells us the direction the optimizer should take but not necessarily the step size. Adjusting the

learning rate can help with convergence and make sure we do not miss any potential local minima of the loss function as well as preventing slow learning and the neural network getting stuck in a suboptimal solution. It is usually set to a small value (around 0.01) but can be adjusted depending on the convergence speed and stability of the training process.

2.5.3 Batch normalization

This section is based on Ioffe & Szegedy (2015) unless stated otherwise.

Although SGD is an effective algorithm, it complicates training because the input to each layer depends on the parameters of all the preceding layers, decreasing stability during training. As the network depth increases, even small changes in earlier layers can be amplified, making optimization more difficult. *Batch Normalizing transform* ($\text{BN}_{\gamma,\beta}$) is a way of reparameterizing the layers by normalizing, scaling and shifting the distribution of the layer outputs in order to achieve faster convergence. It has the following form:

$$BN_{\gamma,\beta}(\mathbf{X}) = \gamma \cdot \frac{\mathbf{X} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta, \tag{8}$$

where $\mathbf{X}$ is the output of a layer, γ and β are trainable parameters, μ_B and σ_B are the mean and standard deviation of the minibatch B and ϵ is a constant added for numerical stability. Faster convergence can be achieved by simply normalizing the output but it may affect the representational power of the layer in the network. The trainable parameters γ and β are added to scale and shift the normalized outputs and mitigate that problem.

2.5.4 Activation function

One of the neural network's main advantages is its ability to model nonlinear functions. This nonlinearity is introduced through activation functions. The most commonly used activation function today is the Rectified Linear Unit (ReLU), defined as $g(z) = max\{0, z\}$ where z is the output of a layer before activation. ReLU is widely used because it enables faster computations due to its piecewise linear nature, which allows for quick gradient calculations. However, since the function is not bounded, the gradient might grow very large in a deeper network and cause unstable learning. This can be solved by *Batch Normalization*. Another activation function that can be used is tanh and has the form $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. In the output layer, the activation function (also called output function) is tied, as mentioned in Section 2.5, to the loss function. When performing binary classification, the sigmoid function is commonly used. It can be viewed as a special case of the *softmax* function

$$\text{softmax}(\mathbf{Z})_i = \frac{e^{Z_i}}{\sum_{j=1}^{K} e^{Z_j}},$$

where K is the number of classes, $Z = (Z_1, \ldots, Z_K)$ are the outputs of all K units of the output layer, before activation. In this thesis, we use the softmax function with two classes, i.e. $K = 2$, although the sigmoid function is also a valid choice for binary classification.

2.5.5 Regularization

This section is based on Srivastava et al. (2014) unless stated otherwise.

To reduce overfitting in neural networks, several regularization techniques can be applied. Common examples include L1 and L2 regularization on the network weights, as well as early stopping when the validation/test error begins to increase. Another widely used technique, particularly in deep networks, is *dropout*. The idea is to improve model performance on test data by averaging the outputs of several models. This technique is also used in other supervised learning methods (for example, in random forests) and is most effective when the individual models differ from each other. In the context of neural networks, however, training multiple models is often computationally expensive and finding optimal hyperparameters for each model can be tedious. *Dropout* offers a way to prevent overfitting without these drawbacks. The term *dropout* refers to the temporary deactivation of a unit within the network. During training the decision of which units should be dropped is random and can be done in the simplest case with a Bernouli(p) distributed random variable where the parameter p is set to a fixed value before training.

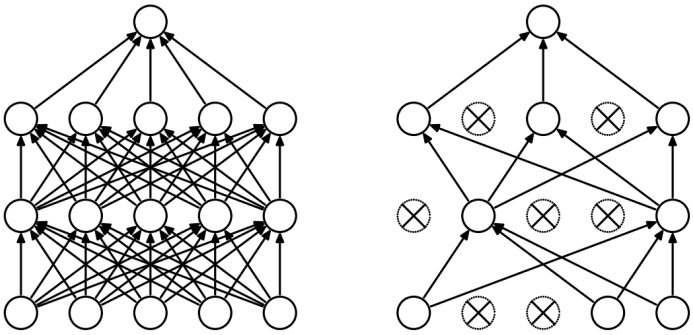

Figure 2: Effect of dropout in a neural network. Taken from Srivastava et al. (2014).

The *forward propagation* in equation (5) is slightly changed when *dropout* is applied. The input vector $\boldsymbol{X}^{(l-1)}$ is replaced with

$$\tilde{\boldsymbol{X}}^{(l-1)} = \boldsymbol{r}^{(l-1)} \cdot \boldsymbol{X}^{(l-1)}, \tag{9}$$

where $\boldsymbol{r}^{(l-1)}$ is a binary vector of size n_{l-1} consisting of independent Bernouli random variables that indicate the probability of the units in that layer not being dropped out.

2.5.6 Optimizer

The SGD with a static learning rate is a solid optimization algorithm but it can be slow. There are a number of strategies for speeding up the learning and model performance by introducing *momentum* and *adaptive learning rate*. A popular and robust adaptive learning rate optimization algorithm is *Adam*, the name derived from adaptive moments. More information about the *Adam* optimizer can be found in Goodfellow et al. (2016) Chapter 8.5.

2.6 Decision trees

In Section 2.6 and 2.7 we will present the theoretical background of decision and boosting trees before introducing XGBoost. Unless stated otherwise, the content is based on Hastie et al. (2017), Chapter 9 and 10.

There are several different tree-based methods, but a popular approach used for both regression and classification, and the one we will focus on, is *classification and regression trees* (also called *CART*).

Tree-based methods work by dividing the feature space into disjoint regions and assigning a simple model to each region. This partitioning is achieved by recursively splitting the feature space into two subregions based on whether a predefined criterion is satisfied or not, a process known as *binary splitting*. This is a greedy algorithm meaning each split minimizes the criterion the most in the current node, even though a different split might produce a better final model. This recursive splitting continues until a stopping rule is applied, creating M regions called *terminal nodes* or *leaves*. Figure 3 illustrates a very simple tree model with two features. The splits made between the initial split and the terminal nodes are called *internal nodes*, and their number is a hyperparameter that can be set prior to training.

The tree-based model can be expressed as

$$f(\boldsymbol{X}) = \sum_{m=1}^{M} c_m I(\boldsymbol{X} \in R_m), \tag{10}$$

where $\boldsymbol{X} = (X_1 \dots X_p)$ is an input vector of the features, c_m is the simple model attached to region m and M is the number of regions the feature space is partitioned into. For classification, c_m will be the predicted class label for all observations in node m.

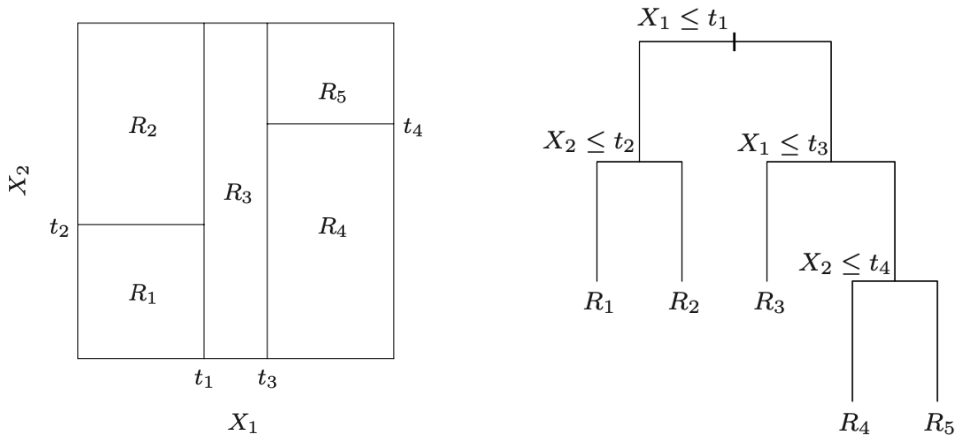

Figure 3: Simple illustration of a tree model with two features X_1 and X_2. The regression tree has $M = 5$ terminal nodes and depth $T = 3$. The left plot shows the terminal nodes created by the tree algorithm and the right plot shows the splitting done in each node of the tree algorithm. Taken from Hastie et al. (2017).

