MACHINE LEARNING FOR FOREIGN EXCHANGE RATE FORECASTING

by

Laurids Gert Nielsen (CID: 01424460)

Department of Mathematics
Imperial College London
London SW7 2AZ
United Kingdom

Thesis submitted as part of the requirements for the award of the MSc in Mathematics and Finance, Imperial College London, 2017-2018

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Signature and date:

Acknowledgements

I would like to thank Deutsche Bank's electronic FX Spot Trading team for providing data, high-speed computation facilities and guidance for my project. Especially Maximilian Butz, Torsten Schoeneborn, Sergey Korovin and Benedict Carter have been a great help through insightful comments and discussions.

Also, I would like to express my gratitude to my project supervisor Blanka Horvath for putting a great effort into facilitating the project with Deutsche Bank and providing guidance whenever needed.

Finally, I would like to thank my family and my girlfriend Katrine Kofoed for their invaluable support and engagement.

Abstract

This thesis investigates the applicability of machine learning methods to forecasting price movements in high frequency foreign exchange markets. We start by reviewing linear methods and then gradually increase the complexity of models considered throughout the thesis, ending with a long-short term memory neural network. We show that while machine learning methods can approximate a rich space of functions, they come with a cost of decreased interpretability and high computation time. In the last part of the thesis, all models are tested with a short prediction horizon of 2 ticks and a longer one of 50 ticks. We show that machine learning methods are capable of predicting the short horizon significantly better than the linear models but tend to struggle when the prediction horizon is longer and data more noisy.

Contents

1 Introduction

Picking up patterns in financial time series is difficult but if you are good at it, the reward is huge. This is why researchers are dragged to the task and new models are constantly being developed. In recent years, many of these new models have fallen within the category of machine learning models where an iterative algorithm is used to 'learn' complex patterns without being explicitly programmed. These models come with a high number of parameters and mathematical proofs of their capabilities are often limited. This means low interpretability and high computation time. As this is not desirable properties to have, the machine learning models will have to make significantly better predictions than the linear ones to be any relevant. The goal of this thesis is to investigate whether this is achievable by considering FX mid price movements in a high frequency setting. Models being tested will span from linear regression with 15 parameters to a neural network model with more than 5,000 parameters.

Machine learning is not a new development, and the methods which have become popular in recent years are in many cases just modifications of models invented more than 15 years ago where computational facilities for utilizing their full potential did not exist. This is also true for the models we will be testing in this thesis. In our description of models in Sections 3, we will start with the most simple models and then gradually increase the complexity. This means that we will start out by reviewing OLS and its extensions ridge (Hoerl & Kennard, 1970), lasso (Tibshirani, 1996) and elastic net (Zou & Hastie, 2005) where $\mathbf{L}^1$ and $\mathbf{L}^2$ penalties are used to regularise the linear regression. We will consider these for both predictive modelling and feature selection, as especially lasso can give a quick estimate of feature importance with least angle regression (Efron et al., 2004).

As an alternative to performing OLS directly on our input variables, we also consider the multivariate adaptive regression spline (MARS) model (Friedman, 1991) which transforms the inputs by products of hinge functions prior to performing linear regression. This results in a partition of the input space, where each region is fitted with a local polynomial capable of capturing non-linearities and low-level variable interactions.

Next, we will consider regression trees which also partitions the input space into disjoint regions but in this case, only fits a constant instead of a polynomial in each region. The regression trees can model high-level variable interaction and will be used as a basis model in the ensembles random forest (Breiman, 2001a) and stochastic gradient boosted trees (Friedman, 2002). These models were some of the first to use randomness in their learning algorithms which enables them to utilise the regression tree's ability to model high-level interactions while keeping prediction variance at a reasonable level.

Injecting randomness into the learning algorithm of machine learning methods quickly became the standard, following the success of random forest and stochastic gradient descent methods. Also, in our final model, the long-short term memory neural network (LSTM) model invented by Hochreiter & Schmidhuber (1997), we will use randomness through dropouts (Srivastava et al., 2014) to regularise the network. This model is a special type of recurrent neural network, meaning that it has a built-in sense of time which all the other models considered do not. LSTM based models have in recent years showed state of the art performance in modelling complex non-financial time series, e.g. in Graves et al. (2013). Motivated by these promising results, we will investigate if it is also applicable to financial data.

2 A Description of the Data

The dataset consists of bid and ask quotes for the four currency pairs EUR/USD, GBP/USD, EUR/CHF and EUR/SWE. Data is collected from three different exchanges which will be referred to as exchange A, B and C, as the data is confidential. All exchanges provide quotes of each currency pair with 20 ms to a few minutes between observations depending on exchange, currency pair, weekday and time of the day. Exchange A has 5-30.000 observations per day while B and C have around 8 times as many. We will be using data from all trading days throughout 2017, resulting in a large dataset. This means that we will have plenty of data to train and test our models on, but also that inconsistencies in the data can be hard to detect. Hence, a deeper understanding of the data is required before any modelling can take place. Additionally, the size of the dataset implies that computation time for any model used will be considered.

The aim of this thesis is to make predictions of mid price movements on one exchange (A) by using data from other exchanges (B and C). This means that our target for any model is the mid price increment for a given currency pair on exchange A. All predictions will be made in terms of ticks ahead in the future and not time, i.e. a 2 tick prediction can be 40 ms or a couple of minutes. The impact this has on the results will be discussed in Section 4.

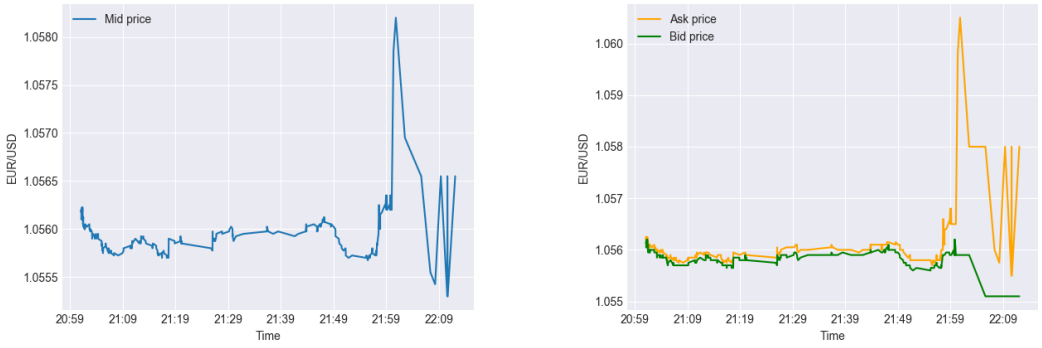
2.1 Handling a Large Dataset

Because we are predicting on exchange A, we choose to synchronise the data from exchange B and C to A. Even though B and C have a lot more observations than A, it is rarely the case that the observations share timestamps, as they are measured in ms. Luckily, it is almost always the case that B and C have observations less than 50 ms prior to the ones of A. Hence, we can construct a new dataset with only timestamps from A containing synchronised observations from all exchanges without making large modifications of the data. In the case that no observations from either B or C are observed between two consecutive observations from A, the values from A will also be filled in as a replacement for B or C respectively. This is to avoid using "too old" values in our models without having to construct models that handle missing values or run multiple models simultaneously for different cases of missing values from B and C. Since B and C updates are much more frequent, it is rarely needed to fill in values from A to B or C (less than 0,1% of observations). It is though important to treat the case of missing updates and not just do a forward fill, as it does happen that exchange B or C shuts off for a couple of minutes. By forward filling in these cases our models would detect a large discrepancy between exchange prices which could result in poor results, as these discrepancies will prove to be the main drivers of our models in Section 4.

2.2 Illiquid Markets

Before we start creating models there is an important factor which we have to consider: Liquidity of markets. We say that a market is illiquid if one cannot execute trades of a reasonable size without moving the market significantly. In this situation, one will frequently observe that the mid price suddenly moves a lot, but shortly afterwards returns to its previous value. By taking a closer look, it will usually become evident that the move in mid price was caused by a move in only the bid price or only the ask price, i.e. it was due to just one trade that took out a significant part of one side of the order book. This situation is depicted in Figure 1 below where we in Figure 1a observe sizeable moves in the mid price but in Figure 1b we see that these are mainly caused by high fluctuations in the ask price.

These large movements in mid price can be seen as unreal movements, and in periods of time



(a) Mid price

(b) Bid and ask prices

Figure 1: Friday evening prices

where we observe a high amount of these, we will avoid making predictions. Since the sizes of these movements are relatively large compared to what is observed during more liquid trading hours, the movements would play a significant role in training and evaluation of prediction models. Generally, financial markets are difficult to predict but most models will have an easy time detecting the pattern that mid price goes down if it just went up a lot and vice versa. Because the movements are so predictable, most models will actually seem to make a worse fit to the data once illiquid trading hours are removed from the data.

As the dataset only contains bid and ask quotes and not quantities in the order book, we will be taking a qualitative approach based on price behaviour to determine when the liquidity is sufficient for making predictions. All results will be accompanied with a note on exactly what data was used for the models to ensure reproducibility of the results.

3 Models

In this section, we provide a description of models used to forecast mid price movements. Assuming that our models take p variables as input, our input space is a subset of $\mathbb{R}^p$ and the space in which our target variable (increment 2-50 ticks ahead) is a subset of $\mathbb{R}$. In this setting, an observation $(\mathbf{x}_t, y_t)$ in our data is a realisation of a stochastic vector (X_t, Y_t) taking values in $\mathbb{R}^{p+1}$. When constructing a predictive model, we seek to find a function $f_{\mathcal{L}}$ which minimises the expected prediction error/generalisation error

$$EPE_t[f_{\mathcal{L}}] = \mathbb{E}[L\big(Y_t, f_{\mathcal{L}}(X_t)\big)|X_t = \mathbf{x}_t], \tag{3.1}$$

where L is a loss function and $\mathcal{L}$ is a set of realised values $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ of stochastic vectors $(X_1, Y_1), \ldots, (X_N, Y_N)$ which we call our learning set. The learning set will usually be a subset of the whole dataset we have available and is used to construct the prediction function $f_{\mathcal{L}}$.

One of the main difficulties of predictive modelling is figuring out how to utilise our data in the best possible way. If we choose to use all of our dataset to construct our models (put all of it into the learning set), we would not have any additional data for testing them. Hence, we would have to also use the learning set (or at least a part of it) for evaluating the models. As we increase the number of parameters in a model, it will be able to fit the learning set better. For instance, if we have T observations, we can construct a polynomial of order T which would fit the learning set perfectly, i.e.

$$f_{\mathcal{L}}(\mathbf{x}_t) = y_t, \quad t = 1, \ldots, T.$$

When evaluating the model on $\mathcal{L}$ it would seem like we have produced a very good model, but if we tested it on unseen data, it would very likely perform poorly. This phenomena is called overfitting and happens because we do not take into account that minimizing (3.1) means to at a given point of time, t, minimise over all possible outcomes of (X_t, Y_t) and not just the values observed in the learning set or the remainder of the dataset. In other words, we want our model to generalise to unseen data.

To avoid drawing false conclusions, we will set aside a part of the dataset for testing our models which we will denote by $\mathcal{T}$. To make a fair comparison between models, this set cannot be used to construct or adjust models in any way. When dealing with time series, we must choose $\mathcal{L}$ and $\mathcal{T}$ such that they consist of consecutive observations and observations from $\mathcal{L}$ must be observed prior the ones of $\mathcal{T}$. If we did not separate the data in this way, we would be using the future to predict the past which would yield an unrealistic test result.

While the rules on how we are allowed to test our models are fairly strict, we are free to use the learning set in any way we find suitable. To avoid overfitting, we will try to choose models which are likely to lead to good predictions on unseen data (out of sample), i.e. minimise (3.1). The typical approach is to first choose a model type with a specific learning algorithm $\mathcal{A}$ which given complexity parameters θ_c and a learning set produces a prediction model. The parameters, we can choose in θ_c, depends on the model type, and we will provide a description of what parameters are available for each model discussed in this section. Assuming that the algorithm is deterministic, this means that we can see a model as an output of the function

$$\mathcal{A}(\theta_c, \mathcal{L}) = f_{\mathcal{L}}. \tag{3.2}$$

To avoid too many subscripts, we will only use the subscript $\mathcal{L}$ and not θ_c for $f_{\mathcal{L}}$ which should not give rise to much confusion. If we consider a specific model type, (3.2) means that choosing the prediction function $f_{\mathcal{L}}$ in (3.1) is reduced to finding an optimal value of θ_c. A data efficient way of doing this is cross-validation. In this technique, a model is fitted on part of the learning set and tested on another part disjoint from the first one. The parts we train and test on are then rotated or shifted and we evaluate the loss on the testing set at each iteration. Taking the average of test errors gives us a cross-validation score. We may repeat this procedure for multiple values of θ_c and then select the one that yields the minimum cross-validation score. As we are dealing with time series data, the training set in each iteration should consist of observations occurring prior to the corresponding test set.

Throughout this thesis, the goal of all models will be to maximise

$$R^2 := 1 - \frac{\sum_{t=N+1}^{T}(y_t - \hat{y}_t)^2}{\sum_{t=N+1}^{T}(y_t - \bar{y})^2},$$

where $\hat{y}_t$ is the prediction of y_t and

$$\bar{y} = \frac{1}{T-N}\sum_{t=N+1}^{T} y_t.$$

This is clearly equivalent to choosing $L(y_t, \hat{y}_t) = \frac{1}{2}(y_t - \hat{y}_t)^2$ in (3.1) which is known as the mean squared error loss function (MSE).

Assume that there exist a function $f_{\mathcal{B}}$ such that

$$Y_t = f_{\mathcal{B}}(X_t) + \epsilon_t, \quad t = 1, 2, \ldots,$$

where $(\epsilon_t)_{t\in\mathbb{N}}$ is a stochastic process independent of $(X_t)_{t\in\mathbb{N}}$ with constant mean 0 and variance σ_t^2. Then $f_{\mathcal{B}}$ is the best possible prediction, i.e. it achieves the lowest possible MSE. This function

is often called the Bayes prediction function which explains the subscript $\mathcal{B}$. In this setting, we can decompose (3.1) in the following way:

$$\begin{aligned} EPE_t[f_{\mathcal{L}}] &= \mathbb{E}[\big(Y_t - f_{\mathcal{L}}(X_t)\big)^2 | X_t = \mathbf{x}_t] \\ &= \mathbb{E}[\big(f_{\mathcal{B}}(X_t) + \epsilon_t - f_{\mathcal{L}}(X_t)\big)^2 | X_t = \mathbf{x}_t] \\ &= \mathbb{E}[\big(f_{\mathcal{B}}(\mathbf{x}_t) - f_{\mathcal{L}}(\mathbf{x}_t)\big)^2] + \sigma_t^2, \end{aligned}$$

using that cross terms with ϵ_t are 0, as ϵ_t is independent of X_t and has mean 0. This expression may seem deterministic at time t, but it still contains randomness, since $f_{\mathcal{L}}$ is based on just one realisation of the path of X_t up to time N and we do not know the distribution of the process. Hence, we can further decompose by taking expectation over possible paths up to time N:

$$\begin{aligned} EPE_t[f_{\mathcal{L}}] =& \mathbb{E}[\big(f_{\mathcal{B}}(\mathbf{x}_t) - \mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)] + \mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)] - f_{\mathcal{L}}(\mathbf{x}_t)\big)^2] + \sigma_t^2 \\ =& \mathbb{E}[\big(f_{\mathcal{B}}(\mathbf{x}_t) - \mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)]\big)^2] + \mathbb{E}[\big(\mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)] - f_{\mathcal{L}}(\mathbf{x}_t)\big)^2] \\ &+ 2\mathbb{E}[\big(f_{\mathcal{B}}(\mathbf{x}_t) - \mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)]\big)\big(\mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)] - f_{\mathcal{L}}(\mathbf{x}_t)\big)] + \sigma_t^2 \\ =& \big(f_{\mathcal{B}}(\mathbf{x}_t) - \mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)]\big)^2 + \mathbb{E}[\big(\mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)] - f_{\mathcal{L}}(\mathbf{x}_t)\big)^2] \\ &+ 2\big(f_{\mathcal{B}}(\mathbf{x}_t) - \mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)]\big)\mathbb{E}[\mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)] - f_{\mathcal{L}}(\mathbf{x}_t)] + \sigma_t^2 \\ =& \big(f_{\mathcal{B}}(\mathbf{x}_t) - \mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)]\big)^2 + \mathbb{E}[\big(\mathbb{E}[f_{\mathcal{L}}(\mathbf{x}_t)] - f_{\mathcal{L}}(\mathbf{x}_t)\big)^2] + \sigma_t^2 \\ =& \text{bias}[f_{\mathcal{L}}(\mathbf{x}_t)]^2 + \text{Var}[f_{\mathcal{L}}(\mathbf{x}_t)] + \sigma_t^2. \end{aligned} \tag{3.3}$$

The above is decomposition is called the bias-variance decomposition. When constructing a model, it is usually the case that increasing complexity will decrease bias but increase variance. Hence, optimal model complexity can be seen as the optimal trade-off between bias and variance.