In each binary split, the two subregions R_1 and R_2 created are defined as:

$$R_1(j, s) = \{\boldsymbol{X} \mid X_j \leq s\} \text{ and } R_2(j, s) = \{\boldsymbol{X} \mid X_j > s\},$$

where j is the splitting variable and s is the splitting criterion. In order to optimize this split we seek to find the splitting variable j and split point s that minimize

$$\min_{c_1} \sum_{\boldsymbol{X}_i \in R_1(j,s)} L(y_i, f(\boldsymbol{X}_i)) \quad + \quad \min_{c_2} \sum_{\boldsymbol{X}_i \in R_2(j,s)} L(y_i, f(\boldsymbol{X}_i)),$$

where $L(y_i, f(\boldsymbol{X}_i))$ is some loss function or in this context also described as splitting criterion.

2.6.1 Splitting criterion

When it comes to classification, there are two main splitting criteria:

$$\begin{aligned} \textit{Gini index} \quad &: \quad \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}), \\ \textit{Cross-entropy/deviance} : \quad &- \sum_{k=1}^{K} \hat{p}_{mk} \log(\hat{p}_{mk}), \end{aligned}$$

where K is the number of classes and $\hat{p}_{mk}$ denotes the proportion of observations belonging to class k in node m. The term $\hat{p}_{mk}$ is defined as:

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{\boldsymbol{X}_i \in R_m} I(y_i = k),$$

where N_m is the number of observations in node m and R_m is the region represented by node m. We classify the observations in node m to the majority class of node m, in mathematical terms $k(m) = argmax_k(\hat{p}_{mk})$. *Gini index* and *cross-entropy* are favoured because they are differentiable, a desired property for numerical optimization which XGBoost utilizes. *Cross-entropy* is the loss function used in our neural network and also serves as the splitting criterion in our tree-based model. For binary classification this can be formulated as

$$-\hat{p}_m \log \hat{p}_m - (1 - \hat{p}_m) \log(1 - \hat{p}_m)$$

for region R_m of the tree, where $\hat{p}_m$ is the proportion of observations in class one in R_m.

2.7 Boosting trees

From now on we will consider the binary classification problem where we have the conditional probabilities $c_m = (c_{m_1}, c_{m_2})$ for class 1 and class 2 in region m. Using the fact that the sum of the conditional probabilities is equal to 1, we can rewrite c_{m_2} as $1 - c_{m_1}$ and therefore the value c_m of f in region R_m is a scalar.

A powerful learning idea that combines multiple decision trees in order to produce a better model is called *boosting*. By sequentially fitting a "weak" classifier (classifiers that are only slightly better than the naive approach) to modified versions of the original data, an ensemble of models is created that produce a final combined weighted prediction. For each new "weak" classifier, the data is modified by adding increased weights to previously incorrectly classifed observations that force the next classifier to focus on those observations. A boosted tree model with T additive tree models can be expressed as

$$\hat{y}_i = \hat{f}_T(\boldsymbol{X}_i) = \sum_{t=1}^{T} f_t(\mathbf{X}_i),$$

where $\hat{y}_i$ is the predicted class of observation i with predictor vector $\boldsymbol{X}_i$, f_t is the t-th tree model added and it has the form of equation (10). The boosted tree is grown in a greedy way, meaning in each iteration t the tree model f_t is added that minimizes

$$L^{(t)} = \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(t-1)} + f_t(\boldsymbol{X}_i)), \tag{11}$$

where l is some loss function, y_i is the true class of observation i, $\hat{y}_i^{(t-1)}$ is the boosted tree prediction of y_i from the previous iteration and N is the number of observations in the training data.

2.8 XGBoost

In this section we present the XGBoost algorithm. Unless stated otherwise, the content is based on Chen & Guestrin (2016), the creators of XGBoost. The notation is also inspired from the same paper.

2.8.1 Tree learning algorithm

XGBoost is a tree boosting algorithm that builds upon the principle of gradient boosting and at each step approximates the loss function l using a second-order Taylor approximation. This allows more traditional optimization methods to be used. Given a differentiable convex loss function l, equation (11) can be approximated with a second-order Taylor approximation around $\hat{y}_i^{(t-1)}$ as:

$$\sum_{i=1}^{N} \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(\boldsymbol{X}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{X}_i) \right], \tag{12}$$

where g_i is the first-order derivative $\frac{\partial l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}$ and h_i is the second-order derivative $\frac{\partial^2 l(y_i, \hat{y}_i^{(t-1)})}{\partial^2 \hat{y}_i^{(t-1)}}$, both evaluated at $\hat{y}_i^{(t-1)}$. Since the loss from previous iteration $l(y_i, \hat{y}_i^{(t-1)})$ is constant, it does not affect the optimization in step t and can therefore be removed from equation (12). Therefore, in step t we seek to minimize

$$L^{(t)} = \sum_{i=1}^{N} \left[g_i f_t(\boldsymbol{X}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{X}_i) \right]. \tag{13}$$

Using the definition of trees in equation (10) and the fact that the regions in a tree are disjoint, we can rewrite equation (13) as

$$L^{(t)} = \sum_{i=1}^{N} \left[g_i \sum_{m=1}^{M_t} c_m I(\boldsymbol{X}_i \in R_m) + \frac{1}{2} h_i \sum_{m=1}^{M_t} c_m^2 I(\boldsymbol{X}_i \in R_m) \right]. \tag{14}$$

XGBoost adds an additional regularization term $\Omega(f_t)$ to the above objective function to help with overfitting and smooth the weights of the final model. $\Omega(f_t)$ is defined as

$$\Omega(f_t) = \Gamma M_t + \frac{1}{2}\lambda \sum_{m=1}^{M_t} c_m^2, \tag{15}$$

where Γ and λ are regularization hyperparmeters that control the size of the trees (discussed further in the next section), M_t is the number of terminal nodes in the tree and c_m is the weight of the corresponding node. Adding $\Omega(f_t)$ for step t to equation (14) we get

$$\begin{aligned} L^{(t)} + \Omega(f_t) &= \sum_{i=1}^{N} \left[g_i \sum_{m=1}^{M_t} c_m I(\boldsymbol{X}_i \in R_m) + \frac{1}{2} h_i \sum_{m=1}^{M_t} c_m^2 I(\boldsymbol{X}_i \in R_m) \right] \\ &\quad + \Gamma M_t + \frac{1}{2}\lambda \sum_{m=1}^{M_t} c_m^2 \\ &= \sum_{m=1}^{M_t} \left[c_m \sum_{\boldsymbol{X}_i \in R_m} g_i + \frac{1}{2} c_m^2 \sum_{\boldsymbol{X}_i \in R_m} h_i \right] + \Gamma M_t + \frac{1}{2}\lambda \sum_{m=1}^{M_t} c_m^2, \end{aligned} \tag{16}$$

where $\boldsymbol{X}_i \in R_m$ denotes the predictor vector of observation i in region R_m. By moving the last term of the expression into the parenthesis we get

$$\sum_{m=1}^{M_t} \left[c_m \sum_{\boldsymbol{X}_i \in R_m} g_i \; + \; \frac{1}{2} c_m^2 (\lambda + \sum_{\boldsymbol{X}_i \in R_m} h_i) \right] + \Gamma M_t. \tag{17}$$

To simplify notation let $G_m = \sum_{\boldsymbol{X}_i \in R_m} g_i$ and $H_m = \sum_{\boldsymbol{X}_i \in R_m} h_i$. In other words let G_m be the sum of the first-order derivatives of the observations in region R_m for boosting iteration t and H_m the corresponding sum of second order derivatives. Equation (17) can then be expressed as

$$\sum_{m=1}^{M_t} \left[G_m c_m \; + \; \frac{1}{2}(H_m + \lambda) c_m^2 \right] + \Gamma M_t. \tag{18}$$