The problem when considering real data is that bias, variance and σ_t^2 are not directly observable. We just observe an error term, but we do not know what caused it. Also, we cannot say for sure, if the error could be made smaller with another prediction function or we are close to the optimal prediction function, as we do not know $f_{\mathcal{B}}$. Our approach in this thesis is to choose a model type/learning algorithm and then select complexity parameters θ_c through cross-validation as described above. Learning algorithms differ in the way they construct prediction functions, but they all seek to minimise an objective function of the form

$$obj(f_{\mathcal{L}}) = L(f_{\mathcal{L}}) + \Omega(f_{\mathcal{L}}), \tag{3.4}$$

where

$$L(f_{\mathcal{L}}) = \sum_{t=1}^{N} L(\mathbf{x}_t, y_t)$$

and Ω is a term which penalises the complexity of models. The idea of Ω is to avoid overly complex models with too high variance on predictions. Some times, we will also write (3.4) as

$$obj(\theta_{\mathcal{L}}) = L(\theta_{\mathcal{L}}) + \Omega(\theta_{\mathcal{L}}), \tag{3.5}$$

where $\theta_{\mathcal{L}}$ contains the model parameters chosen by the learning algorithm (e.g. coefficients in a linear regression). As it was the case for $f_{\mathcal{L}}$, the optimal values of $\theta_{\mathcal{L}}$ clearly depend on θ_c, but we will not use subscripts to denote this dependence.

As mentioned above, we will focus on the MSE loss function throughout this thesis. Because it squares the error terms, it will put great emphasis on extreme values. In some cases, this may result in the learning function $f_{\mathcal{L}}$ putting too much emphasis on rare events that are unlikely to generalise to unseen data. When this is true, it can be useful to choose L in (3.5) to be different from the L in (3.1). For instance, we may choose L in training to be the Huber loss function which has linear tails and hence put relatively less emphasis on large errors:

$$H_\delta(y_t, \hat{y}_t) = \begin{cases} \frac{1}{2}(y_t - \hat{y}_t)^2, & |y_t - \hat{y}_t| \leq \delta \\ \delta|y_t - \hat{y}_t| - \frac{1}{2}\delta^2, & |y_t - \hat{y}_t| > \delta. \end{cases}$$

In this loss function, δ determines the threshold for when the loss function becomes linear instead of quadratic. The Huber loss function is pictured in Figure 2 below for $\delta = 1$ and $\delta = 3$. The loss function

$$L(y_t, \hat{y}_t) = \log(\cosh(y_t - \hat{y}_t))$$

is also shown in the plot, as we will use it as an approximation of the Huber loss function with $\delta = 1$ when considering neural networks in Section 3.7. It is clear from the plot that this approximation is fairly good. As we will only rarely change loss function, we will assume that the loss functions in (3.1) and (3.5) are the same unless otherwise stated.

In the following sections, we will start by briefly summarising linear methods and then move on to examine more complex machine learning techniques in greater detail. We will denote the set of input variables which we run the learning algorithm on by X and the corresponding target variables by Y. (X, Y) is a subset of $\mathcal{L}$ consisting of $N \leq T$ samples, but we will not put much emphasis on whether it is the full learning set or just a part of it (e.g. during a cross-validation), as this should be clear from the context.

All models are made in Python and the libraries used will be mentioned.

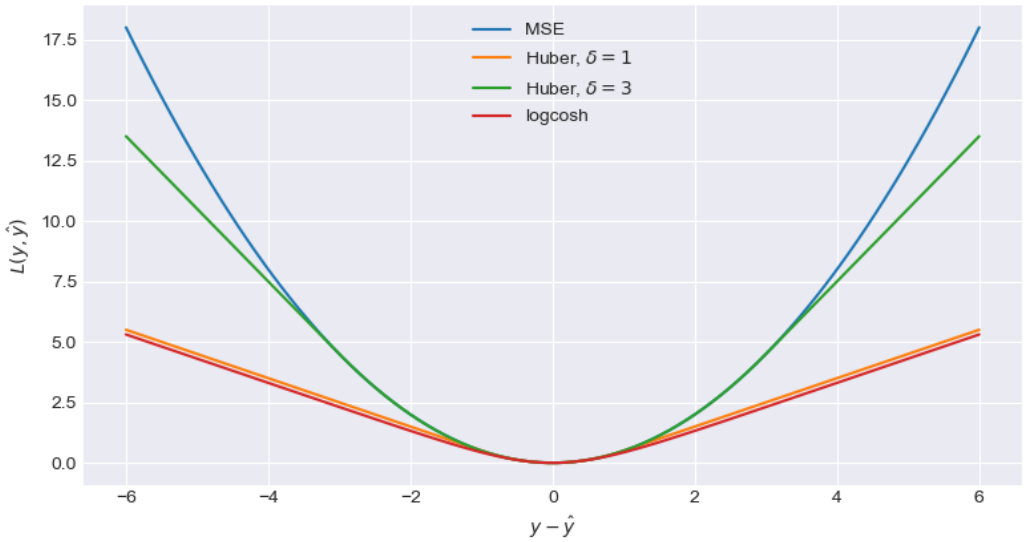


Figure 2: Comparison of loss functions

3.1 Ordinary Least Squares

Ordinary least squares (OLS) is a simple form of regression with no complexity parameters to tune before fitting but it is still fundamental for regression analysis. Its objective function has no regularisation term and hence only seeks to minimise its loss function:

$$obj(\theta_{\mathcal{L}}) = L(\theta_{\mathcal{L}}) = \left\| X\theta_{\mathcal{L}} - Y \right\|_2^2, \tag{3.6}$$

where $\left\| \cdot \right\|_2^2$ is the squared $\mathbf{L}^2$-norm. One of the main reasons why OLS is widely popular is the high interpretability of parameters. A positive (negative) coefficient for a variable means that the target variable correlates positively (negatively) with the variable. Another advantage of OLS is its low computation time. As it has no complexity parameters to tune by cross-validation, one can fit the model to the data right away which itself is fast. On the negative side, OLS does not take non-linear dependencies or interactions between variables into account. This problem can partly be mitigated by constructing new features through transformations of existing ones and then perform OLS on these. MARS, which is described in Section 3.3, is an example of this. Another disadvantage of OLS is that the variance of its predictions is very high when columns in our data are collinear. This happens because entries of $(X^T X)^{-1}$ becomes very large and the solution of OLS is given by

$$\hat{\theta}_{\mathcal{L}} = (X^T X)^{-1} X^T Y,$$

meaning that a small change in the data could result in a large change in predictions. To deal with this problem we add a regularisation term to the objective function. The effect of this is explained

in Section 3.2.

Predictions are made with Python's OLS implementation
`sklearn.linear_models.LinearRegression`.

3.2 Shrinkage Methods

In this section, we discuss shrinkage methods for linear regression which by penalising the size of regression coefficients treats collinearity of columns and prevents overfitting. One way to perform regularisation is ridge regression (Hoerl & Kennard, 1970) which adds a regularisation term

$$\Omega(\theta_{\mathcal{L}}) = \lambda \left\| \theta_{\mathcal{L}} \right\|_2^2$$

to the objective function (3.6) of OLS. Increasing λ will increase the bias of our predictions but decrease the variance. Selecting λ is a matter of selecting the right bias/variance trade-off which can be done by cross-validation. The modified objective function is still differentiable, and hence we can differentiate to get

$$\hat{\theta}_{\mathcal{L}} = (X^T X + \lambda I_p)^{-1} X^T Y,$$

where p is the number of variables used for the model and I_p is the p-dimensional identity matrix. As λ increases, coefficients will shrink towards 0 in a relatively smooth manner, but they rarely become exactly 0, i.e. ridge regression does not perform any feature selection.

If we want our model to perform feature selection, we should instead consider lasso regression (Tibshirani, 1996) where the regularisation term added to (3.6) is given by

$$\Omega(\theta) = \alpha \left\| \theta_{\mathcal{L}} \right\|_1$$

The objective function is no longer differentiable, but fast algorithms for solving the regression problems exist (Hastie, 2008, chap. 3.4.4). It is even possible to produce the whole lasso path at once with least angle regression (Efron et al., 2004), i.e. calculate the coefficients for a range of αs in one go. This makes cross-validation computationally cheap and can be a reason to choose lasso over ridge regression when considering large datasets. In Python, this fast lasso cross-validation can be done with `sklearn.linear_models.LassoCV`.

One disadvantage of lasso is that the regularisation is often not as smooth as it is the case for ridge regression, meaning that small changes in α can yield large differences in coefficients. This is clearly not desirable in terms of interpretability and stability of the model. We would like that if a variable is assigned a large coefficient (in absolute value) it should mean that it is important

for the regression, and that should not change due to small changes of input.

Finally, the last regularisation method we will consider is a combination of ridge and lasso called elastic net (Zou & Hastie, 2005). It combines the regularisation terms from ridge and lasso to make a regularisation that can be both relatively smooth and include feature selection. Its regularisation term is given by

$$\Omega(\theta_{\mathcal{L}}) = \alpha\left\|\theta_{\mathcal{L}}\right\|_1 + \lambda\left\|\theta_{\mathcal{L}}\right\|_2^2.$$

For one value of λ, the path of multiple αs can be computed cheaply with `sklearn.linear_model.ElasticNetCV`. This means that using elastic net is not much more expensive than ridge.

Besides being useful to prevent overfitting, shrinkage methods are also useful for data exploration ahead of performing other regression techniques. As we increase the penalty parameters, it becomes expensive for the models' objective functions to assign large coefficients (in absolute value) to variables that are not particularly important for the regression. Hence, the shrinkage methods will only assign sizeable coefficients to variables which they consider to be important. Especially lasso is good for data exploration, as it can produce the whole path of coefficients for multiple values of α quickly and is capable of performing feature selection. In Figure 3 below, the Lasso path for EUR/USD 2 ticks predictions is shown. We see that some variables get zeroed very quickly, indicating that they are non-important for the regression, while particularly the variable $\text{Mid}_\text{A} - \text{Mid}_\text{B}$ retains a high value even for high values of α. One shortfall of data exploration with linear shrinkage methods is that they will assign no importance to non-directional variables. For instance, variables describing spreads or volatilities are likely to be zeroed quickly even though they may carry important information about future price movements. We will see how to use another feature importance measure in Section 3.5 which is capable of assigning importance to non-directional variables.

Before applying any of the shrinkage methods above, data should be standardised, as the scale of variables otherwise will play an important role in determining the regression coefficients. Additionally, it is desirable that a shift of the target variable will not affect any coefficients other than the intercept, i.e. we do not wish to penalise the intercept coefficient. By standardisation, the intercept becomes 0 and this problem is avoided.

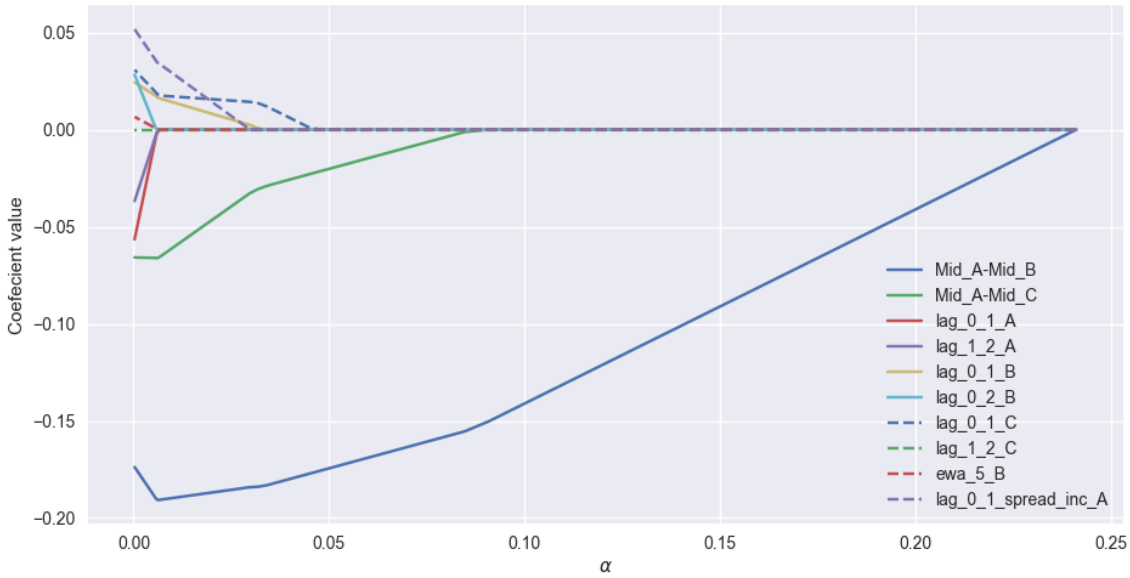


Figure 3: Lasso path for EUR/USD 2 ticks predictions

3.3 MARS

An alternative to performing OLS directly is to apply a set of basis functions to our data before performing regression. This means that we will model our data with a function of the form

$$f(X_t) = \beta_0 + \sum_{m=1}^{M} \beta_m g_m(X_t).$$

Multivariate adaptive regression splines (MARS) introduced by Friedman (1991) is a structured approach to selecting these basis functions. Consider the set

$$\mathcal{B} = \{(\mathbf{x}_j - u)^+, (u - \mathbf{x}_j)^+ \quad | \quad j = 1, \ldots, p, \quad u = x_{1,j}, \ldots, x_{N,j}\}$$

consisting of $2np$ different hinge functions. Here p is the number of variables in our data, $\mathbf{x}_j$ is the j'th row of our feature matrix X and $x_{t,j}$ is the $(t,j)_{th}$ entry of X. In MARS, any basis function g_m will be a function from $\mathcal{B}$ or a product of functions from $\mathcal{B}$. If we do not consider products of functions from $\mathcal{B}$, we will be making piecewise linear regression (piecewise polynomial of order 1). If we restrict our basis functions to be a product of maximum two elements from $\mathcal{B}$ we get a piecewise polynomial of order 2 etc.

Even though we have restricted our basis functions to be of a specific class, the space of basis functions is still way too large for attempting all possible combinations. Denoting the set of selected basis functions $\mathcal{M}$, the MARS algorithm is a greedy algorithm which selects functions for $\mathcal{M}$ as described below.

Algorithm 3.1.

1. *At first* $\mathcal{M}$ *only consists of the constant 1, i.e. an intercept term*
2. *For each pair of hinge functions,* $\big((X_j - u)^+, (u - X_j)^+\big)$*, from* $\mathcal{B}$*, we calculate the maximum reduction in* MSE *that can be attained by multiplying these hinge functions with a function from* $\mathcal{M}$ *and adding these new terms to* $\mathcal{M}$*.*
3. *The pair which causes the largest reduction in* MSE *is added to* $\mathcal{M}$
4. *2 and 3 are repeated until a chosen maximum number of terms in* $\mathcal{M}$ *has been reached or the improvement of adding more terms is less than some threshold*
5. *Another greedy algorithm removes one term from* $\mathcal{M}$ *at a time depending on which one would cause the smallest decrease in* MSE *by being removed. For each iteration the generalised cross-validation score*
$$GCV(\lambda) = \frac{\sum_{t=1}^{N}(f(\mathbf{x}_t) - y_t)^2}{(1 - M(\lambda)/N)^2}$$
is calculated, where $M(\lambda) = |\mathcal{M}| + \lambda(|\mathcal{M}| - 1)$
6. *The iteration in 5 with the lowest GCV score is selected as the final model*

Note that GCV penalises the complexity of the model, since $M(\lambda)$ increases with the number of hinge functions in $\mathcal{M}$. Hence, step 5 and 6 are similar to adding a regularisation term Ω to (3.6). Suitable λ can be chosen by cross-validation.

MARS is an additive model which is able to model low-level interactions between variables. This could prove to be useful as it enables us to incorporate non-directional variables in our model. An example of such a variable is the spread at a given exchange; it is unlikely that the size of the spread alone can give any information of which direction the market will move, but maybe it could be indicative of what the size of future movements is going to be. Because the polynomials constructed by MARS are local, we are also able to capture interactions that are only relevant for certain values of the variables.

Regarding understanding of MARS, the most difficult part lies in step 2 of the Algorithm 3.1. The naive approach to implementing MARS would be to try adding each possible pair of hinge functions to M and perform OLS at each iteration. Clearly, this would be very expensive computationally, as we have Np different pairs of hinge functions. Instead, we will rely on the similarities between each fit to construct a search function which is much cheaper. The following derivation is adopted from Zhang (1994).

Assume that we have K basis functions in M. Applying these to our feature matrix, will give

vectors $\mathbf{b}_0, \ldots, \mathbf{b}_{K-1}$, where $\mathbf{b}_0$ is just a vector of ones. For convenience, we will in the following use typical vector notation

$$\mathbf{1} = (1, \ldots, 1)^T,$$
$$(\mathbf{x}_j - u_m\mathbf{1}) = ((x_{1,j} - u_m)^+, \ldots, (x_{N,j} - u_m)^+)$$

etc. The basis functions that we are adding to $\mathcal{M}$ will be of the form

$$\mathbf{b}_m \circ (\mathbf{x}_j - u\mathbf{1})^+, \ \mathbf{b}_m \circ (u\mathbf{1} - \mathbf{x}_j)^+,$$

where $\circ$ is the entry-wise/Hadamard product. We note that the second term can be written as

$$\mathbf{b}_m \circ (\mathbf{x}_j - u\mathbf{1})^+ - \mathbf{b}_m \circ \mathbf{x}_j + u\mathbf{b}_m.$$

As $\mathbf{b}_m$ is already in the basis, this means that adding the two hinge functions to $\mathcal{M}$ is the same as adding $\mathbf{b}_m \circ (\mathbf{x}_j - u\mathbf{1})^+$ and $\mathbf{b}_m \circ \mathbf{x}_j$, i.e. it results in the same span of vectors. If $\mathbf{b}_m \circ \mathbf{x}_j$ is already in the basis, then we only add the hinge function. Given that this is not the case, we will define

$$Z := (\mathbf{b_0}, \ldots, \mathbf{b}_{K-1}, \mathbf{b}_m \circ \mathbf{x}_j),$$

which is the basis matrix containing the first K basis vectors and $\mathbf{b}_m \circ \mathbf{x}_j$. If $\mathbf{b}_m \circ \mathbf{x}_j$ is already in the basis, this term is left out of Z. Additionally, we will define

$$\mathbf{b}_{K+1}(u) := \mathbf{b}_m \circ (\mathbf{x}_j - u\mathbf{1})^+.$$

Performing OLS with Z as matrix of input variables and a vector $\mathbf{Y}$ as target yields the residuals

$$\mathbf{r} = (I - P)\mathbf{Y},$$

where $P = Z(Z^TZ)^-Z^T$ is the orthogonal projection onto the columns of Z. When adding $\mathbf{b}_{K+1}(u)$ to the matrix, the least squares optimisation problem becomes

$$\begin{aligned}
&\min_{\beta,u} \left\|\mathbf{r} - \beta(I - P)\mathbf{b}_{K+1}(u)\right\|_2^2 \\
=&\min_{\beta,u} \left(\mathbf{r} - \beta(I - P)\mathbf{b}_{K+1}(u)\right)^T \left(\mathbf{r} - \beta(I - P)\mathbf{b}_{K+1}(u)\right) \\
=&\min_{\beta,u} \left\|\mathbf{r}\right\|_2^2 - 2\beta\mathbf{r}^T(I - P)\mathbf{b}_{K+1}(u) + \beta^2\mathbf{b}_{K+1}^T(u)(I - P)\mathbf{b}_{K+1}(u) \qquad (3.7)
\end{aligned}$$

Noting that (3.7) is convex in β, we can differentiate it w.r.t to β, set it to 0 and obtain

$$\beta = \frac{\mathbf{r}^T(I - P)\mathbf{b}_{K+1}(u)}{\mathbf{b}_{K+1}^T(u)(I - P)\mathbf{b}_{K+1}(u)}$$

for any given u. Inserting this into (3.7) gives us a new optimisation problem in just one variable:

$$\min_u \left\|r\right\|_2^2 - \frac{\left(\mathbf{r}^T(I - P)\mathbf{b}_{K+1}(u)\right)^2}{\mathbf{b}_{K+1}^T(u)(I - P)\mathbf{b}_{K+1}(u)}. \qquad (3.8)$$

The first term of (3.8) does clearly not depend on u. Hence, it is enough to optimise the second term which we will denote $h(u)$. As $h(u)$ is a non-differentiable transformation, we will have to iterate through all possible values of u. Without loss of generality, we may assume that $x_{1,j}, \ldots, x_{N_j}$ are sorted in increasing order for $j \in \{1, \ldots, p\}$. For $u = x_{t,j}$, we then have that

$$\mathbf{b}_{K+1}(u) = (\mathbf{b}_m \circ \mathbf{x}_j)_{(-t)} - t\mathbf{b}_{m(-t)}, \tag{3.9}$$

where $\mathbf{v}_{(-t)} = (0, \ldots, 0, v_{t+1}, \ldots, v_N)$.

As P is a symmetric matrix with rank $K+1$ (number of independent columns in Z), we can write it as $\tilde{P}\tilde{P}^T$, where $\tilde{P}$ is a $N \times (K+1)$ matrix with full rank. By using this decomposition, we can insert (3.9) in $h(u)$ to get

$$h(u) = \frac{(c_{1,t} - uc_{2,t})^2}{c_{3,t} - 2uc_{4,t} + u^2 c_{5,t}},$$

where

$$\begin{aligned}
c_{1,t} &= \mathbf{r}^T(I - \tilde{P}\tilde{P}^T)(\mathbf{b}_m \circ \mathbf{x}_j)_{(-t)}, \\
c_{2,t} &= \mathbf{r}^T(I - \tilde{P}\tilde{P}^T)\mathbf{b}_{m(-t)}, \\
c_{3,t} &= \left\|(\mathbf{b}_m \circ \mathbf{x}_j)_{(-t)}\right\|_2^2 - \left\|\tilde{P}^T(\mathbf{b}_m \circ \mathbf{x}_j)_{(-t)}\right\|_2^2, \\
c_{4,t} &= \mathbf{b}_{m(-t)}^T(\mathbf{b}_m \circ \mathbf{x}_j)_{(-t)} - (\mathbf{b}_m \circ \mathbf{x}_j)_{(-t)}^T\tilde{P}\tilde{P}^T\mathbf{b}_{m(-t)}, \\
c_{5,t} &= \left\|\mathbf{b}_{m(-t)}\right\|_2^2 - \left\|\tilde{P}^T\mathbf{b}_{m(-t)}\right\|_2^2.
\end{aligned}$$

It is clear, that once all the c's have been calculated, it is easy to calculate $h(u)$ which is what we need to compare all the splits. The important takeaway from all the above calculations is that going from $c_{l,t}$ to $c_{l,t+1}$ is a fixed number of steps independent of t for $l = 1, \ldots, 5$. For instance, if we denote $\mathbf{b}_m$=$(b_{1,m}, \ldots, b_{N,m})$, updating $c_{1,t}$ to $c_{1,t+1}$ is done by the calculation

$$c_{1,t+1} = c_{1,t} - \left(\mathbf{r}^T(I - P)\right)_{t+1} b_{t+1,m} x_{t+1,m},$$

where $\left(\mathbf{r}^T(I - P)\right)$ is a vector that can be precomputed and used for all iterations, as it does not depend on t. Updating $c_{2,t}$ is done similarly. This means that updating these two c's does not depend on N or K. The cost of updating the last 3 c's does also not depend on N, but it scales linearly with K. To update these, we need to create two temporary $(K+1)$-sized vectors

$$\begin{aligned}
w_{1,t} &= \tilde{P}^T(\mathbf{b}_m \circ \mathbf{x}_j)_{(-t)}, \\
w_{2,t} &= \tilde{P}^T\mathbf{b}_{m(-t)},
\end{aligned}$$

which can be updated by calculating

$$w_{1,t+1} = w_{1,t} - b_{t,m} x_{t,j} \tilde{\mathbf{p}}_t,$$

$$w_{2,t+1} = w_{2,t} - b_{t,m}\tilde{\mathbf{p}}_t,$$

where $\tilde{\mathbf{p}}_t$ is the t'th row of $\tilde{P}$. After calculating these, we can update the last 3 c's in a fixed number of steps similar to the first two.

All these recursive calculations are important when considering what sort of problems the MARS algorithm is appropriate for. The above shows that with K terms in the basis, the cost of calculating MSE reduction for splitting at each possible point for a given variable paired with a given basis term is of the order KN. To select a pair of hinge functions for the basis, we will have to perform at most Kp of these searches, meaning that adding a pair of hinge functions to the basis has cost of order NpK^2.

Adding terms to a linear regression will always increase the in sample R^2 but since the algorithm has to terminate at some point, we will often restrict the algorithm to end up with a fixed number of terms K^* in the forward pass which requires at lest $\frac{K^*-1}{2}$ iterations. This means that running the MARS algorithm with K^* maximum terms has a cost of the order NpK^{*3}. In order to keep computation time in a feasible region, we must hence restrict the model to only look at a low number of terms and unimportant features should be removed before initializing the algorithm. On the other hand, linearity in N means that the algorithm is useful for large datasets like the one we are considering.

3.4 Regression Trees

The MARS algorithm was a way of splitting the features space into subsets and then create an additive model with low-level interactions. If we would like to keep the subset split but allow for high-level interactions between variables, regression trees are worth considering. Like the MARS algorithm, a regression tree splits the feature space into regions $r_1, \dots, r_M$. The response variable is then modelled by a function of the form

$$f(X_t) = \sum_{m=1}^{M} c_m \mathbb{1}_{r_m}(X_t).$$

Due to this simple form of f, we can without loss of generality assume that $r_1, \dots, r_M$ are disjoint sets. This means that if our loss function is MSE and we apply no regularisation, then, according to (3.1), we have that

$$c_m = \min_c \mathbb{E}[(Y_t - c)^2 | X_t \in r_m] = \mathbb{E}[Y_t | X_t \in r_m].$$

As we generally do not know the distribution of $Y_t|X_t$, we will use the empirical estimate

$$\hat{c}_m = \min_c \frac{1}{|r_m|} \sum_{t=1}^{N} (y_t - c)^2 \mathbb{1}_{r_m}(\mathbf{x}_t) = \frac{1}{|r_m|} \sum_{t=1}^{N} y_t \mathbb{1}_{r_m}(\mathbf{x}_t),$$

where $|r_m|$ is number of observations in region r_m. For a large dataset with multiple features, it is computationally infeasible to find the optimal regions r_m (Louppe, 2014, page 30). We will instead use a greedy algorithm which makes recursive binary splits of our feature space. We denote the disjoint subset decomposition of our feature space by R and will perform splits of sets r in R of the form

$$r_1(j,s)\{\mathbf{x} \in r | x_j \leq s\}, \quad r_2(j,s) = \{\mathbf{x} \in r | x_j > s\}, \quad j = 1, \ldots, p, \ s \in \mathbb{R},$$

where x_j is the j'th coordinate of the p-dimensional observation $\mathbf{x}$. In order to avoid overfitting and save computation time, we will stop splitting a region r if one of the following stopping criteria is met:

a) $|r|$ is less than two times a minimum number of elements in terminal regions (leaves).

b) The number of splits made to each region r has reached some threshold (maximum tree depth has been achieved).

c) The decrease in MSE achieved by the optimal split is below some threshold (leaf impurity decrease is too low).

Note that using criteria c) is similar to adding a regularisation term

$$\Omega(\theta_{\mathcal{L}}) = \gamma |R|, \quad \gamma > 0$$

to the learning algorithm's objective function which without any minimum leaf impurity decrease only will seek to minimise the loss function at each split. When using criteria c), the learning algorithm will only make a split if the split decreases MSE of the whole model by at least γ.

Algorithm 3.2 below outlines how a regression tree is constructed.

Algorithm 3.2.

1. *R is initialised as the whole feature space*

2. *For every r in R where no stopping criteria have been met, we calculate*

$$\min_{j,s} \left[\min_{c_1} \sum_{t=1}^{N} \mathbb{1}_{r_1(j,s)}(\mathbf{x}_t) L(y_t, c_1) + \min_{c_2} \sum_{i=1}^{N} \mathbb{1}_{r_2(j,s)}(\mathbf{x}_t) L(y_t, c_2) \right]$$

3. *If stopping criteria c) is not met, then r is replaced by $r_1(j^*, s^*)$ and $r_2(j^*, s^*)$, where (j^*, s^*) is the solution found in 2*

4. *Repeat 2 and 3 until all regions in R have met one of the stopping criteria*

After termination of the algorithm, we have partitioned the feature space $\mathbb{R}^p$ into $|R|$ p-dimensional rectangles which each have assigned a constant $\hat{c}_m$ (the average of the target variables in the region if the loss function is MSE). This means that our predictive model $f_{\mathcal{L}}$ will be a multidimensional step function. If we choose restrictive stopping criteria, it will result in a very simple model which probably will suffer from a high bias. On the other hand, if we grow the tree with almost no stopping criteria, we will end up with a model with high variance. In the extreme case with no stopping criteria, all leaves will be of size 1. This means that $f_{\mathcal{L}}$ will return the target value of the observation from $\mathcal{L}$ 'closest' to the new observation.

In Figure 4 the top of a decision tree graph made for EUR/CHF 2 ticks prediction with a minimum leaf size of 25 is shown. Input and target variables have been standardised, as this makes the values easier to interpret, i.e. it shows how many standard deviations the split values and predictions values are from the mean. Note that standardisation has no effect on the output of the regression tree, as the ordering of values in each variable is preserved. As an example of the high variance of a regression tree discussed above, we see that in the second layer, the tree makes a split where it makes one node with over 100,000 observations and one with just 27. The node with just 27 observations is assigned a value less than -8 which is a very small value with the standardisation in mind. It is unlikely that this prediction value will be optimal for unseen data. Even within the node, there is a high variety of values, as the MSE is above 200.

One thing about the tree that may seem strange is that it starts with 1,264,376 samples. This is the case, because the tree is actually taken from a random forest where a data subsampling technique called bagging is applied to a learning set of size 2,000,000. The explanation of this technique will be provided in Section 3.5, where we also will see how to use regression trees in a way which avoids the high variance of estimates. In fact, a single regression tree is rarely used as a prediction model, as it is better as a basis model for ensemble models where multiple regression trees are combined into one predictive model. We will see examples of this in Section 3.5 and 3.6.