Since the objective function is a convex differentiable function, the optimal terminal node weights c_m can be calculated by computing the derivative of expression (18) w.r.t. c_m, set it equal to zero and solve for c_m. We then get the optimal terminal node weights

$$c_m^* = -\frac{G_m}{H_m + \lambda}.$$

Given c_m^*, we can compute the best possible loss reduction in a given boosting iteration t by substituting c_m in equation (18) with c_m^*:

$$-\frac{1}{2}\sum_{m=1}^{M_t}\frac{(G_m)^2}{H_m+\lambda}+\Gamma M_t. \tag{19}$$

This score (expression (19)) is used as a measure of performance and is helpful when calculating potential splits at each iteration. In a greedy process, from one single node, new nodes are added based on the best possible loss reduction (expression (19)) of each split, also called *gain*. In each split into the left L and right R node, the *gain* can be expressed as

$$gain=\frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right]-\Gamma, \tag{20}$$

where the first term is the *gain* from the left node, the second term is the gain from the right node and the last term is the *gain* from not splitting the node. From equation (20), we see that if the *gain* from splitting into the left and right node is not higher than the *gain* of not splitting, the tree does not grow (given that Γ is set to zero). The splitting of nodes for each tree f_t of the XGBoost algorithm continues as long as the *gain* for the best proposed split is positive.

2.8.2 Approximate tree learning

When the split points are proposed in the boosting algorithm, it is usually done in a greedy manner, meaning each split point is visited in order to find the optimal split; a powerful but very computationally expensive algorithm. XGBoost uses an *approximate* algorithm that proposes split points first based on percentilies of the feature distributions. It then discretizes the continuous features into buckets defined by these candidates, aggregates statistics within each bucket, and selects the best split based on the aggregated data (Chen & Guestrin, (2016) Section 3.2).

2.8.3 Sparsity aware splitting

XGBoost also includes a default direction for each split to be taken for sparse data inputs. Sparsity can be caused by missing values in the data or other structural properties of the data such as one-hot encoding. By adding a default direction for the algorithm to take when there are missing values the model is made aware of the sparsity structure in the data. The default direction is chosen by calculating the *gain* of the left and right direction based on the available data, given that all missing values are assigned to the left and right direction. The direction that has the highest *gain* is chosen as the default direction.

2.8.4 Hyperparameters for growing trees and regularization

Number of trees, T

The number of trees or number of boosting rounds is one of the main regularization techniques for tree methods. With each iteration a tree is added and the model complexity is increased, achieving a better fit of the data. However after some number of trees the variance increases (bias-variance tradeoff, Section 2.3) with each subsequent iteration and reduces the performance on unseen data. By choosing an appropriate value for T, overfitting can be avoided.

Learning rate, η

The learning rate is used in the neural network and discussed in Section 2.5.2. In the tree model it has a similar purpose, to regulate the influence and magnitude of the tree added in each boosting iteration.

Maximum depth, T_{max}

Maximum depth is a hyperparameter that controls the maximum depth of all trees in the boosted tree. The value of T_{max} will be individual for each dataset but according to Hastie et al. (2017) values between 4 and 8 work well in the context of boosting and adding more depth rarely shows significant improvement.

Gamma, Γ

Γ is a regularization parameter that controls the size of the tree by penalizing the *gain* of a potential node split. In equation (20), Γ acts as a threshold where a split will occur only if the resulting *gain* exceeds this value. This helps to limit the tree's complexity and reduce overfitting.

L2 regularization, λ

The regularization term $\Omega(f_t)$ (equation (15)) consists of two terms where the first includes Γ and the second includes λ. λ is also a regularization parameter that controls the size of the weights of the terminal nodes $\sum_{m=1}^{M_t} c_m^2$. In equation (20) we also see that an increase in λ influences the *gain* of a split and therefore affects the structure and complexity of the tree.

L1 regularization, α

In the paper by Chen & Guestrin (2016) that this section is based on, there is no mention of L1 regularization, α. However, in the documentation of the XGBoost library for python there is a parameter `reg_alpha` that allows

for L1 regularization. It is similar to L2 regularization, with the difference that the weights may be reduced to zero. The L1 regularization term has the form $\alpha \sum_{m=1}^{M_t} |c_m|$.

Subsampling

The model performance can be improved by only using a subset of the available data. In each boosting iteration, a fraction of the data is chosen and used to fit the added tree. The subsample can be chosen by only using a fraction of the features and/or a fraction of the samples in the dataset. Using a subsample to grow each tree reduces the correlation between each tree which helps to reduce overfitting (Hastie et al. (2017)).

2.9 Betting

In this section we will shortly discuss some concepts of betting that are used in the modeling and the betting strategy when evaluating the models.

2.9.1 Odds

In the world of betting, the likelihood of an outcome is expressed using odds. According to Cortis (2015), there are three main ways of presenting these odds: European, English and American odds. These odds describe the return of a wager should you win and are calculated with the probabilities that the bookmakers believe the outcome has. We will be using European odds (decimal odds) and they are defined as the inverse of the probability,

$$\text{odds} = \frac{1}{\text{probability}}.$$

Solving for the probability we get

$$\text{probability} = \frac{1}{\text{odds}}. \tag{21}$$

There are two types of bookmakers: soft and sharp. Sharp bookmakers are known for providing fast moving odds that are close to the true probability of the outcome, while soft bookmakers focus on attracting casual bettors by offering more attractive odds with a higher profit margin. In this thesis, we utilize the odds provided by Pinnacle, a sharp bookmaker, to estimate probabilities through equation (21), which are then added as features in our models.

2.9.2 Value bet

The expected value (EV) of a bet is a formula that gives the value of a bet, given the odds offered by the bookmaker and the probability of the outcome. The EV formula is defined as

$$\text{EV} = p \cdot o_b - 1, \tag{22}$$

where p is the probability of the outcome and o_b is the odds offered by the bookmaker. If the EV is positive, the bet holds value and the expected return of a bet placed on the outcome is positive. The probability p, in our case, will be the probability predicted by our models. For a more intuitive description of EV, we can use equation (21) to replace p in the EV formula (22). We then get

$$\text{EV} = \frac{o_b}{o_m} - 1, \tag{23}$$

where o_b are the odds provided by the bookmaker and o_m are the implied odds of our model. We see that the EV is only positive if the odds provided by the bookmaker are higher than the implied odds of our model. If an outcome meets this condition and is our model's predicted outcome, we place a bet on it.

3 Analysis

In this section, we will present and analyze the data used in this study and the paremeter choices for each model.

3.1 Data

When it comes to data from football matches, there are multiple sources, such as websites with downloadable datasets containing various match statistics. However, these APIs are most often not free. For this thesis we use datasets containing stats from different seasons and leagues, downloaded from *Football-Data.co.uk* and combine them in order to get a final dataset.

The downloaded datasets contain general information like date, home and away team names as well as a collection of statistics from each match (see Table 5 in the Appendx for the statistics available in the dataset and their corresponding descriptions). Additionally, the datasets provide odds from multiple bookies for several different markets such as match outcome (home/away win or draw), totals (more or less than 2.5 goals in the match) and etc. These odds also provide implied probabilities for the different match outcomes, which will serve as part of the input to the models. The dataset includes both opening and closing odds, that is, the initial odds offered by the bookmakers and the final odds available just before the start of the match, respectively.

In order to obtain a sufficiently large dataset, we consider data from the 2017/2018 season up to the current 2024/2025 season (eight seasons) for the five highest-ranked leagues in Europe: the English Premier League,

Italian Serie A, Spanish La Liga, German Bundesliga and French Ligue 1. The first seven seasons are used for the training and validation set and the eighth season (current 2024/2025 season) will be used as a test set. Since the current season is not finished during the process of writing this thesis, we consider the matches played from August 15th to March 16th. The final dataset consists of 13931 observations (matches) and 155 columns. The choice of number of seasons used is purely based on the desire to use a sufficiently large dataset when training the models and can be set to a different number.