We will not dive too much into the computational complexity of constructing a regression tree, as that would require us to consider many different cases depending on how the splits are made. A comprehensive description of the computational cost of regression trees is given in (Louppe, 2014, chap. 5). In this thesis, we will just note that the computational complexity of a regression tree is very low compared to other machine learning methods due to its simple structure. Instead of showing this analytically, we will see it empirically in Section 4. The low computational cost of tree construction makes regression tree-based modelling one of the most attractive ways of creating prediction models with high-level interactions.

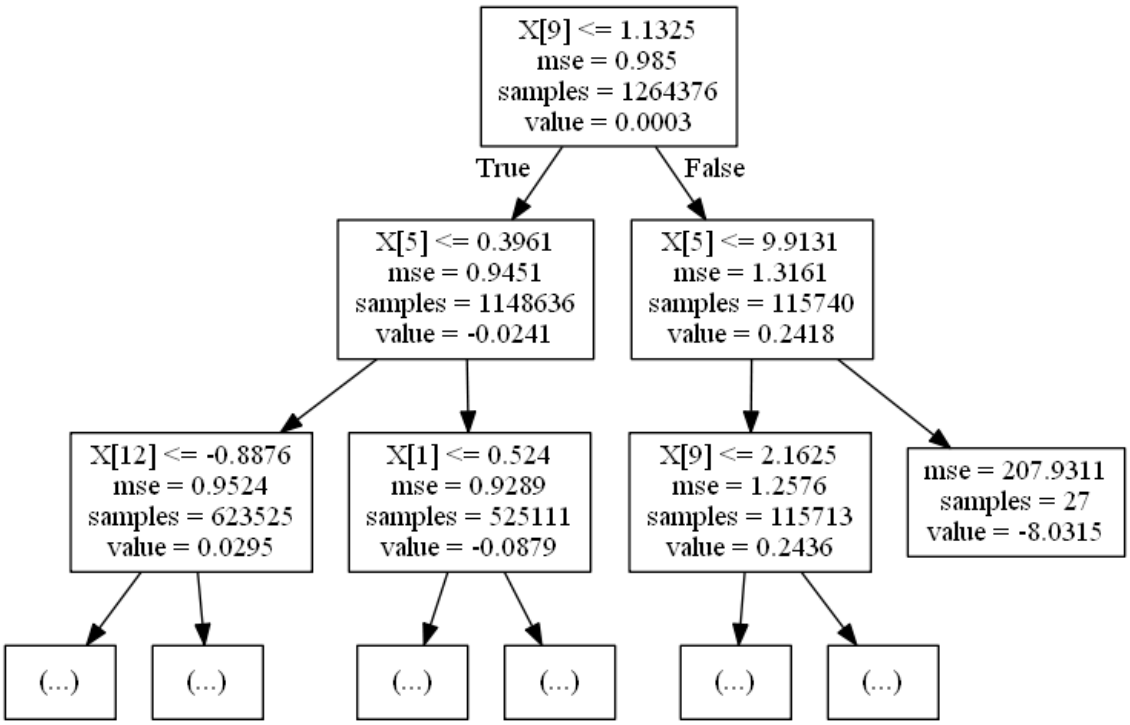


Figure 4: Regression tree made with EUR/CHF for 2 ticks predictions. $X[j] <= s$ is the splitting variable and splitting value, mse is the MSE in the node, samples is the number of observations in the node and value is the prediction value $\hat{c}_m$.

3.5 Random Forest

Random forest attempts to mitigate the problem with high variance for deep trees explained in the previous section by averaging over multiple regression trees. This means that our predictive function will be of the form

$$f(X_t) = \frac{1}{M} \sum_{m=1}^{M} g_m(X_t), \tag{3.10}$$

where each g_m is a regression tree build with a learning algorithm which is very similar to Algorithm 3.2. The major difference between the learning algorithm used for trees in a random forest and the one used for a regression tree is that the first one contains randomness. In fact, without this randomness, the random forest would be exactly the same as a normal regression tree, as we would have that $g_1 = g_2 = \cdots = g_M$. As all previous learning algorithms have been deterministic, we will start by discussing what effect randomness in the learning algorithm has on the resulting predictive model. After that, we will go more in-depth with which parts of the random forest learning algorithm is random and how we can choose optimal randomness. We assume that the randomness of the learning algorithm $\mathcal{A}$ is contained in a random seed stored in the complexity parameter vector θ_c. We will denote the random seed for $g_1, \ldots, g_M$ by $\xi_1, \ldots, \xi_M$ and assume

that they all have the same distribution and are independent (iid), meaning that

$$g_i(X_t) \sim g_j(X_t), \quad i, j \in \{1, \dots, M\}, \quad t = 1, 2, \dots$$

As we will consider many sources of randomness in this section, we will use subscripts when taking expectation and denoting predictive functions to indicate what we consider as stochastic. If we denote the vector of random seeds by ξ, this means that we will write (3.10) as

$$f_{\mathcal{L},\xi}(X_t) = \frac{1}{M} \sum_{m=1}^{M} g_{\mathcal{L},\xi_m}(X_t). \tag{3.11}$$

Similarly, the bias-variance decomposition in (3.3) can now be written as

$$\begin{aligned} EPE_t[f_{\mathcal{L},\xi}] &= \mathbb{E}_{\mathcal{L},\xi}[\big(Y_t - f_{\mathcal{L},\xi}(X_t)\big)^2 | X_t = \mathbf{x}_t)] \\ &= \text{bias}[f_{\mathcal{L},\xi}(\mathbf{x}_t)]^2 + \text{Var}[f_{\mathcal{L},\xi}(\mathbf{x}_t)] + \sigma_t^2, \end{aligned}$$

where

$$\begin{aligned} \text{bias}[f_{\mathcal{L},\xi}(\mathbf{x}_t]^2 &= \big(f_{\mathcal{B}}(\mathbf{x}_t) - \mathbb{E}_{\mathcal{L},\xi}[f_{\mathcal{L},\xi}(\mathbf{x}_t)]\big)^2, \\ \text{Var}[f_{\mathcal{L},\xi}(\mathbf{x}_t] &= \mathbb{E}_{\mathcal{L},\xi}[\big(\mathbb{E}_{\mathcal{L},\xi}[f_{\mathcal{L},\xi}(\mathbf{x}_t)] - f_{\mathcal{L},\xi}(\mathbf{x}_t)\big)^2]. \end{aligned}$$

If $f_{\mathcal{L},\xi}$ consists of just a single tree, there is no reason to believe that this model should be better than just a normal regression tree with deterministic learning algorithm; by increasing randomness, it is more likely that bias and variance would increase rather than decrease. However, when considering a random forest with many trees, the result is different. In the following, we will discuss how bias and variance changes as we increase the number of trees in a random forest. To simplify the notation a bit, we will denote the mean and variance of a prediction of a single tree in the forest at time t by

$$\begin{aligned} \mu_{\mathcal{L},\xi_m}(\mathbf{x}_t) &:= \mathbb{E}_{\mathcal{L},\xi_m}[g_{\mathcal{L},\xi_m}(\mathbf{x}_t)], \\ \sigma^2_{\mathcal{L},\xi_m}(\mathbf{x}_t) &:= \text{Var}[g_{\mathcal{L},\xi_m}(\mathbf{x}_t)]. \end{aligned}$$

Considering the squared bias term first, we see that for any M in $\mathbb{N}$ it holds that

$$\begin{aligned} \mathbb{E}_{\mathcal{L},\xi}[f_{\mathcal{L},\xi}(\mathbf{x}_t)] &= \mathbb{E}_{\mathcal{L},\xi_1,\dots,\xi_M}\big[\frac{1}{M} \sum_{m=1}^{M} g_{\mathcal{L},\xi_m}(\mathbf{x}_t)\big] \\ &= \frac{1}{M} \sum_{m=1}^{M} \mathbb{E}_{\mathcal{L},\xi_m}[g_{\mathcal{L},\xi_m}(\mathbf{x}_t)] \\ &= \frac{1}{M} \sum_{m=1}^{M} \mathbb{E}_{\mathcal{L},\xi_1}[g_{\mathcal{L},\xi_1}(\mathbf{x}_t)] \\ &= \mu_{\mathcal{L},\xi_1}(\mathbf{x}_t), \end{aligned}$$

where we use that $\xi_1, \ldots, \xi_M$ are iid. Since this is the only part of the bias term which depends on the selected prediction model, it follows that

$$\text{bias}[f_{\mathcal{L},\xi}(\mathbf{x}_t)]^2 = \big(f_{\mathcal{B}}(\mathbf{x}_t) - \mu_{\mathcal{L},\xi_1}(\mathbf{x}_t)\big)^2 = \text{bias}[g_{\mathcal{L},\xi_1}(\mathbf{x}_t)]^2,$$

meaning that the squared bias term is constant for all M in $\mathbb{N}$. As the squared bias term is likely to be larger than the one coming from the non-randomised regression tree and the noise term σ_t^2 is the same for all prediction models, random forest needs to achieve a low prediction variance to be useful.

Before calculating the prediction variance, we will consider the correlation between predictions made by two randomised models with different random seed. Due to the iid assumption of the random seeds, it is sufficient to calculate the correlation between two random models with seeds ξ_i and ξ_j for $i \neq j$. By definition, we have that

$$\begin{aligned}
\rho\big(g_{\mathcal{L},\xi_i}(\mathbf{x}_t), g_{\mathcal{L},\xi_j}(\mathbf{x}_t)\big) =& \frac{\mathbb{E}_{\mathcal{L},\xi_i,\xi_j}[(g_{\mathcal{L},\xi_i}(\mathbf{x}_t) - \mu_{\mathcal{L},\xi_i}(\mathbf{x}_t))(g_{\mathcal{L},\xi_j}(\mathbf{x}_t) - \mu_{\mathcal{L},\xi_j}(\mathbf{x}_t))]}{\sigma_{\mathcal{L},\xi_i}(\mathbf{x}_t)\sigma_{\mathcal{L},\xi_j}(\mathbf{x}_t)} \\
=& \frac{\mathbb{E}_{\mathcal{L},\xi_i,\xi_j}[g_{\mathcal{L},\xi_i}(\mathbf{x}_t)g_{\mathcal{L},\xi_j}(\mathbf{x}_t) - g_{\mathcal{L},\xi_i}(\mathbf{x}_t)\mu_{\mathcal{L},\xi_j}(\mathbf{x}_t)]}{\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)} \\
&+ \frac{\mathbb{E}_{\mathcal{L},\xi_i,\xi_j}[\mu_{\mathcal{L},\xi_i}(\mathbf{x}_t)\mu_{\mathcal{L},\xi_j}(\mathbf{x}_t) - g_{\mathcal{L},\xi_j}(\mathbf{x}_t)\mu_{\mathcal{L},\xi_i}(\mathbf{x}_t)]}{\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)} \\
=& \frac{\mathbb{E}_{\mathcal{L},\xi_i,\xi_j}[g_{\mathcal{L},\xi_i}(\mathbf{x}_t)g_{\mathcal{L},\xi_j}(\mathbf{x}_t) - \mu^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)]}{\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)},
\end{aligned}$$

which we for notational convenience will denote by $\rho_{\mathcal{L},\xi_i,\xi_j}(\mathbf{x}_t)$.

The correlation measures how random the learning algorithms of the trees are. If $\rho_{\mathcal{L},\xi_i,\xi_j}(\mathbf{x}_t) = 1$, there is no randomness, whereas $\rho_{\mathcal{L},\xi_i,\xi_j}(\mathbf{x}_t) = 0$ means perfectly random models. The following calculation shows the impact this correlation has on the variance of predictions made by a random forest model:

$$\begin{aligned}
\text{Var}[f_{\mathcal{L},\xi}(\mathbf{x}_t)] =& \mathbb{E}_{\mathcal{L},\xi}[f_{\mathcal{L},\xi}(\mathbf{x}_t)^2] - \mathbb{E}_{\mathcal{L},\xi}[f_{\mathcal{L},\xi}(\mathbf{x}_t)]^2 \\
=& \mathbb{E}_{\mathcal{L},\xi_1,\ldots,\xi_M}\big[\big(\frac{1}{M}\sum_{m=1}^{M} g_{\mathcal{L},\xi_m}(\mathbf{x}_t)\big)^2\big] - \mathbb{E}_{\mathcal{L},\xi_1,\ldots,\xi_M}\big[\frac{1}{M}\sum_{m=1}^{M} g_{\mathcal{L},\xi_m}(\mathbf{x}_t)\big]^2 \\
=& \frac{1}{M^2}\Big(\mathbb{E}_{\mathcal{L},\xi_1,\ldots,\xi_M}\big[\big(\sum_{m=1}^{M} g_{\mathcal{L},\xi_m}(\mathbf{x}_t)\big)^2\big] - \mathbb{E}_{\mathcal{L},\xi_1,\ldots,\xi_M}\big[\sum_{m=1}^{M} g_{\mathcal{L},\xi_m}(\mathbf{x}_t)\big]^2\Big) \\
=& \frac{1}{M^2}\Big(\mathbb{E}_{\mathcal{L},\xi_1,\ldots,\xi_M}\big[\sum_{i,j=1,\ldots,M} g_{\mathcal{L},\xi_i}(\mathbf{x}_t)g_{\mathcal{L},\xi_j}(\mathbf{x}_t)\big] - \big(M\mu_{\mathcal{L},\xi_1}(\mathbf{x}_t)\big)^2\Big) \\
=& \frac{1}{M^2}\Big(\sum_{i,j=1,\ldots,M} \mathbb{E}_{\mathcal{L},\xi_i,\xi_j}\big[g_{\mathcal{L},\xi_i}(\mathbf{x}_t)g_{\mathcal{L},\xi_j}(\mathbf{x}_t)\big] - M^2\mu^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)\Big) \\
=& \frac{1}{M^2}\Big(M\mathbb{E}_{\mathcal{L},\xi_1}\big[g_{\mathcal{L},\xi_1}(\mathbf{x}_t)^2\big] + (M^2 - M)\mathbb{E}_{\mathcal{L},\xi_1,\xi_2}\big[g_{\mathcal{L},\xi_1}(\mathbf{x}_t)g_{\mathcal{L},\xi_2}(\mathbf{x}_t)\big] - M^2\mu^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)\Big)
\end{aligned}$$

$$
\begin{aligned}
=&\frac{1}{M^2}\Big(M\big(\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)+\mu^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)\big)\\
&+(M^2-M)\big(\rho_{\mathcal{L},\xi_1,\xi_2}(\mathbf{x}_t)\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)+\mu^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)\big)-M^2\mu^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)\Big)\\
=&\frac{\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)}{M}+\rho_{\mathcal{L},\xi_1,\xi_2}(\mathbf{x}_t)\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)-\rho_{\mathcal{L},\xi_1,\xi_2}(\mathbf{x}_t)\frac{\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)}{M}\\
=&\rho_{\mathcal{L},\xi_1,\xi_2}(\mathbf{x}_t)\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)+\frac{1-\rho_{\mathcal{L},\xi_1,\xi_2}(\mathbf{x}_t)}{M}\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t). \qquad (3.12)
\end{aligned}
$$

The second term of (3.12) can clearly be shrunk close to 0 by increasing the number of trees in the forest, while the first term only can be shrunk by increasing the randomness of the trees. It is though worth noting that the variance of a single tree $\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)$ may increase as we increase the amount of randomness. In addition, the bias will also typically increase, as we increase randomness of the model. When considering real data, we cannot tell what $\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}_t)$, $\rho_{\mathcal{L},\xi_1,\xi_2}(\mathbf{x}_t)$ and bias$[f_{\mathcal{L},\xi}]$ are, but by having complexity parameters controlling the amount of randomness in the model, we can approximate optimal randomness through cross-validation.

Now that we have shown why it may be desirable to build a model with a randomised learning algorithm, we will have a look at how randomness is applied in a random forest through bootstrap aggregation (bagging) and randomised feature selection.

Bagging is a data subsampling technique proposed by Breiman (1996) in which datasets of the same size as $\mathcal{L}$ are drawn from $\mathcal{L}$ with replacement. At each draw, all samples have the same probability of being drawn, meaning that the probability of a sample being in the new data set $\mathcal{B}$ is

$$
\mathbb{P}(\mathbf{x}_t\in\mathcal{B})=1-\mathbb{P}(\mathbf{x}_t\notin\mathcal{B})=1-\big(1-\frac{1}{N}\big)^N\approx 1-\mathrm{e}^{-1}\approx 0.632,
$$

using that $\big(1-\frac{z}{N}\big)^N\to\mathrm{e}^{-z}$ for $N\to\infty$. Each of these new sets are said to be a bootstrapped dataset. Using bagging, is to construct a new bootstrapped set for each tree in the random forest, meaning that each tree will be trained on different subsets of $\mathcal{L}$. As each of these bootstrapped subsets only contain approximately 63% of the data, bagging is likely to have a decorrelating effect on the trees. The use of bagging explains the seemingly random number of samples used to construct the regression tree pictured in Figure 4, as $\frac{1{,}264{,}376}{0{,}632}\approx 2,000,000$ which is the actual number of samples in the learning set used to construct the tree.

Not only is bagging useful for reducing prediction variance, it also allows us to estimate the generalisation error (3.1) in a way which is much faster than cross-validation. As each tree only uses 63% of the learning set, there is 37% of the learning set which can be considered as unseen

data for each tree. If we denote the bootstrap datasets used for each tree by $\mathcal{B}_1, \ldots, \mathcal{B}_M$ and define

$$M_{(-t)} := \{m \in \{1, \ldots, M\} \mid (\mathbf{x}_t, y_t) \notin \mathcal{B}_m, \quad m = 1, \ldots, M\},$$

we can define an error measure by

$$L^{\text{OOB}}(f_{\mathcal{L},\xi}) = \frac{1}{N} \sum_{t=1}^{N} L\big(\frac{1}{|M_{(-t)}|} \sum_{m \in M_{(-t)}} g_{\mathcal{B}_m,\xi_m}(\mathbf{x}_t), y_i\big), \tag{3.13}$$

which is called the out of bag estimate. By estimating generalisation error with this measure, we only have to perform one fit of the model as opposed to cross-validation where 3-10 fits usually are made. Wolpert & Macready (1999) show that in many cases this estimate is at least as good at estimating generalisation error as cross-validation. The only shortfall of this technique is that fitting and testing data are not divided into consecutive blocks as it was the case in cross-validation. This may result in a too optimistic estimate of the generalisation error, as we will be using the future to predict the past.

The second source of randomness, randomised feature selection, is first introduced by Wilder et al. (1997) and then later used in combination with bagging by Breiman (2001a) to form what we will refer to as the random forest model. Constructing a tree with randomised feature selection is almost the same procedure as outlined in 3.2 but with the key difference that only a random subset of the p variables are considered for splitting in step 2. This forces the trees to rely on multiple variables and decorrelates the trees in the forest further. Additionally, by only considering a subset of the variables at each split, the computation time can be reduced significantly. If we choose to consider a low number of variables at each split, we will increase the decorrelation of the trees which according to (3.12) is likely to decrease the prediction variance. A high amount of randomness does though come with a cost; the bias of the random forest will increase. Finding the right trade-off between bias and variance, by selecting the number of variables we consider at each split, will prove to be key for the performance of the random forest model in Section 4.

Similar to the shrinkage methods discussed in Section 3.2, the random forest model can be used for feature selection. One way of doing this is a method introduced by Breiman (2001b) in which we consider the decrease in loss during tree construction caused by each variable. We call the loss in a single node impurity and will denote it by $i(n)$. Similarly, we will define $i(n_l)$ and $i(n_r)$ as the impurity in the left and right node respectively made by the split of node n. With this notation in place, we can define the impurity decrease of optimally splitting node n (given the random subset of variables available in the node) by

$$\Delta i(n) = i(n) - p_{n_l} i(n_l) - p_{n_r} i(n_r), \tag{3.14}$$

where p_{n_l} and p_{n_r} are the proportion of samples from node n going into the left and right node respectively. we can then use this measure to define the mean decrease impurity measure (MDI) of variable X_j by

$$\text{MDI}(X_j) = \frac{1}{M}\sum_{m=1}^{M}\sum_{n\in g_{\mathcal{B}_m,\xi_m}} \mathbb{1}_{\{j_n=j\}} p_n \Delta i(n), \tag{3.15}$$

where p_n is the proportion of samples from $\mathcal{L}$ reaching node n and $\mathbb{1}_{\{j_n=j\}}$ is an indicator function taking the value 1 when the split in node n is performed on variable X_j and 0 otherwise. A high MDI will indicate that a variable is important, as it decreases the loss significantly when being split on. Unlike feature selection with shrinkage methods, MDI is capable of measuring the importance of non-directional variables. This is the case, as splits in nodes are made conditional on previous splits. The tree will usually split on directional variables first and then use the non-directional ones to 'validate' the directional signal. One should though not perform feature selection solely based on MDI, since selecting the top-ranked variables will not necessarily yield a great fit. If we, for instance, have a variable X_j which is important for the regression and put two copies of it into the random forest model, it will assign a high MDI value to both of them, which is clearly not ideal. It is though generally true that if the MDI value of a variable is close to 0, it is unlikely to be important for the prediction task.

As mentioned in Section 3.4, constructing a regression tree can be done quickly and even faster when we only consider a subset of variables at each split. Since random forest is based on independent regression trees which easily can be parallelised, constructing a random forest is a computationally cheap way to create a prediction model with high-level interactions. In this thesis, we will be using the Python implementation `sklearn.ensemble.RandomForestRegressor` to construct random forest models.

If a prediction model has a high computational cost, it is usually because it tries to model many variable interactions similarly to the random forest but in a more expensive way. Given that we through cross-validation (or OOB scoring) of a random forest's complexity parameters find that a very complex model is optimal (non-restrictive stopping criteria), it is likely that high complexity in these more expensive models is optimal as well. Because of this and random forest's low computational cost, it can be useful to run a random forest model prior to more expensive prediction models to get an idea of optimal model complexity.

3.6 Stochastic Gradient Boosted Trees

Another way of constructing a tree-based ensemble is stochastic gradient boosting. The term 'boosting' refers to an iterative procedure in which an additive model

$$f(X_t) = \sum_{m=1}^{M} g_m(X_t)$$

is constructed by adding one term to the sum at a time. In each iteration of the learning algorithm, a new predictive function

$$f^k(X_t) := f^{k-1}(X_t) + g_k(X_t) \tag{3.16}$$

is constructed, where the specifications of g_k depends on the residuals $L\big(y_1, f^{k-1}(\mathbf{x}_1)\big), \ldots, L\big(y_N, f^{k-1}(\mathbf{x}_N)\big)$ obtained from the previous iteration.

When we consider gradient boosting, g_k is a function fitted on the gradient of the residuals, i.e. it will be fitted on the pseudo residuals defined by

$$d_t = \frac{\partial L\big(y_t, f^{k-1}(\mathbf{x}_t)\big)}{\partial f^{k-1}(\mathbf{x}_1)}$$

This technique was invented by Friedman (2001) and was quickly extended by Friedman (2002) to stochastic gradient descent where the learning algorithm is randomised by subsampling the learning set used for each tree. In this thesis, we will use a slight modification of Friedman's original algorithm which differs in the following ways:

a) Randomised feature selection is applied.

b) The objective function will include a regularisation term

$$\Omega(f_{\mathcal{L}}) = \sum_{m=1}^{M} \Omega(g_{\mathcal{L},m}),$$

where

$$\Omega(g_{\mathcal{L},m}) = \gamma|R_m| + \frac{\lambda}{2}\sum_{i=1}^{|R_m|} \omega_{m,i}^2.$$

c) We will be making a second order approximation of the loss terms instead of only considering the gradient.

In the above, $|R_m|$ is the number of leaves in tree $g_{\mathcal{L},m}$ and $\omega_{m,1}, \ldots, \omega_{m,|R_m|}$ are the values of the leaves in the tree. The idea of modification a) is the same as in random forest and modification b) is a combination of stopping criteria c) used in Algorithm 3.2 and a $\mathbf{L}^2$-penalty known from ridge regression. This version of stochastic gradient boosting is often referred to as extreme gradient boosting, or just XGB. It can be performed in Python with the implementation `xgboost.XGBRegressor`.

The construction of each tree in this boosted ensemble follows Algorithm 3.2 but with some modifications made to step 2. Similar to random forest, we apply randomness to the learning algorithm through data subsampling and randomised feature selection. The subsampling here is though not bagging, as we will just draw samples from $\mathcal{L}$ without replacement until a desired sample size has been reached. While randomness in the learning algorithm was a necessity for the random forest to be any different from a single regression tree, the iterative procedure of the boosted tree learning algorithm uses the randomness as a source of regularisation together with adjustments of the other complexity parameters tree depths, minimum leaf sizes, number of trees and parameters λ and γ. To give an intuition about how these parameters impact the learning algorithm, we will in the following derive how trees are constructed for the XGB learning algorithm. We will derive the construction of these trees for a general twice differentiable loss function and afterwards discuss the specific case of MSE loss that we are focusing on in this thesis.

Assume we use M trees in our boosted model. Then after iteration k, we have a prediction function of the form given in (3.16) which makes predictions

$$\hat{y}_t^k = f_{\mathcal{L},k}(\mathbf{x}_t) = \sum_{m=1}^{k} g_{\mathcal{L},m}(\mathbf{x}_t) = \hat{y}_t^{k-1} + g_{\mathcal{L},k}(X_t).$$

Assuming that

$$f_{\mathcal{L},0}(\mathbf{x}) := 0,$$

this recursive structure implies that we only need to derive how $g_{\mathcal{L},k}$ is constructed in each iteration. In iteration k, the objective function is

$$\begin{aligned} \text{obj}(f_{\mathcal{L},k}) &= \sum_{t=1}^{N} L(y_t, \hat{y}_t^k) + \sum_{m=1}^{k} \Omega(g_{\mathcal{L},m}) \\ &= \sum_{t=1}^{N} L(y_t, \hat{y}_t^{k-1} + g_{\mathcal{L},k}) + \sum_{m=1}^{k} \Omega(g_{\mathcal{L},m}). \end{aligned} \tag{3.17}$$

Denoting the time t first and second order derivatives of $L(y_t, \hat{y}_t^{k-1})$ w.r.t. $\hat{y}_t^{k-1}$ by

$$\begin{aligned} d_{t,1} &= \frac{\partial L(y_t, \hat{y}_t^{k-1})}{\partial \hat{y}_t^{k-1}}, \\ d_{t,2} &= \frac{\partial^2 L(y_t, \hat{y}_t^{k-1})}{\partial (\hat{y}_t^{k-1})^2}, \end{aligned}$$

we can make a second order Taylor approximation of the loss term in (3.17) to get

$$\text{obj}(f_{\mathcal{L},k}) \approx \sum_{t=1}^{N} \left(L(y_t, \hat{y}_t^{k-1}) + d_{t,1} g_{\mathcal{L},m}(\mathbf{x}_t) + \frac{1}{2} d_{t,2} g_{\mathcal{L},m}(\mathbf{x}_t)^2 \right) + \sum_{m=1}^{k} \Omega(g_{\mathcal{L},m}).$$

Since $L(y_t, \hat{y}_t^{k-1})$ and $\Omega(g_{\mathcal{L},1}), \ldots, \Omega(g_{\mathcal{L},k-1})$ are independent of $g_{\mathcal{L},k}$, they can be removed from the object function. We are then left with the objective function for tree k:

$$\begin{aligned}\text{obj}(g_{\mathcal{L},k}) &= \sum_{t=1}^{N} d_{t,1} g_{\mathcal{L},m}(\mathbf{x}_t) + \frac{1}{2} d_{t,2} g_{\mathcal{L},m}(\mathbf{x}_t)^2\big) + \Omega(g_{\mathcal{L},k}) \\ &= \sum_{t=1}^{N} d_{t,1} g_{\mathcal{L},m}(\mathbf{x}_t) + \frac{1}{2} d_{t,2} g_{\mathcal{L},m}(\mathbf{x}_t)^2\big) + \gamma|R_k| + \frac{\lambda}{2}\sum_{i=1}^{|R_k|}\omega_{k,i}^2. \end{aligned} \tag{3.18}$$

Since $g_{\mathcal{L},k}$ is a regression tree, it only takes finitely many values $\omega_{k,1}, \ldots, \omega_{k,|R_k|}$, and we can group the observations by these:

$$N_i := \{t \,|\, g_{\mathcal{L},k}(\mathbf{x}_t) = \omega_{k,i}\}.$$

Now, we can write (3.18) as

$$\begin{aligned}\text{obj}(g_{\mathcal{L},k}) &= \sum_{i=1}^{|R_k|}\left(\big(\sum_{t\in N_i} d_{t,1}\big)\omega_{k,i} + \frac{1}{2}\big(\sum_{t\in N_i} d_{t,2}\big)\omega_{k,i}^2\right) + \gamma|R_k| + \frac{\lambda}{2}\sum_{i=1}^{|R_k|}\omega_{k,i}^2 \\ &= \sum_{i=1}^{|R_k|}\left(\big(\sum_{t\in N_i} d_{t,1}\big)\omega_{k,i} + \frac{1}{2}\big(\big(\sum_{t\in N_i} d_{t,2}\big) + \lambda\big)\omega_{k,i}^2\right) + \gamma|R_k| \\ &= \sum_{i=1}^{|R_k|}\left(D_{i,1}\omega_{k,i} + \frac{1}{2}(D_{i,2}+\lambda)\omega_{k,i}^2\right) + \gamma|R_k|, \end{aligned} \tag{3.19}$$

where

$$D_{i,l} := \sum_{t\in N_i} d_{t,l}, \quad l = 1, 2.$$

As $\omega_{k,1}, \ldots, \omega_{k,|R_k|}$ are independent, we may optimise each term in the sum (3.19) separately. We note that each term is convex in $\omega_{k,i}$, and we may hence differentiate to get

$$\omega_{k,i}^* = -\frac{D_{i,1}}{D_{i,2}+\lambda}. \tag{3.20}$$

Inserting this into (3.19) gives the tree score

$$\text{obj}^*(g_{\mathcal{L},k}) = -\frac{1}{2}\sum_{i=1}^{|R_k|}\frac{D_{i,1}^2}{D_{i,2}+\lambda} + \gamma|R_k| \tag{3.21}$$

which is the lowest possible value of the object function for a given tree structure (split of regions).

When we search for the best split of region r_i in step 2 of Algorithm 3.2, we are looking for the split which reduces the trees object function the most, i.e. we want to solve

$$max_{j,s}\ \frac{1}{2}\left(\frac{D_{i,1}^l(j,s)^2}{D_{i,2}^l(j,s)+\lambda} + \frac{D_{i,1}^r(j,s)^2}{D_{i,2}^r(j,s)+\lambda} - \frac{D_{i,1}^2}{D_{i,2}+\lambda}\right) - \gamma, \tag{3.22}$$

where the superscripts l and r refer to the sums of derivatives in the left and right regions formed by splitting variable X_j at value s. Note that the term we a minimising in (3.22) is exactly the

object function evaluated with no split minus the object function evaluate after splitting variable X_j at value s in the given leaf. Also, subtracting gamma is exactly the same as stopping criteria c) in Algorithm 3.2.

We will now turn our attention to the case where the loss function is MSE as this is the loss function we will be using in the empirical studies in Section 4. In this case, a lot of terms simplify as we have that

$$D_{i,1} = \sum_{t \in N_i} \frac{\partial L(y_t, \hat{y}_t^{k-1})}{\partial \hat{y}_t^{k-1}} = \sum_{t \in N_i} (\hat{y}_t^{k-1} - y_t),$$
$$D_{i,2} = \sum_{t \in N_i} \frac{\partial^2 L(y_t, \hat{y}_t^{k-1})}{\partial (\hat{y}_t^{k-1})^2} = \sum_{t \in N_i} 1 = |N_i|,$$

and the value in the leaves will be

$$\omega_{k,i}^* = \frac{\sum_{t \in N_i} (y_t - \hat{y}_t^{k-1})}{|N_i| + \lambda}.$$

This means that for $\lambda = 0$, the values in the leaves will be the mean of the residuals from the previous iteration, i.e. we just fit a normal regression tree on these residuals. As we increase λ, the leaf values will shrink towards 0, meaning that the tree will avoid making large predictions (in absolute terms). This makes λ very useful for regularisation, as we will clearly decrease the variance and increase the bias of our predictions when increasing λ.