3.1.1 Exploratory data analysis

Before discussing the hyperparameters and architectures of the models we aim to employ, we first examine the characteristics of the data to gain a better understanding of its structure.

Missing values

Missing values need to be addressed before model selection. Although XGBoost can handle missing data internally, other models, such as the neural network we employ, cannot. The dataset contains 810245 missing values, most of which correspond to various bookmaker odds that are not considered in our models. However, there are 3712 observations missing values for Pinnacle's over/under 2.5 goals odds. Six of these observations are also missing values for Pinnacle's home win, away win, and draw odds, and an additional eight observations are missing values for these odds as well. The rest of the missing values correspond to the match referee and the start time of matches. Importantly, all columns related to match statistics are complete and contain no missing values. When handling missing values, various strategies can be applied. However, because odds are highly specific to each match, there are no straightforward or reliable methods to fill these missing values, such as using average odds for each team or league. Additionally, when looking at these missing values we see that all 3652 matches from seasons 2017/2018 and 2018/2019 are missing values for Pinnacle odds, making these observations useless in our models. Therefore, we do not consider these observations as well as the remaining 68 observations from the six other seasons that are missing the Pinnacle odds. The final dataset that is used for training and testing the models contains 10211 observations from the current as well as the past five seasons.

Class distribution

By examining the frequency of match outcomes (more or less than 2.5 goals) in the dataset, we obtain the empirical distribution of the two classes. As shown in Figure 4, the dataset is fairly balanced, with around 46.8% of

matches having fewer than 2.5 goals and around 53.2% having more than 2.5 goals. These percentages suggest that the probability of a match having more than 2.5 goals is only slightly higher than that of having fewer than 2.5 goals.

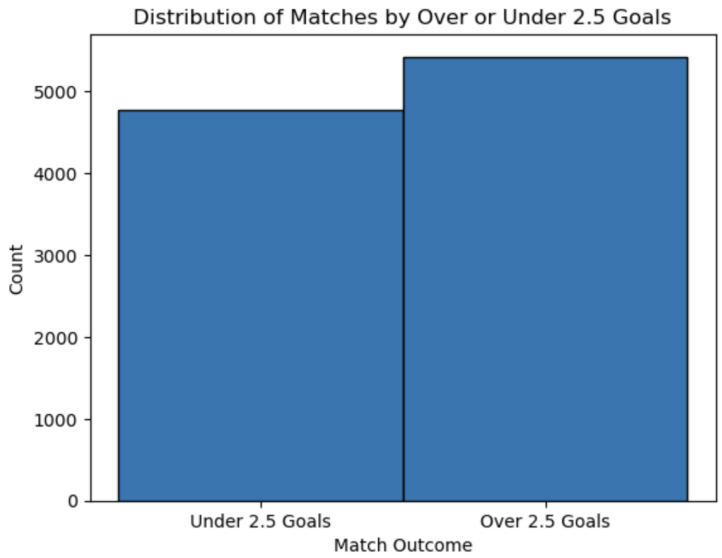

Figure 4: Class distribution of the dataset. 4776 matches had less than 2.5 goals and 5435 had more than 2.5 goals.

3.1.2 Feature engineering

The statistics included in the dataset are match-specific and primarily describe the performance of the two teams in a single match. While these values can indicate how many goals might be expected in that particular match, they provide limited information on whether the next match will have many goals. For the purpose of future predictions, we want features that reflect each team's underlying performance during the season as these metrics are more likely to indicate whether a future match will have many goals or not. Table 1 includes all features created from the original dataset, that will be considered for our models.

Missing values

Since some features are based on rolling averages, the first five matches of each team in each season do not have values for those features. To avoid losing additional data by discarding these rows, we fill the missing values with the average value of the respective feature for each team in each season.

Table 1: The features created and their corresponding descriptions.

Feature	Description
League2.5Perc	League-wide percentage of games with over 2.5 goals
HomeOver2.5Perc	Home team percentage of games with over 2.5 goals
AwayOver2.5Perc	Away team percentage of games with over 2.5 goals
HTHomeOver2.5Perc	Home team percentage of games with over 2.5 goals at half time
HTAwayOver2.5Perc	Away team percentage of games with over 2.5 goals at half time
AvgLast5HomeGoalsScored	Avg. goals scored by home team (last 5 matches)
AvgLast5HomeGoalsConceded	Avg. goals conceded by home team (last 5 matches)
AvgLast5AwayGoalsScored	Avg. goals scored by away team (last 5 matches)
AvgLast5AwayGoalsConceded	Avg. goals conceded by away team (last 5 matches)
AvgLast5HomeShots	Avg. shots taken by home team (last 5 matches)
AvgLast5HomeShotsConceded	Avg. shots conceded by home team (last 5 matches)
AvgLast5AwayShots	Avg. shots taken by away team (last 5 matches)
AvgLast5AwayShotsConceded	Avg. shots conceded by away team (last 5 matches)
AvgLast5HomeCorners	Avg. corners by home team (last 5 matches)
AvgLast5AwayCorners	Avg. corners by away team (last 5 matches)
HomeBTTS_Perc	Home team percentage of BTTS (both teams to score)
AwayBTTS_Perc	Away team percentage of BTTS (both teams to score)
Last5HomeShotsPerGoal	Avg. shots per goal for home team (last 5 matches)
Last5AwayShotsPerGoal	Avg. shots per goal for away team (last 5 matches)
HomeSuspensionProbability	Probability of home team player suspension
AwaySuspensionProbability	Probability of away team player suspension
ImpliedProbabilityOver	Pinnacle's implied probability of over 2.5 goals
ImpliedProbabilityHomeWin	Pinnacle's implied probability of home win

Correlation

Before we move on to modeling, we also want to consider the correlation between the different features and the target variable. Strong correlation between features can be problematic and reduce model performance. If there is a strong correlation between features, some feature selection or adjustments to the models need to be made. Figure 5 shows the correlations between the features and the target variable (the traget variable is labeled Over2.5). Some features have a notable correlation but not strong enough to justify any actions.

3.1.3 Test/train split

We mentioned in Section 2.3 that before modeling, the dataset is split into two or three parts. We split our full dataset into training set, validation set

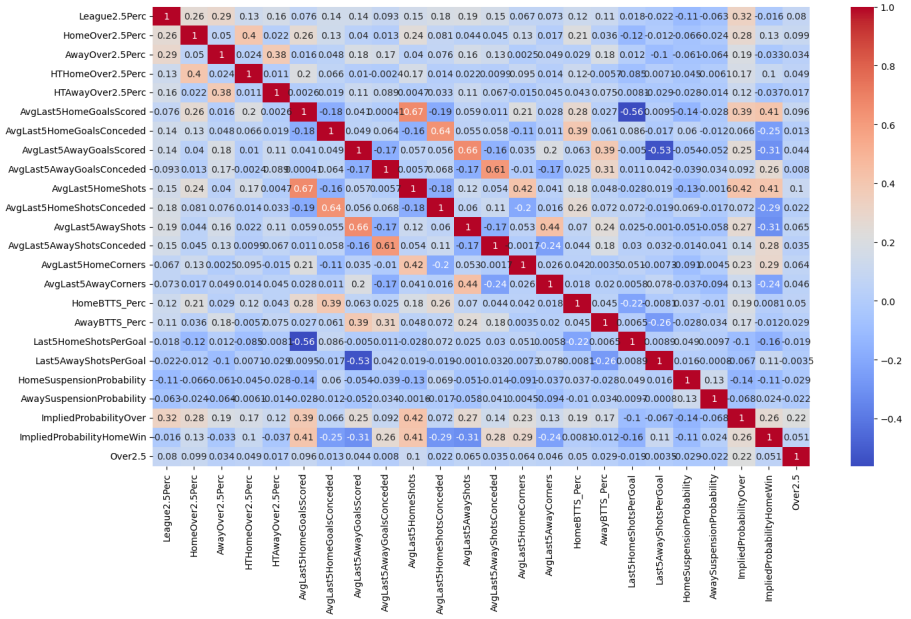

Figure 5: Correlation between the features and the target variable. Over2.5 is the target variable.