When having constructed a tree $g_{\mathcal{L},m}$, we will in fact only be adding a fraction, α, of it to the current prediction function, i.e. the model will be of the form

$$f_{\mathcal{L}}(\mathbf{x}_t) = \sum_{m=1}^{M} \alpha g_{\mathcal{L},m}(\mathbf{x}_t),$$

where α is the parameter controlling how much of $g_{\mathcal{L},m}$ is added in iteration m of the learning algorithm. We will refer to α as the learning rate of the model, since it controls how much the model 'learns' in each iteration. By choosing a low learning rate, trees constructed in consecutive iterations will be fitted on almost the same pseudo residuals. As each of these trees is constructed with randomness in their learning algorithm, this should decrease the variance of predictions in a way similar to what we saw in the random forest model.

Besides choosing α, λ and γ, we will also have to choose minimum leaf sizes, maximum tree depths, amount of data subsampling, number of features considered at each split and number of trees. It is important to note that when we increase the number of trees, we will increase the complexity of the model which will probably lead to a need for additional regularisation. This makes cross-validation for optimal parameters of a boosted tree ensemble a more complicated task

than it was the case for the random forest, as more trees were always better in the random forest model.

An increased number of parameters to tune means increased computation time. On top of that, each fit during cross-validation takes more time, as the construction of each tree depends on the previous one. Hence, tree construction cannot be parallelised as it was the case for the random forest.

3.7 LSTM Neural Network

Even though we are modelling time series, all the models we have considered so far have only been able to model time structure through the features that we have created. For instance, if we thought that market trends were important for our models, we would include moving averages of past increments in our features. This way of modelling time series can be seen as a bit restrictive, as our models are only capable of using the subset of past information, we choose to feed them. To treat this potential issue, we will in this section discuss a model called long-short term memory neural network (LSTM) invented by Hochreiter & Schmidhuber (1997). The model is a special type of recurrent neural networks (RNNs), where information about previous inputs and outputs is used to find complicated patterns in the data. In recent years, the model has become very popular within speech recognition, where it shows state of the art results (e.g. Graves et al. (2013)) due to its ability to model complex time series structures with long-range dependencies. Motivated by these promising results, we will in this section investigate whether the LSTM model also can perform well when we consider financial data. We will start by giving an introduction to recurrent neural networks and then discuss how the LSTM extends the RNN concept to form a model capable of capturing long-range dependencies in time series. The figures provided in this section are taken from Olah (2015).

A recurrent neural network produces an output, $\mathbf{h}_t$, which is recursively constructed as a function of the output from time $t-1$ and the time t input. This output will be a d-dimensional vector which we may use as an input for another function which produces predictions. Hence, when we say that we use a recurrent neural network, it means that we have an intermediate step in our prediction model in which we make a specific recursive data transformation prior to making predictions. RNNs can have much more complicated structures than it was the case in the MARS algorithm, but the concept of transforming data prior to making predictions is the same. Figure 5 shows a general unfolded RNN. In each of the A-boxes, a transformation of the input, which depends on the RNN type, happens.

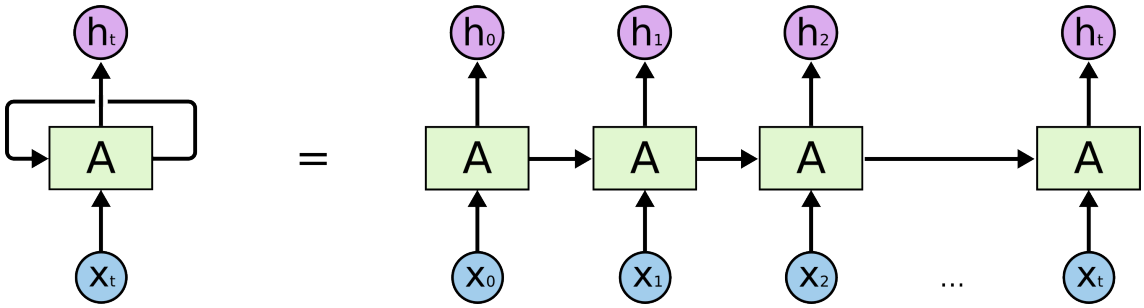


Figure 5: General RNN

If we consider a RNN in its simplest form, the transformation in the A-boxes is given by

$$h_t = \sigma_h(W\mathbf{x}_t + U\mathbf{h}_{t-1} + \mathbf{b}),$$

where we have used the following notation:

- $\mathbf{h}_t$ is the time t output.
- σ_h is an entry-wise function (we will be using tanh) which we will refer to as the activation function.
- $\mathbf{x}_t$ is the time t input from $\mathcal{L}$.
- W is a $d \times p$ matrix.
- U is a $d \times d$ matrix.
- $\mathbf{b}$ is a $d \times 1$ vector which we will refer to as the bias vector.

With the notation in place, we see that $\mathbf{h}_t$ is produced by taking matrix multiplicatives of inputs $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ and then making an entry wise transformation with the function σ_h. In addition to the matrix transformations, we also add the bias-vector before using the activation function. The name 'bias vector' is chosen, because it shifts the transformation and not because it has anything to do with bias of the final prediction model. This simple RNN is pictured in Figure 6, where the A-blocks from Figure 5 are replaced by the simple RNN structure. As mentioned above, the recurrent neural network is not a prediction model in itself. The $\mathbf{h}_t$s are passed on to another function such that we have a prediction function of the form

$$\begin{cases} f(\mathbf{x}_t) &= g(\mathbf{h}_t), \\ h_t &= \sigma_h(W\mathbf{x}_t + U\mathbf{h}_{t-1} + \mathbf{b}). \end{cases}$$

The function g may be a simple OLS model or it may be complex and contain more input transformations where $\mathbf{h}_t$ will be taken as input in the same way as $\mathbf{x}_t$ was taken as input to construct $\mathbf{h}_t$ in the RNN. Each transformation we make before the final prediction step is called a layer in

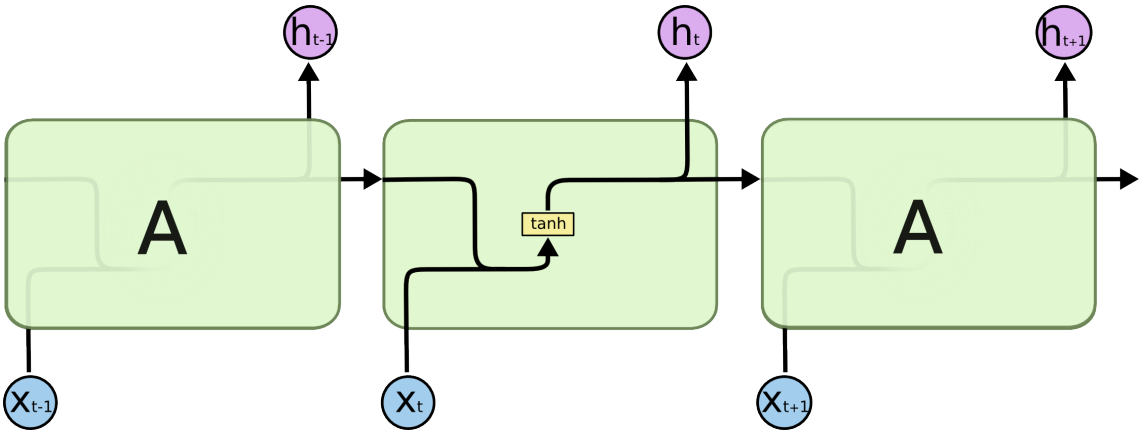


Figure 6: Simple RNN

neural network terms.

What makes neural networks special is that all these transformations are chosen in a data dependent way, where all entries in W, U and $\mathbf{b}$ are parameters chosen during the fitting of the model together with the parameters of the outer function g. This clearly makes the space of possible prediction functions very rich. In fact, the whole motivation of neural networks relies on universal approximation theorems (e.g. Hornik (1991) and Schäfer & Zimmermann (2006)), where it is shown that any continuous function can under mild assumptions be approximated by a neural network. This means that any other prediction function, we have considered so far, could be replicated by a neural network (almost at least). So why would we ever consider any other models than neural networks? The problem with the universal approximation theorems is that they do not provide any algorithm which guarantees that we can find the optimal parameters for the network. If we consider a prediction model with an RNN part, the RNN part alone will have $p(d+1)+d^2$ parameters which we will need to choose optimally.

As the space of parameters is high-dimensional, we have to use an iterative procedure to estimate the parameters. The typical approach, which we also will be using, is to use a method called backpropagation through time (BPTT). It is a form of gradient descent where the prediction errors are propagated backwards through the network and gradients are calculated iteratively by using the chain rule. The method is carried out by first performing a forward pass where we iteratively calculate

$$\begin{cases} \mathbf{a}_t &= Wx_t + U\mathbf{h}_{t-1} + b \\ \mathbf{h}_t &= \sigma(\mathbf{a}_t) \\ \hat{y}_t &= g(\mathbf{h}_t) \end{cases}$$

for $t = 1, \ldots, N$, where $\mathbf{h}_0 = 0$. Next, we calculate the derivative of the loss function w.r.t. each of the parameters in the network, i.e. calculate $\frac{\partial L}{\partial w_{i,j}}$, $\frac{\partial L}{\partial u_{i,l}}$ and $\frac{\partial L}{\partial b_i}$ for $i, l = 1, \ldots, d$ and $j = 1, \ldots, p$. When we update one of these parameters, we adjust the value of the $\mathbf{a}_t$s and thereby the $\mathbf{h}_t$s immediately. This is though far from the only adjustments happening when we update a parameter. Because of the recursive structure, where $\mathbf{h}_{t+1}$ depends on $\mathbf{h}_t$, a change in $\mathbf{h}_t$ affects the value of $\mathbf{h}_{t+1}, \ldots, \mathbf{h}_N$. Using this, we get from the chain rule that

$$\begin{aligned}
\frac{\partial L}{\partial w_{i,j}} &= \sum_{t=1}^{N} \sum_{k=1}^{t} \frac{\partial L}{\partial a_{k,i}} x_{t,j}, \\
\frac{\partial L}{\partial u_{i,l}} &= \sum_{t=1}^{N} \sum_{k=1}^{t} \frac{\partial L}{\partial a_{k,i}} h_{t,l}, \\
\frac{\partial L}{\partial b_i} &= \sum_{t=1}^{N} \sum_{k=1}^{t} \frac{\partial L}{\partial a_{k,i}},
\end{aligned}$$

where

$$\frac{\partial L}{\partial a_{k,i}} = \frac{\partial L}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_{t,i}} \frac{\partial h_{t,i}}{\partial h_{k,i}} \frac{\partial h_{k,i}}{\partial a_{k,i}}.$$

The only term in the above which cannot be computed directly with the values from the forward pass is the third term of $\frac{\partial L}{\partial a_{k,i}}$, which me must compute iteratively:

$$\frac{\partial h_{t,i}}{\partial h_{k,i}} = \frac{\partial h_{t,i}}{\partial h_{t-1,i}} \frac{\partial h_{t-1,i}}{\partial h_{t-2,i}} \cdots \frac{\partial h_{k+1,i}}{\partial h_{k,i}}. \tag{3.23}$$

These terms a most efficiently calculated in reverse order, meaning that we calculate

$$\frac{\partial h_{t,i}}{\partial h_{k,i}}, \quad t = k+1, \ldots, N$$

for $k = N-1, \ldots, 1$. Each iteration only require us to calculate $\frac{\partial h_{t,i}}{\partial h_{k,i}}$ and then multiply it on to terms from the previous iteration.

After having computed the gradients, we update all parameters by subtracting a fraction of their gradient. For instance, we update $w_{i,j}$ by

$$w_{i,j} \leftarrow w_{i,j} - \alpha \frac{\partial L}{\partial w_{i,j}},$$

where α is the parameter controlling the step size of parameter updates. As in the previous section, α is called the learning rate.

For large values of N, calculating all gradients to make a single parameter update is computationally expensive. To handle this problem, we can divide the learning set into disjoint subsets $\mathcal{L}_1, \ldots, \mathcal{L}_q$ consisting of consecutive observations. Instead of just making 1 parameter update when we iterate through the N observations, we make q updates by performing backpropagation on each

of the subsets as described above. These subsets are in machine learning terms called batches, and a run through all the N observations is called an epoch. After running multiple epochs, the RNN should ideally converge (parameter updates become insignificantly small) and leave us with a good prediction model. Unfortunately, this outcome does often not come easy. Firstly, with the high number of parameters involved in an RNN architecture, there is a severe risk of overfitting. Secondly, we are optimizing a non-convex function in a high-dimensional space of parameters, meaning that if the network converges, it is likely towards a local minimum. Regarding the second issue, the randomness in the learning algorithm provided by data subsampling (splitting into batches) can help getting out of local minima (Graves, 2011, p. 30). Choosing a learning rate which increases with the size of the gradients can also help to mitigate this problem (Graves, 2011, p. 29). One example of such a learning rate algorithm is 'adam', developed by Kingma & Ba (2014), which we will be using in this thesis, as it tends to show good results on all types of data. The issue of overfitting, we will discuss at the end of this section.

Besides the two issues described above, which a present for all types of neural networks, the simple RNN also has another shortfall; it struggles to capture long-range dependencies in time series. Because our activation function is tanh, which has a derivative bounded in the interval $(0, 1]$, (3.23) is likely to have an exponential decay as $t - k$ increases. When this happens, the impact which h_k has on the value of h_t decreases rapidly, and the RNN will therefore not be able to capture long-range dependencies. If we try to fix this problem by using another activation function with a gradient that can be larger than 1, we may instead end up with the gradient getting extremely large as $t-k$ increases. The problem of too small gradients is called the vanishing gradient problem and the one with too large gradients is called the exploding gradient problem. A more in-depth discussion of these issues can, for instance, be found in Hochreiter et al. (2001).

To overcome the issues of the vanishing gradient, we will use an extension of the simple RNN called LSTM. It is a model proposed by Hochreiter & Schmidhuber (1997) in which memory blocks are passed between prediction steps in parallel with h_t. A visual representation of an LSTM network can be seen in Figure 7 where the A-blocks clearly have become a bit more complex than it was the case for the simple RNN. Pink nodes are entry-wise operations and the yellow ones are neural networks, i.e. they involve applying an entry-wise function to a matrix multiplicative of the inputs plus a bias vector. The yellow σ-nodes are neural networks with the activation function being the sigmoid function

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}.$$

The outputs of these subnets are used for entry-wise multiplication with other flows in the A-blocks to control how much of the flows is passed on in the network and how much is blocked. Because

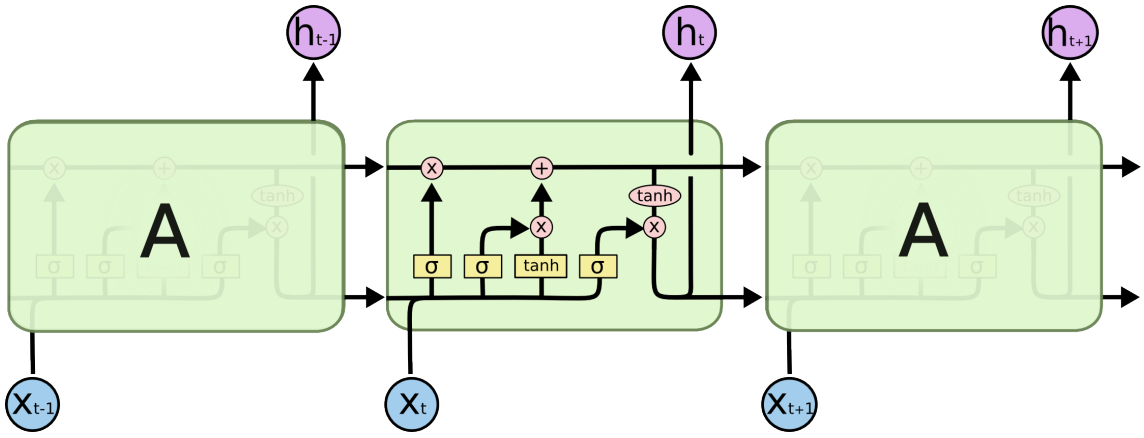


Figure 7: LSTM nerual network

of this role, they are called gates.