and test set. The validation set is used to evaluate the predictive performance of the models while the test set is used for testing the profitability of our models compared to the profit from following Pinnacle's odds. By using two sets for evaluating the models on unseen data we can check that the performance of the model is not subject to randomness. The test set contains the matches from the current 2024/2025 season. The previous five seasons are used for the training set and validation set and they are divided in the following way: each of the five seasons is randomly split into two sets where set a_i contains 80% of season i:s matches where $i = 1, ..., 5$ and set b_i contains the remaining 20%. All a_i sets are concatenated and used as the training set. Equally, all b_i sets are concatenated and used as validation set. The split is done in this manner in order to keep the number of observations from each season the same in the training and validation set. This helps eliminate any bias a single season might have on the model's performance. Further discussion on train/test split can be found in Section 5.

3.1.4 Standardization

Lastly, before we look at the model selection, the dataset is standardized. For tree methods and neural networks this is not necessary due to the structure of these models, minimizing the effect big-scale differences in the features can have on model performance but it is good practice. Standardization can also help with convergence, since the gradient sizes are

smaller. It is important to note that standardization should only be applied to continuous features and not categorical. All our features are in fact continuous and therefore will be standardized, except for the four features that represent probabilities: 'ImpliedProbabilityOver', 'ImpliedProbability-HomeWin', 'HomeSuspensionProbability' and 'AwaySuspensionProbability' since they already only take values between 0 and 1.

3.2 Model parameters

For the practical implementation of the models, the programming language python is used along with the libraries *tensorflow* and *XGBoost* for the neural network and the XGBoost model respectively. We will now describe the models used and the hyperparameter tuning done to try and improve them.

Neural network

When it comes to neural network architecture, there are not many definitive guiding theoretical principles, however, there are recommended guidelines and ideas for how the layers should be placed and grown from layer to layer.

The input layer has the same number of units as the number of features in our dataset and the output layer has two units corresponding to the two classes but the hidden layers (the layers between the input and output layer) can have very different sizes. Since we want to "untangle" the datapoints, we aim to increase the dimensionality of the original feature space in order to get clearer separation between the observations of different classes. After adding enough layers with increasing size, the next layers have a decreasing number of units (reducing dimensionality) until we reach the output layer. These increases and decreases of unit size can be done in drastic ways, however, it is recommended to change the number of units between two layers with no more than a factor of two when increasing and a factor of 0.5 when decreasing the number of units.

The optimal number of hidden layers depends on the dataset, and there is no universal rule. Sufficient depth is necessary for class separation, but too many layers can harm performance and convergence and may require more advanced techniques, such as *skip connections*, that are beyond the scope of this thesis.

We start by using a neural network with 5 hidden layers and modify the architecture depending on the result. We use `ReLU` activation and add batch normalization to all hidden layers as well as the input layer before the activation for better performance and convergence (see Sections 2.5.3 and 2.5.4). After each hidden layer except for the last one we also add dropout in order to combat overfitting. The percentage of neurons to drop out after each layer is different and will be tweaked depending on the model

performance (if the model seems to be severely overfitting, the percentage of dropped out neurons will be increased and vice versa). The epochs and batch size will also be tweaked depending on model performance and loss convergence.

The initial network consists of five hidden layers $l_1, ..., l_5$ with 32, 64, 32, 16, 8 units respectively. Additional model parameters are listed in Table 2.

Table 2: Initial neural network parameters.

Parameter	**Value**
Epochs	90
Batch size	16
Loss function	Cross-entropy
Activation function	ReLU
Output function	Softmax
Training algorithm	SGD with Adam optimizer

XGBoost

There are different strategies for hyperparameter tuning of XGBoost models (among other supervised learning models) with no universally optimal approach. Grid search and random search are viable strategies however, we want to monitor the learning curve (how the loss changes with each boosting iteration) in order to see how the loss converges as well as understand how the different hyperparameters affect model performance. Therefore we will manually tweak the hyperparameter values in order to understand how the performance changes when increasing and decreasing these hyperparameter values. The hyperparameters we consider are listed in Table 3.

Table 3: XGBoost hyperparameters considered.

Hyperparameter
Number of trees (T)
Learning rate (η)
Maximum tree depth (T_{max})
Subsample ratio of training observations
Subsample ratio of columns per tree
Minimum loss reduction (Γ)
L1 regularization term (α)
L2 regularization term (λ)

4 Results

In this section we present the results of the implemented classifiers. Further discussion of the results can be found in Section 5.

4.1 Model optimization and hyperparameter tuning

We start by looking at model performance and the effect of hyperparameter tuning.

4.1.1 Neural network

In Section 3.2 we define the initial neural network to be trained. Figure 11 in the Appendix illustrates the training and validation curves of this network. There are signs of overfitting and also very little decrease in both training and validation loss as the epochs are increased. The model is simply not learning much from the training data. We tried to improve performance and the network's ability to separate the two classes by adding two more layers, resulting in seven hidden layers where layers $l_1, ..., l_7$ have 32, 64, 128, 64, 32, 16, 8 units, respectively. Adding more layers would require, design changes that are outside of the scope of this thesis, like *skip connections* and not necessarily improve model performance. The difference in performance was marginal with the same simultaneous overfitting and underfitting problem present. Further experimentation with changes in learning rate, batch size and dropout percentages was done with no significant improvement. Finally, a more aggressive change in units from layer to layer was tested as well as trying `tanh` as the activation function. Again, no significant improvement was observed although the test accuracy was marginally improved. Therefore, we settled on a neural network with four hidden layers where layers $l_1, ..., l_4$ have 64, 128, 64, 16 hidden units, respectively. The additional parameters included in Table 2 remain the same apart from the number of epochs and activation function, which are changed to 70 and `tanh` respectively.

4.1.2 XGBoost

As we mentioned in Section 3.2 we manually tune the hyperparameters. It is not an exhaustive search since there is a huge number of hyperparameter combinations. However, we iteratively tweak the hyperparameters depending on how the training curve looks until the model is no longer improved or we are satisfied with the result. The results from this hyperparameter search are the following:

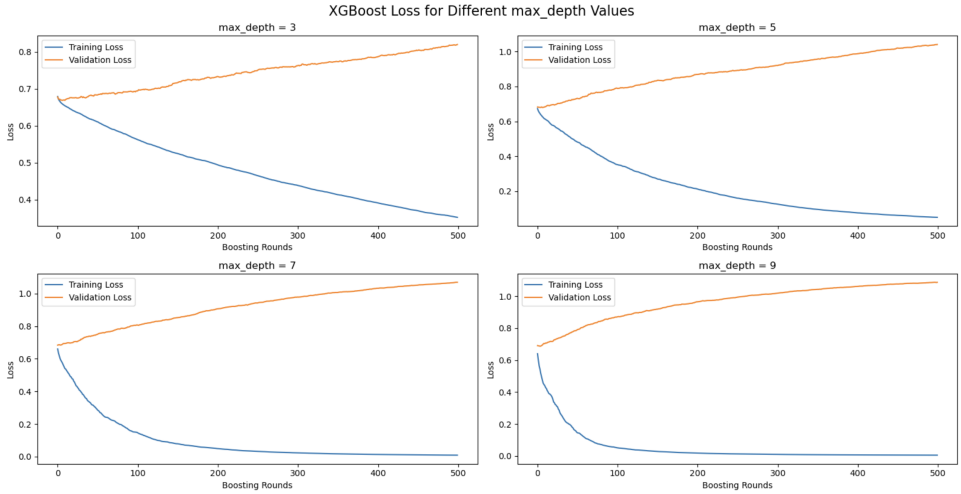

Figure 6: Training curves for four different values of tree depth for XGBoost.

Maximum tree depth (T_{max})

One of the most important hyperparameters for model training is maximum tree depth T_{max}. When setting all other hyperparameters to their default values and only changing T_{max}, we obtain the training curves in Figure 6. Similar plots for changes in regularization hyperparameters λ and α can be found in the Appendix. The plots in Figure 6 show extreme overfitting in all four cases and an increased separation between training loss and validation loss as T_{max} is increased. We also note that out of the four values tested for T_{max}, $T_{max} = 3$ is the only one where the validation loss decreases at all (indicating that the model training does somewhat improve performance on unseen data). Therefore we set $T_{max} = 3$ and focus on regularization to reduce overfitting.

Regularization

The default value of the learning rate η is 0.3; quite a high value which decreases the number of boosting rounds needed. However, a lower learning rate combined with a higher number of boosting iterations is usually recommended for better results and therefore we decrease η to 0.05. The two main regularization hyperparameters we use are L1 regularization α and L2 regularization λ (see Section 2.8.4). Figures 12 and 13 in the Appendix, illustrate the difference in the training curves when λ and α are changed separately. The smallest validation loss does not seem to change much when increasing λ from 10 to 1000 (the difference in cross-entropy loss is 0.00354) but the boosting iterations at which these minima are attained are quite different.

The difference in validation loss when α is increased from 0.1 to 100 is illustrated in Figure 13. Similarly to λ, the validation loss does not change

significantly. However, when α is large, the training loss and validation loss remain constant after 76 boosting iterations.

We experimented further by also changing the learning rate, as well as the row sub-sampling ratio, in order to try and reduce overfitting further. As we mentioned in previous sections, this is not an exhaustive analysis of all possible hyperparameter combinations and there may be better hyperparameter configurations than what we have explored. However, this is the best possible XGBoost classifier we obtained in this thesis.

In Table 4 we present the final XGBoost classifier that is used to test its profitability on the test set.

Table 4: Final XGBoost model hyperparameter values.

Hyperparameter	**Value**
Number of trees (T)	177
Learning rate (η)	0.06
Maximum tree depth (T_{max})	3
Subsample ratio of training observations	0.55
Subsample ratio of columns per tree	0
Minimum loss reduction (Γ)	0
L2 regularization term (λ)	0
L1 regularization term (α)	30

4.2 Model performance

Given the final neural network and XGBoost classifiers, we now evaluate their predictive performance. On the validation set, the neural network and the XGBoost model achieve an accuracy of around 60.36% and 60.98% respectively. These accuracies are quite poor and all other models tested have similar performances with one to three percent lower accuracies. In Figure 7, the confusion matrix for each classifier on the test set is illustrated. We see that the XGBoost classifier performs better at predicting matches with more than 2.5 goals (class 1) than it does at classifying matches with fewer than 2.5 goals (class 0). It manages to only classify 50.4% of the class 0 observations correctly and 67.7% of the class 1 observations. The neural network has very similar performance with only 4 more misclassifications.

When the models are evaluated on the test set, we get similar accuracies (the neural network and XGBoost model have around 59.36% and 59.66% accuracy respectively) indicating that the performance of the models is the result of model training and not randomness.

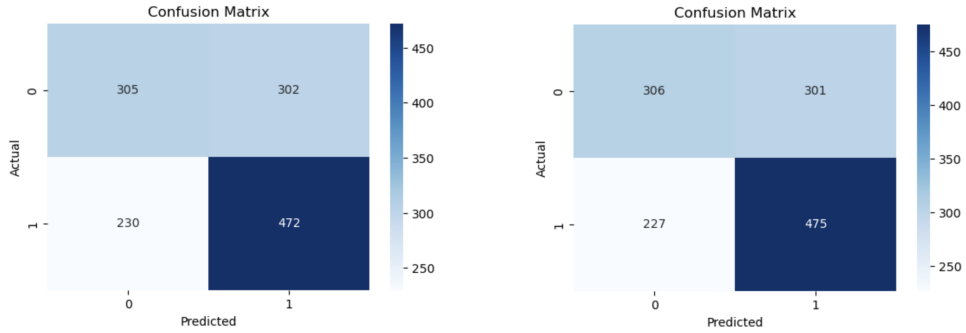

Figure 7: Confusion matrices for both classifiers. The confusion matrix of the neural network is shown in the left plot and the confusion matrix of the XGBoost model in the right plot.

4.3 Betting profits on test set

The original goal of this thesis was to determine whether machine learning models can be used to achieve profitability in football betting. Figure 8 illustrates the cumulative profit on the test set, for each classifier. The following betting strategy is applied: for every match (observation in the test set) the expected value (EV) of a bet is calculated (see Section 2.9) for the outcome predicted by each model (more or less than 2.5 goals). If the EV is positive, a 1000 SEK bet is placed on that outcome, otherwise no bets are placed on the match. In other words, we place a bet if the odds offered by Betfair are higher than our model's implied odds for its predicted outcome, otherwise, no bet is placed. The XGBoost model had a cumulative profit of 47450 SEK and collected over 50% of that cumulative profit in the first approximately 150 matches. The neural network made a similar cumulative profit of 41480 SEK and was very profitable in the first 300 matches, however, after 1000 matches it was quite unsuccessful.

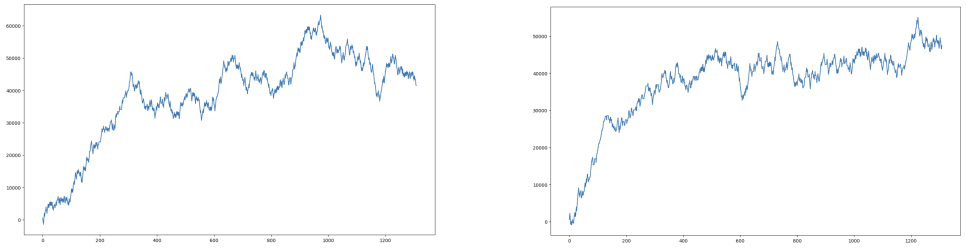

Figure 8: Comparison of profits predicted by the two classifiers. The profits from the neural network are shown in the left plot and the profits from the XGBoost model in the right plot.

For comparison, the same betting strategy was applied but instead of using our model probabilities, the implied probability from Pinnacle's odds

were used when calculating the (EV). The cumulative profit of this strategy is illustrated in Figure 9. It shows that if a bettor were to consistently bet 1000 SEK on the outcome with the lowest Pinnacle odds, when Betfair's odds for the same outcome are higher, their cumulative profit would be 39040 SEK. Comparing the plots from Figure 8 with the plot from Figure 9 we see that both our models outperform the strategy of following Pinnacle's odds. Using XGBoost had a 22.3% higher profit than using Pinnacle's odds while the neural network only was slightly more profitable (around 6%). Notably all strategies make a large part of the cumulative profit early in the season and then continue to follow a positive trend, albeit with clear fluctuations. However, only the XGBoost classifier does not see a significant decline in profitability at the final stages of the season. Although the test set does not cover the whole season (until mid March) it still represents the later stages of the season.

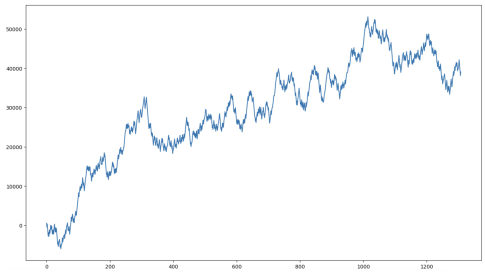

Figure 9: Profit made by placing bets based on odds given by the sharp bookmaker Pinnacle.

5 Discussion

In this section, we examine the work done in this thesis and discuss future improvements.