The most important part of the LSTM block is the flow in the top of the block which we will call $\mathbf{c}_t$. At time t, $\mathbf{c}_{t-1}$ is taken as an input and an element-wise multiplication with the output of the left-most gate (the forget gate) is performed. If all values of the forget gate's output are close to 1, we say that the gate is open, since this will mean that most of $\mathbf{c}_{t-1}$ is retained at time t. The tanh layer is similar to the one we saw in the simple RNN. Here it is used to form a 'memory candidate', $\tilde{\mathbf{c}}_t$, which after multiplication with the middle gate's (input gate's) output is added to the part of $\mathbf{c}_{t-1}$ which is retained to form the new memory $\mathbf{c}_t$. This new memory is used for two things. Firstly, the time t output, $\mathbf{h}_t$, is made by transforming $\mathbf{c}_t$ by an element-wise tanh operation followed by multiplication with the output of the right-most gate (output gate). Secondly, it is used as an input at time $t+1$. As long as the forget gate stays open through multiple timesteps, the memory can be stored for a long period. This is exactly what allows the LSTM to model long-range dependencies.

The above may seem a little complicated but the LSTM is nothing more than a recursive compu-

tation given by

$$\begin{cases} \mathbf{f}_t &= \sigma(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + \mathbf{b}_f) \\ \mathbf{i}_t &= \sigma(W_i x_t + U_i \mathbf{h}_{t-1} + \mathbf{b}_i) \\ \tilde{\mathbf{c}}_t &= \tanh(W_{\tilde{c}} \mathbf{x}_t + U_{\tilde{c}} \mathbf{h}_{t-1} + \mathbf{b}_{\tilde{c}}) \\ \mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tilde{\mathbf{c}}_t \\ \mathbf{o}_t &= \sigma(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + \mathbf{b}_o) \\ \mathbf{h}_t &= \mathbf{o}_t \circ \tanh(\mathbf{c}_t) \\ \hat{y}_t &= g(\mathbf{h}_t), \end{cases}$$

where all W-matrices are $d \times p$ dimensional, U-matrices are $d \times d$ dimensional and b-vectors are of size d. This makes up a total of $4\big(d(p+1) + d^2\big)$ parameters which is 4 times as many as a simple RNN with the same input and output dimensions.

Updating all these parameters is done by BPTT, as it was the case for the simple RNN. The only difference is that we have a lot more recurrent terms, making the calculations longer. We will not present these calculations here but they can for instance be found in Chen (2016).

We mentioned earlier that with the number of parameters in the simple RNN, it was prone to overfitting. Now that we have multiplied the number of parameters by 4, this is definitely still an issue which requires attention. To treat the problem, we will inject randomness into the learning algorithm of LSTM with a technique called dropout. This technique is a recent invention by Srivastava et al. (2014) in which connections in a neural network are dropped with a probability q. It has shown to be a very effective way of treating overfitting, and it is also easy to implement. To apply dropout to a layer, we just have to simulate a vector (same size as the layer input) of iid Bernoulli variables with probability parameter $1 - q$ and then perform element-wise multiplication with the input. At test time, we will not apply any dropout but instead multiply all weights by $1 - q$ such that all parameters have the same mean as in training. Gal & Ghahramani (2016) has shown that dropouts can be seen as a way of averaging over multiple network architectures which decreases prediction variance and thereby the amount of overfitting. Besides applying dropout, we will also experiment with changing the loss function to a more robust one. We will do this by considering the logcosh loss function shown in Figure 2.

In Figure 8, the in sample and out of sample R^2 is shown as a function of epochs for an LSTM with no regularisation and one with dropout and logcosh loss function. It is clear, that the LSTM with no regularisation overfits very quickly, as the out of sample R^2 starts decreasing after just 3 epochs, while the in sample R^2 keeps decreasing. On the other hand, by applying regularisation,

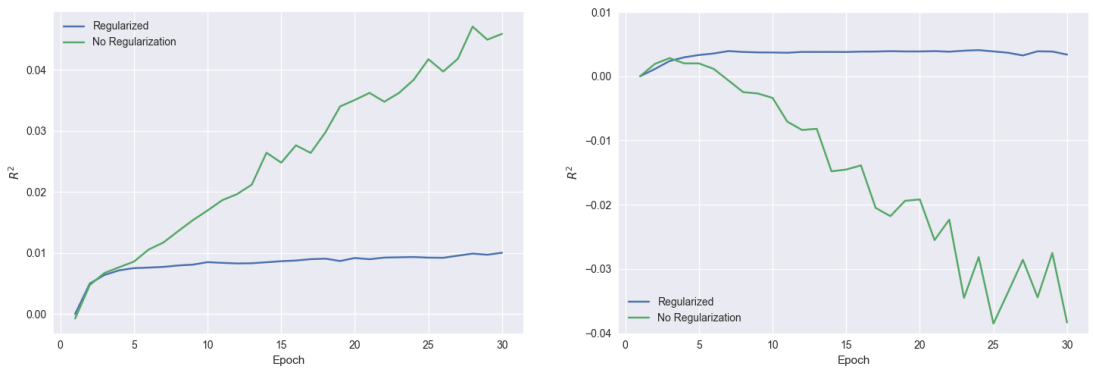
the in sample loss seems to converge and the in sample R^2 only increases slowly after the first few epochs. Most of the regularisation is gained from the dropout, and we will apply this technique in all prediction tasks when producing results in Section 4. Appropriate loss function will be chosen through cross-validation for each currency pair.

The plots in Figure 8 are made using an LSTM with output of size 25 ($d = 25$) which is passed into another transformation given by

$$j_t = (W_j h_t + b_j)^+,$$

where W_j is a 10×25 matrix and b_j is a vector of size 10. Finally, j_t is passed into a standard OLS regression. The transform of h_t into j_t is seen as another layer in the model which we choose to include in this overfitting analysis, as we found through cross-validation that this extra layer generally improved R^2. The parameters involved in constructing j_t are updated together with the LSTM and OLS parameters through BPTT. All LSTM based models in this thesis were constructed with the Python library `Keras`.

One problem with neural network transformations is that they decrease interpretability of the



(a) In sample R^2 (b) Out of sample R^2

Figure 8: Training and testing R^2 for LSTM with training size 1,000,000 and testing size 500,000 drawn from EUR/USD. 20% dropout is apllied to input variable, 50% to recurrent connections and 20% between the two layers.

prediction model. Even though we usually just perform OLS in the final prediction step, the OLS parameters do not have any clear interpretation, as they are assigned to a heavily transformed version of the original input. This is especially true for LSTM based models where the time t input for the final OLS step is constructed by using inputs from all the previous observations. Also, it is difficult to find a logical explanation for parameters used for a specific transformation, as each

transformation is only an intermediate step in the prediction model. This low interpretability is why neural networks are generally classified as black box methods.

Not only are the recurrent LSTM transformation bad for model interpretability; they are also computationally expensive, since predicting at each time step requires many calculations and these are carried out in a recurrent fashion, meaning that they cannot be parellelised much.

4 Emperical Results

Having discussed a lot of theoretical properties for different models, it is now time to evaluate the performance of the models on the FX data. We will evaluate performance for 2 ticks and 50 ticks prediction horizons for each of the four currency pairs EUR/USD, GBP/USD, EUR/CHF and EUR/SWE. All models will be considered but results from the linear shrinkage methods are not presented, as an insignificant amount of penalisation was found to be optimal in all cases, meaning that they were making predictions no different from OLS. First, we will discuss how optimal model structures change as we go from a short prediction horizon to a longer one. Afterwards, we take a closer look at how the different models perform in each of the prediction tasks. We will be using data from 2017 and split it into training and testing sets as shown in Table 1.

	EUR/USD	**GBP/USD**	**EUR/CHF**	**EUR/SWE**
Train size	6,000,000	3,000,000	2,000,000	800,000
Test size	2,500,000	1,300,000	1,100,000	350,000
Hours	All	All	7am-6pm	7am-6pm

Table 1: Training sizes, test sizes and prediction hours. No predictions are made on Saturdays, Sundays, public holidays and Fridays after 8:30 pm.

4.1 Short and Long Prediction Horizons

In Table 2 and 3 below, we see the out of sample R^2 for 2 and 50 ticks predictions for each of the currency pairs. It is evident that the R^2 numbers for 2 ticks predictions are much higher than the ones for 50 ticks predictions. Also, the ranking of methods is much more clear when we consider 2 ticks predictions.

	EUR/USD	**GBP/USD**	**EUR/CHF**	**EUR/SWE**
OLS	6.72	4.43	7.59	7.87
XGB	7.93	5.15	8.92	12.38
RF	7.85	5.15	8.46	11.71
MARS	6.35	4.35	7.49	1.52
LSTM	8.63	5.19	10.42	12.24

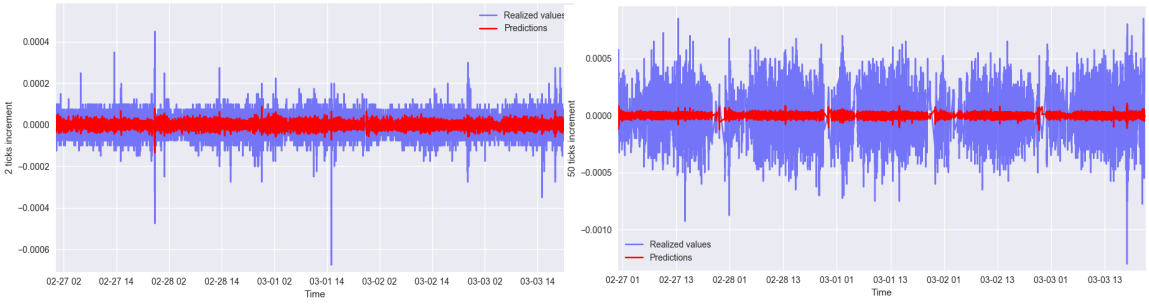
Table 2: Percentage R^2 for 2 ticks predictions

	EUR/USD	GBP/USD	EUR/CHF	EUR/SWE
OLS	0.541	0.403	1.04	1.202
XGB	0.523	0.257	0.90	1.085
RF	0.542	0.289	1.09	1.576
MARS	0.503	0.113	0.97	1.243
LSTM	0.549	0.367	0.89	1.329

Table 3: Percentage R^2 for 50 ticks predictions

If we believe that markets are close to being efficient, it is not a surprising result that predicting a long horizon is more difficult than a short one, as we would expect that any mispricing would be corrected quickly. This means that our information at a given point of time is mostly useful to predict the very near future, e.g. 2 ticks ahead. Because we know much less about what will happen in a more distant future, the size of predictions relative to realized values will decrease when we increase the prediction horizon. This is illustrated in Figure 9 where the 2 ticks predictions are clearly much closer to the real values than it is the case for the 50 ticks predictions.

Less predictability means a higher noise to signal ratio. Particularly for complex models, this



(a) 2 ticks predictions

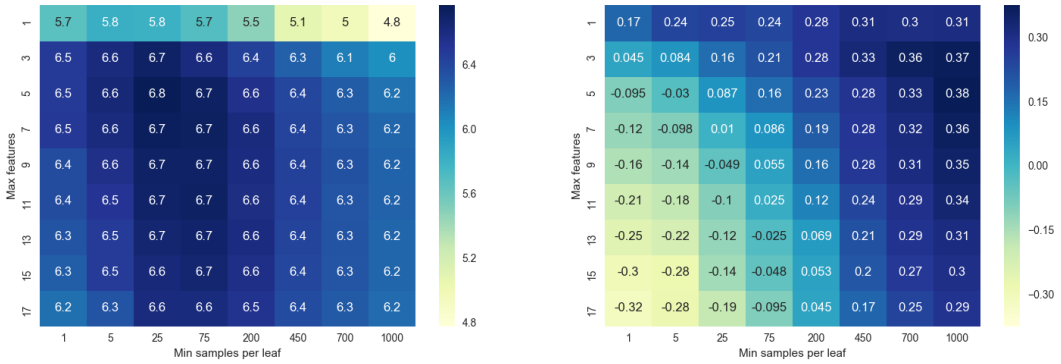
(b) 50 ticks predictions

Figure 9: One week of predictions and realised values for EUR/USD

showed to result in overfitting which forced us to choose restrictive complexity parameters. For instance, in the random forest, we attained optimal complexity for the 50 ticks prediction task with much smaller trees than it was the case for the 2 ticks prediction task. This is shown in Figure 10 where we see that optimal minimum leaf size increases significantly with the prediction horizon. While complexity had to be decreased when increasing the prediction horizon, the optimal amount of randomness seemed to be constant. When increasing the prediction horizon for the other machine learning methods, we also found increased complexity (on a similar scale to RF) and constant randomness to be optimal. This supports the claim in Section 3.5 of random forest

being useful for indicating optimal complexity and randomness in other machine learning models.

Not only optimal complexity changed when increasing the prediction horizon; the importance



(a) 2 ticks predictions

(b) 50 ticks predictions

Figure 10: 5 fold crossvalidated R^2 of random forest with maximum depth 15, using 1,000,000 samples of EUR/USD

of features did as well. As we saw in Figure 3, the most important directional signals for 2 ticks predictions are the discrepancies between mid prices on the different exchanges. This importance of exchange discrepancies we could further confirm by running a linear regression only using these as input variables. By doing this for the 2 ticks predictions task, we achieved 65% (on average) of the R^2 obtained from using all variables, we found useful. In the case of 50 ticks predictions, this number went up to 85%, indicating a very concentrated feature importance. This concentration could also be observed in the random forest feature importance measure where the discrepancies in mid prices accounted for 61% of the feature importances for 50 ticks predictions compared to 50% for the 2 ticks predictions (averaged across the 4 currency pairs). The importance of exchange price discrepancy variables is shown in Figure 11, where we clearly see that it is much easier to make predictions when there is a significant difference between the prices on exchange A and B.

4.2 A Closer Look at Model Performance

Until now, we have only considered the R^2 numbers in Table 2 and 3 to measure the performance of the individual methods. The quality of the models should though also be assessed by how they achieved these results. We will start by discussing the 2 ticks predictions where R^2 rankings were quite consistent and afterwards have a closer look at the more mixed results from the 50 ticks predictions.