5.1 Summary

In this thesis, we have studied a binary classification problem using two supervised learning models: a fully connected feedforward neural network and an XGBoost model. The goal was to predict whether a football match will have more or less than 2.5 goals. To do this, we downloaded datasets from *Football-Data.co.uk*, conducted an exploratory data analysis and created new features that would improve the predictive performance of the models. After model training and hyperparameter tuning, we evaluated their accuracy on a validation set where the neural network achieved 60.36% accuracy and the XGBoost classifier achieved 60.98% accuracy. The models were then used on a test set where their profitability was examined by simulating bets on

the over/under 2.5 goals market. Both classifiers were profitable and beat the strategy of following the odds of the bookmaker Pinnacle, with XGBoost making around 22% more profit and the neural network around 6%.

5.2 Model performance

As mentioned in Section 4, the predictive performance of the models is quite poor. While they outperform the naive approach of random guessing, the improvement is marginal. Notably, the training loss and accuracy of our two classifiers are quite similar. For comparison, three additional models were trained with limited hyperparameter tuning: logistic regression, random forest and support vector classifier (SVC). More information on these models can be found in Hastie et al. (2017). The models achieved very similar accuracies, ranging from approximately 57% to 60%, comparable to the two classifiers used in this thesis. This suggests that we may be approachig the limit of possible performance given the dataset and feature set at hand. The fact that the training and validation loss curves for both models have minimal decrease as the epochs and boosting iterations are increased, enforces the idea that the data does not include enough information to separate the two classes.

Additionally, we created a synthetic dataset to determine if the issue of limited learning of the neural network is due to its design or the nature of the dataset at hand. The synthetic dataset had also two classes, clearly separable with the same class distribution as the downloaded dataset, and the same number of features (23). The data points were sampled from a multivariate normal distribution $N(\mathbf{0}, \boldsymbol{I})$ and assigned to class 1 if the norm of the input vector was between 0 and 3.5 and to class 0 if the norm was between 3.5 and 4. On the synthetic dataset, the neural network achieved an accuracy of 74%. This result further suggests that the feature set created or/and the original dataset is limiting the model's effectiveness to separate the two classes.

In Section 4, the confusion matrices are illustrated and we noted that both classifiers only predicted around 50% of the matches with less than 2.5 goals (class 0) correctly, while having noticeably better success at prediciting observations of class 1. This model performance (one class having significantly better performance) is similar to what you would expect to see when dealing with an imbalanced dataset, which our dataset is not. However, class 1 is majority class by a small margin. In order to try and increase the XGBoost classifier's performance on class 0, we added a balancing weight. XGBoost has a parameter called `scale_pos_weight` that is used for unbalanced datasets and controls the balance of positive and negative weights. We set this parameter to $\frac{\text{sum of observations of class 1}}{\text{sum of observations of class 0}} \approx 1.13$. The model then places 1.13 times more importance on the negative weights (weights applied on the minority class). The confusion matrix for this model can be seen

in Figure 10. The model performs better on class 0 but sacrifices some performance on class 1. The overall accuracy is very similar with 59,91% and 59.74% on the validation set and test set respectively. However, the profit made by this model on the test set is significantly less, 20540 SEK. Further investigation on the difference in prediction probabilities is needed to understand this behavior.

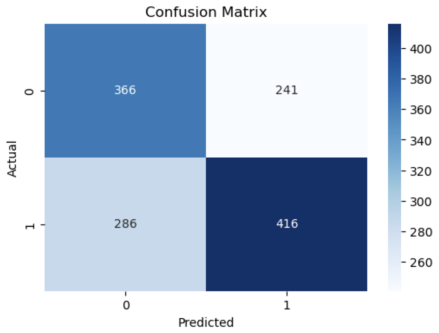

Figure 10: Confusion matrix for XGBoost classifier with `scale_pos_weight` set to 1.13 in order to improve model performance on class 0.

No detailed analysis of feature importance was done but the average gain (equation (20) in Section 2.8.1) of each feature when used was investigated. The feature that had the highest average gain was the implied probability from Pinnacle's odds for the outcome of more than 2.5 goals. It was significantly higher than all other features with an average gain of around 9.78 while all other features had values in the range of 1-3. This means that it has a big impact when used for splitting, during the growing of the trees. However, when removed from the feature set, the accuracy for the XGBoost classifier on the validation set did not change significantly but on the test set the accuracy dropped to 55% - 56%. This suggests that the implied probability from Pinnacle's odds for over 2.5 goals is more important for the 2024/2025 season (which is the test set).

Betting Profitability

During experimentation and hyperparameter tuning, we saw that the model accuracy did not change drastically with each try. However, for the XGBoost classifier, the profitability varied significantly from model to model. Small changes in for example the regularization parameter α had profit decreases of 60% compared to the profit from our final XGBoost classifier (if not more in some cases). Notably, we found an XGBoost classifier with higher validation accuracy (around 0.3% higher) than our final XGBoost classifier but it had around 40% less profit. The neural network showed similar problems with fluctuating profits when changing the architecture, which is to be expected, though not as severe when making minor parameter changes like batch size.

Due to time constraints, no further analysis was done on the bets that made these differences or how the model probabilities differed between models.

5.3 Possible improvements

Intuitively, predicting any outcome of a football match is difficult. Even if you have tactical knowledge or a model to help you make predictions, there are uncontrollable factors that can change the direction of a match in a heartbeat, such as a red card, a penalty or a last minute winner. This notion has been partly reinforced by the results of this thesis, although the results indicate that our models have learned from data to some extent. Throughout the thesis, various efforts were made to improve model performance, however, there are still several changes that could be explored in order to try and improve the results even further.

Data and features

The main improvement would be to obtain data or features that better represent the classes. The features created in this thesis should be thoroughly investigated in terms of feature importance and modified or changed accordingly. Since the methodology of feature engineering is not a one-size-fits-all process, further experimentation and consideration into what features could help explain the difference between high- and low-scoring matches should lead to better results. Additional features that describe the different play styles, along with more detailed data of in-game actions such as defensive actions in own third, number of different types of passes and the distances covered by these passes, distance covered by each team and etc, could provide valuable insights. Such data could help identify whether a team is playing more defensively or whether high-scoring matches, for example, have a high number of crosses. Unfortunately, this kind of data was found too late to be included in this thesis.

It is also worth investigating whether certain features are more important in the earlier stages of the season and other features important for the later stages of the season. It is not uncommon to see performances drop towards the end of the season as teams get fatigued and injuries increase. This could cause some changes in feature importance.

Additionally, the models in this thesis have been trained on five leagues combined. This approach does not account for any possible league-specific patterns that could be important for model performance within individual leagues. The goal was to create models that could generalize beyond a single league. However, training separate models on each league could yield better results on that specific league but at the cost of a significantly reduced dataset. Further analysis could be done on our models' performance on each separate league to determine whether the prediction accuracy or betting

profitability is higher in certain leagues.

Training/validation split

Another thing that could improve performance or indicate whether the training set, validation set and test set come from slightly different feature distributions, is training the models on multiple different splits. In this thesis, the same training/validation split was used in all model attempts and no cross-validation or changes in splits was done.

Alternatively, using a smaller window (fewer seasons) for the training set and using a separate season for the validation set might result in different model performance. It is worth investigating the following split: using fewer seasons in the training set (for example one, two or three) and assigning the next season to the validation set. For example using season 2017/2018 and 2018/2019 for the training set and 2019/2020 for the validation set. When the models have been trained on this configuration, iteratively apply the same split for the next seasons. This approach would return multiple model accuracies on unseen data, giving us a better understanding of whether the models are learning but also if the features are equally important in all seasons.

Model tuning and other models

Additionally, further hyperparameter tuning could be done to try and improve model performance, using for example Bayesian optimization for the XGBoost classifier or *weight decay* for the neural network in order to further combat overfitting.

Another alternative would be to explore other more complex models such as a Bayesian neural network or an LSTM model. A Bayesian network might be able to handle overfitting better and give an understanding of predictive uncertainty of the model since the model parameters $\hat{\boldsymbol{\theta}}$ are not point estimates but rather posterior distributions. An LSTM model could possibly help capture the long-term trend of goals in a match as well as the short-term fluctuations. However, if the data is still not representative of the classes, the model performance might still not be good. Interested readers can find more information about these models in Bishop (2006) and in Goodfellow et al. (2016), respectively.

Combining multiple models is also a suggestion for future improvement that could produce better results and reduce any biases the individual models might have while the variation between the models could indicate the confidence level of the predictions.

Betting strategy

The betting strategy used in this thesis is not extensive and could be changed or combined with more complex strategies. In this thesis, we only placed bets on our model's predicted outcomes, however, a different strategy where, for example, bets are placed on the outcomes that have the highest value according to the EV formula described in Section 2.9 might yield better returns, even though bets are not exclusively placed on the most probable outcome according to our models.

It is also worth investigating whether restricting the size of change in odds from the opening odds to the closing odds can improve model performance and/or profitability from the models. Significant changes in odds may reflect the change in perceived match outcome probabilities due to factors not related to team performance such as weather conditions, injuries, suspensions, etc.

It is also worth investigating whether the models would yield higher profits in other leagues. This thesis focused on the top five European leagues, which are highly popular and generate significant betting activity. As a result, there is more data available that bookmakers use when calculating their odds and therefore can offer more accurate odds. Lower-tier leagues that are less popular may offer more favourable odds that could be exploited by our models.

In summary, there is a lot of work that can still be done in order to gain a better understanding of the relationship between betting odds, match outcomes and machine learning in the context of football.

6 Appendix

Table 5: Description of the statistics for each game included in the dataset.

Variable	**Description**
Div	League Division
Date	Match Date (dd/mm/yy)
Time	Time of match kick-off
HomeTeam	Home Team
AwayTeam	Away Team
FTHG	Full Time Home Team Goals
FTAG	Full Time Away Team Goals
FTR	Full Time Result
HTHG	Half Time Home Team Goals
HTAG	Half Time Away Team Goals
HTR	Half Time Result
Referee	Match Referee
HS	Home Team Shots
AS	Away Team Shots
HST	Home Team Shots on Target
AST	Away Team Shots on Target
HC	Home Team Corners
AC	Away Team Corners
HF	Home Team Fouls Committed
AF	Away Team Fouls Committed
HY	Home Team Yellow Cards
AY	Away Team Yellow Cards
HR	Home Team Red Cards
AR	Away Team Red Cards

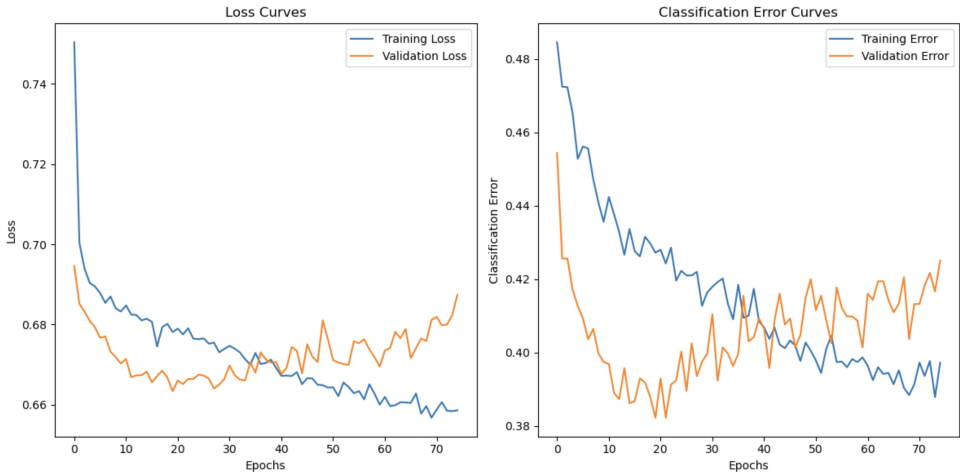

Figure 11: Loss curves as well as missclassification error curves for the training set and validation set.

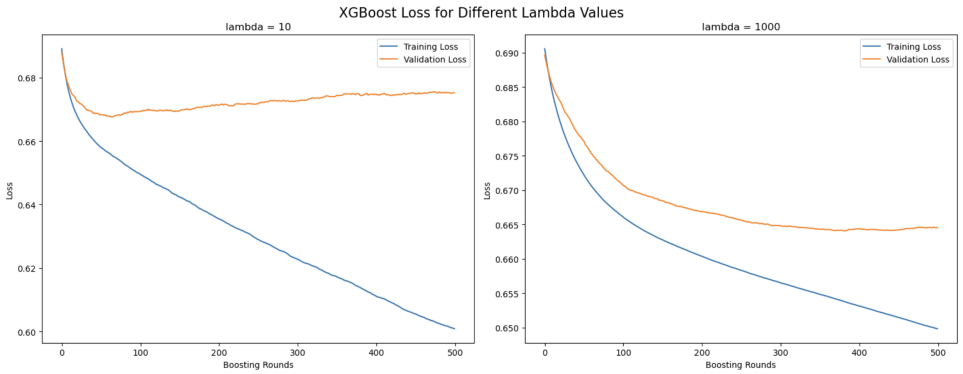

Figure 12: Loss curves on training and validation sets for two different values of the regularization parameter lambda λ for XGBoost.

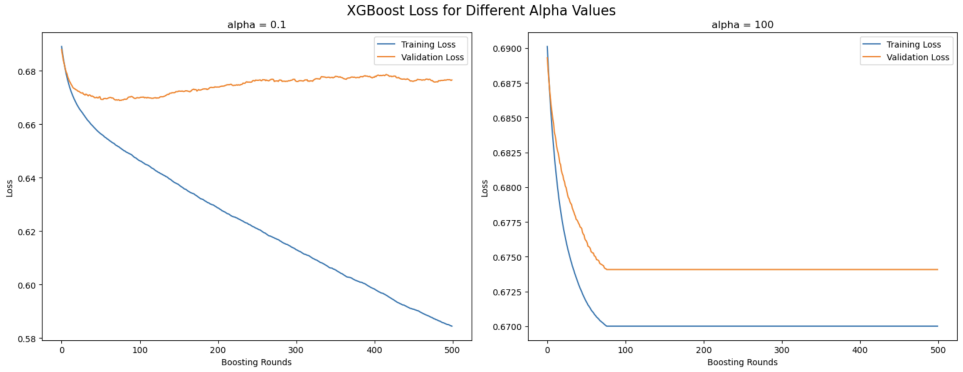

Figure 13: Loss curves on training and validation sets for two different values of the regularization parameter alpha α for XGBoost.

7 References

References

[1] Bishop, C.M. (2006). *Pattern Recognition and Machine Learning*, Springer.

[2] Chen, T. & Guestrin, C. (2016). *Xgboost: A scalable tree boosting system.* https://dl.acm.org/doi/pdf/10.1145/2939672.2939785

[3] Cortis, D. (2015). Expected values and variances in bookmaker payouts: a theoretical approach towards setting limits on odds, *The Journal of Prediction Markets.* https://www.ubplj.org/index.php/jpm/article/view/987

[4] football-data.co.uk. *Historical football results and odds data.* https://www.football-data.co.uk/data.php [Retrieved: 2025-03-17]

[5] Goodfellow, I., Bengio Y., & Courville A. (2016). *Deep Learning*, MIT Press. http://www.deeplearningbook.org

[6] Hastie, T., Tibshirani, R., & Friedman, J. (2017). *The Elements of Statistical Learning*, Second Edition, Springer.

[7] Ioffe, S. & Szegedy, C. (2015). *Batch normalization: Accelerating deep network training by reducing internal covariate shift* https://arxiv.org/abs/1502.03167

[8] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting, *Journal of Machine Learning Research, 15(56), 1929–1958.* https://www.jmlr.org/papers/v15/srivastava14a.html