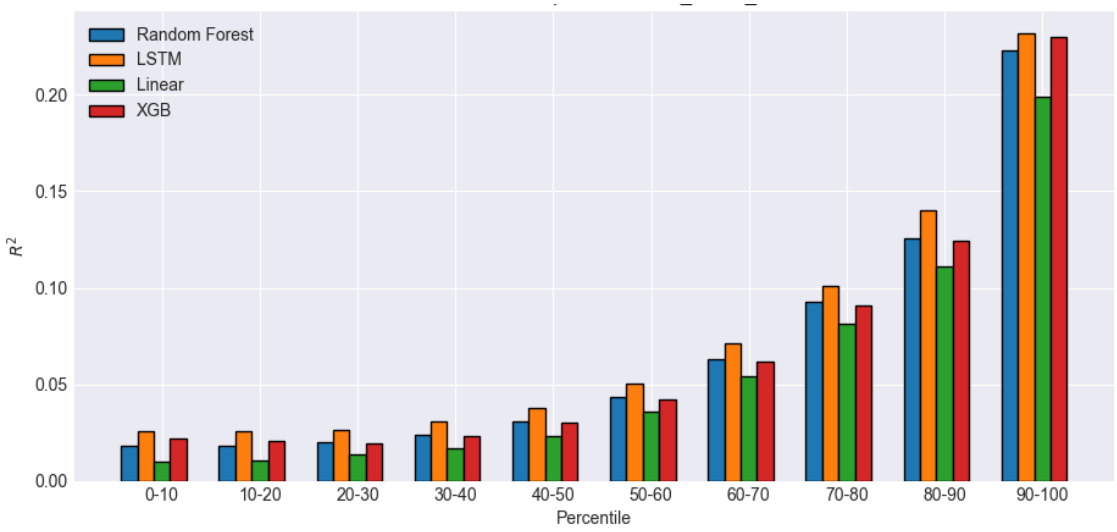


Figure 11: R^2 grouped by size of discrepancy between mid prices on exchange A and B (absolute value)

In the 2 ticks predictions task, the LSTM model had the highest R^2 for 3 out of 4 currency pairs, and it was only slightly lower than XGB for EUR/SWE. XGB seemed to consistently get in second place, RF in third, OLS in fourth and MARS fifth spot. Because the performance of MARS generally was quite poor, we will not spend much time discussing this model, as it was clearly not the best one. To see how we ended up with this ranking, we will start by looking at two R^2-gain plots shown in Figure 12 and 13. To construct them, we calculate

$$\begin{cases} SS_t & = \sum_{i=N+1}^{t}(y_i - \bar{y})^2 \\ SSD_t & = \sum_{i=N+1}^{t}(y_i - \hat{y}_i)^2 \\ R_t & = 1 - \frac{SSD_t}{SS_t} \end{cases}$$

for $t = N+1, \ldots, T$. This gives us a plot arriving at the true R^2 where the slope of it tells us how much R^2 we are gaining (or losing if negative slope) in a specific period.

The general pattern in the 2 ticks predictions task was that LSTM outperformed the other models as it is shown for EUR/USD in Figure 13 where the slope of LSTM seems to be consistently larger than it is the case for the other models. The only currency pair for which this was not the case is EUR/SWE shown in Figure 12. Here, LSTM and XGB seem to perform very equally. This is though also by far the smallest dataset, meaning that any results attained from it are less useful for the evaluating models than it is the case for the other currency pairs. The R^2-gain plots obtained from GBP/USD and EUR/CHF were very similar to the one of EUR/USD and are hence not included.

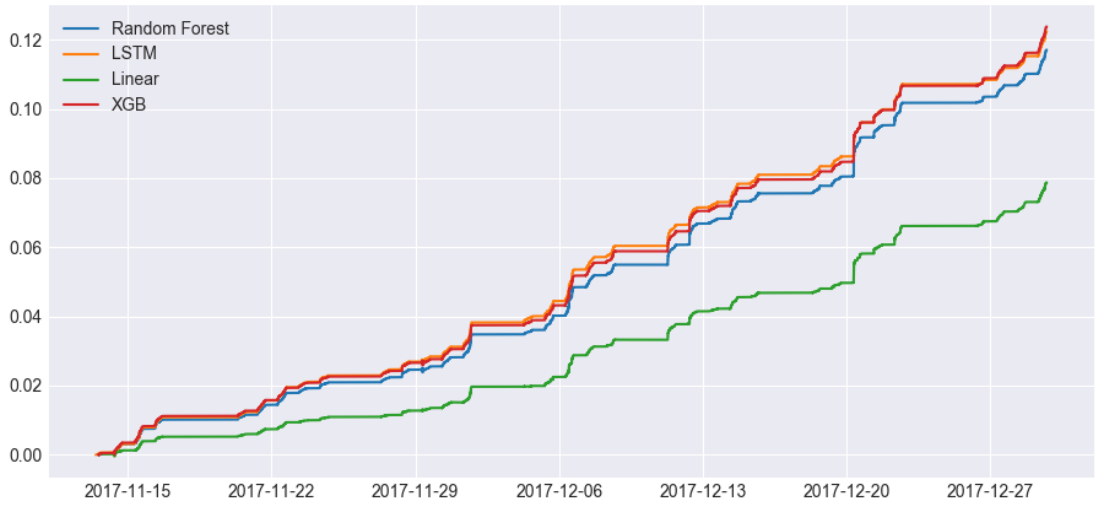


Figure 12: R^2-gain for EUR/SWE 2 ticks predictions

LSTM's outperformance is further confirmed by looking at R^2 grouped by different conditions. If we, for instance, consider the daily R^2, LSTM has the highest score at least 85% of the time for all currency pairs but EUR/SWE where it is best 45% of the time. We also measured the models' performance conditional on the size of spreads, volatility and exchange price discrepancies and in all cases, LSTM looked very strong.

In the 50 ticks predictions task, the only clear conclusion we could make was that MARS and XGB generally performed poorly. OLS, RF and LSTM tended to perform very similarly and only differ in the case of large market movements. This is illustrated in Figure 14 where the slope of the curves only seem to differ at the jumps. These jumps usually happen due to economic news events and getting them right is very important for the final R^2 score. It is worth noting that large movements in the mid price play a more significant role in the 50 ticks prediction task, as a large one-tick movement will result in a large value of 50 consecutive target variables. For 2 ticks predictions, a large increment is only reflected in 2 target variables and it is hence a relatively smaller part of the data set which consists of extreme-valued target variables. For all models, there seem to be a lot of randomness involved in getting the jumps right, and it is hence important to have a large testing set to avoid making conclusions based on pure randomness.

A good example of the importance of getting the large movements right is EUR/SWE. If we just consider the R^2 numbers in Table 3, it looks like RF is performing much better than LSTM but if we look at Figure 15 it is clear that it is just being outperformed on two economic events, i.e. if we removed 100 observations out of the 350,000 in the dataset, we would probably not see any particular difference between the two models. In fact, the LSTM model shows very good

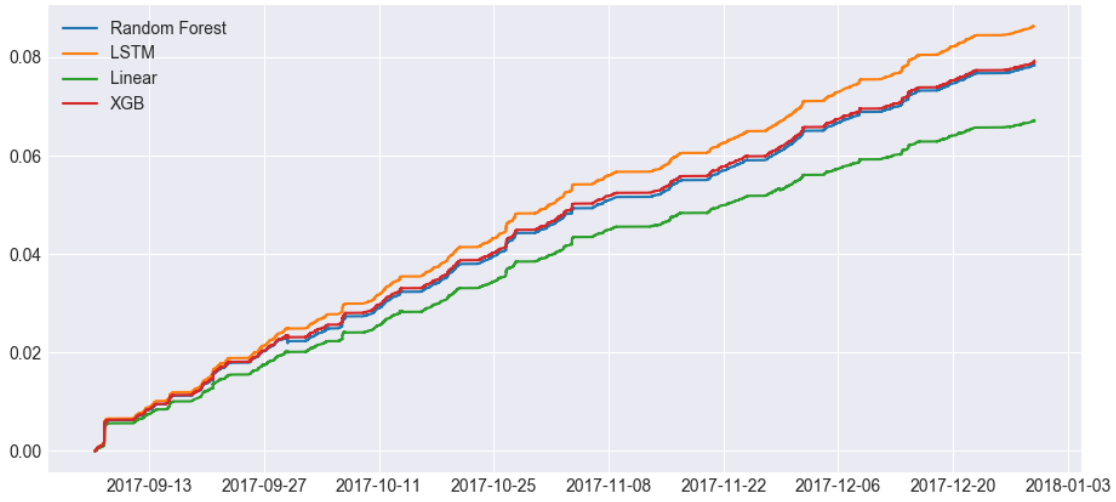


Figure 13: R^2-gain for EUR/USD 2 ticks predictions

performance in most market conditions, as it generally outperforms RF besides the times where the spread on venue A is very high which is shown in Figure 16. A high value of the venue A spread can be seen as an indication of unusual market conditions, and hence it seems like LSTM only loses out when such are present. Due to the testing set only containing two economic events, it is impossible to tell whether the LSTM generally is incapable of predicting economic events for EUR/SWE or it just got 'unlucky'.

When we consider the other two currency pairs EUR/CHF and GBP/USD, the conclusions are similar to the ones of EUR/USD and EUR/SWE; the models mostly just differ on the economic events, and there can be drawn no certain conclusions on which model is best. Only when considering EUR/SWE, do we see signs of machine learning methods outperforming the OLS model. As the dataset for EUR/SWE is by far the smallest and results generally depend a lot on economic events, we can though not draw any strong conclusions from this.

If we consider factors such as model interpretation and computation time (see Table 4), we would though favour OLS, as factors are much better for OLS than for the machine learning methods. We saw indications of LSTM performing well in some cases, but with the high computation time (see Table 4) and low interpretation of predictions, it is not justifiable to recommend it over OLS.

It is likely that the combination of a high noise to signal ratio (causing overfitting) and a concentration in feature importances (making high-level interactions lees relevant) is what keeps the machine learning models from consistently outperforming the linear regression, as it is the case when we consider 2 ticks predictions.

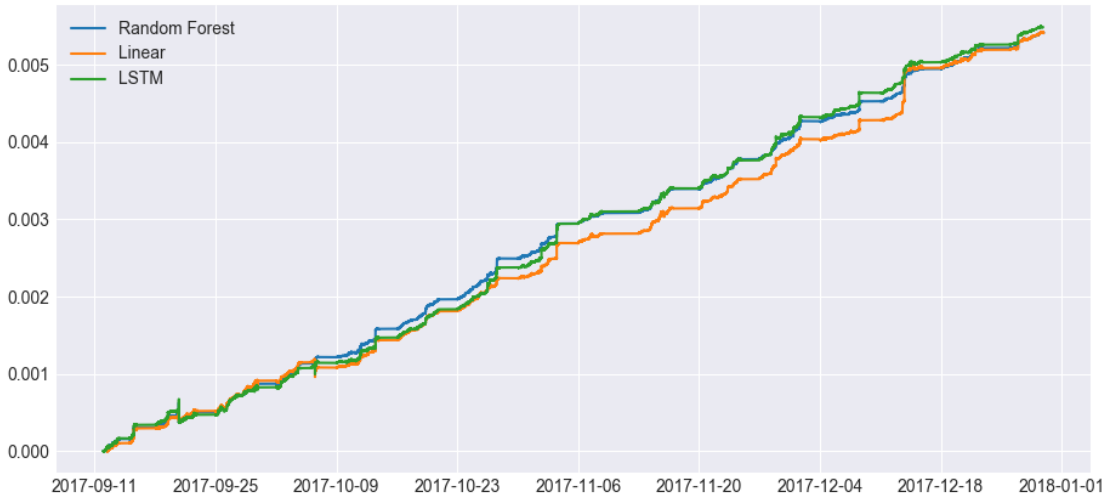


Figure 14: R^2-gain for EUR/USD 50 ticks predictions

	EUR/USD	**GBP/USD**	**EUR/CHF**	**EUR/SWE**
OLS	0.2	0.1	0.1	0.05
XGB	50	28	23	12
RF	11	6	4	2
MARS	28	17	11	6
LSTM	876	512	315	145

Table 4: Computation time in minutes for 50 ticks predictions. Computations were carried out on a 32 core, 300 GB RAM computer with no GPU.

4.3 Further Research

An interesting extension of this thesis would be to make short-term predictions with machine learning models, e.g. LSTM, and then use these predictions as an input for a more simple model, e.g. OLS. An accurate short-term prediction could prove to be a valuable input for a longer prediction horizon, as it actually makes up a large part of the long term prediction. For instance, substituting our 50 ticks predictions with the 2 ticks predictions yielded 60% of the 50 ticks prediction task's R^2 on average. By only making short-term predictions with the machine learning methods, overfitting would be much less of an issue and their capability of modelling high-level interactions could be used. As machine learning methods showed to perform much better than OLS for the short-range predictions, they must be picking up signals that the OLS does not. By using these short-range predictions in the OLS model, we would convert these signals into a single directional signal which could prove to be useful for the OLS model.

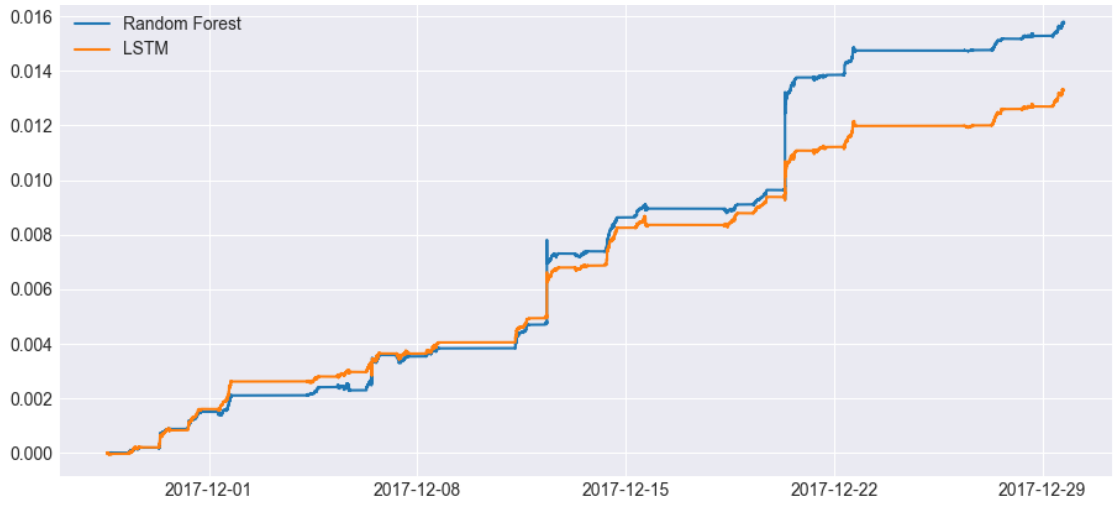


Figure 15: R^2-gain for EUR/SWE 50 ticks predictions

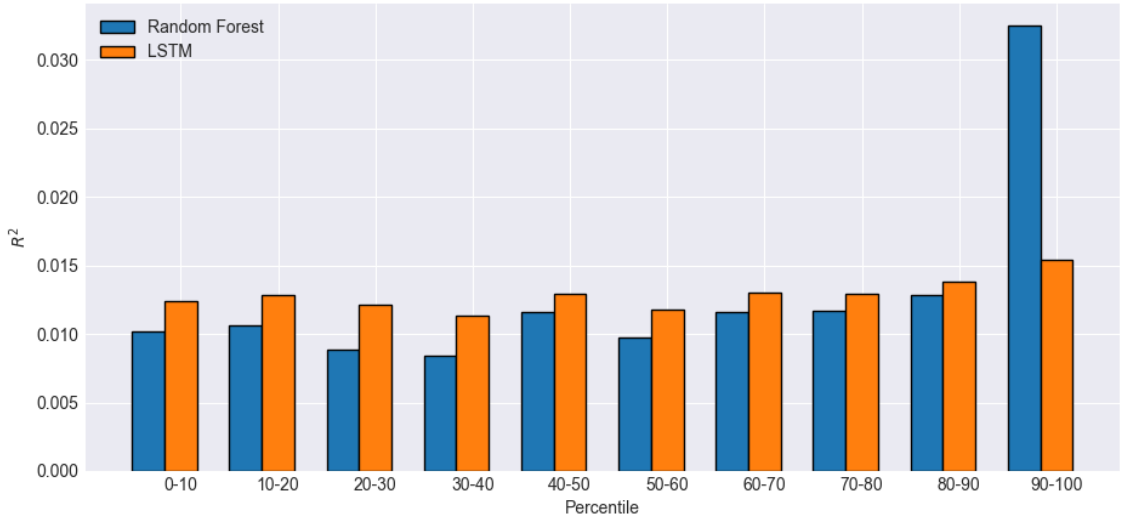


Figure 16: R^2 grouped by size of spread on venue A

Conclusion

Throughout Section 3, we saw the theoretical capabilities of the models increase, as we considered more and more complex models. On the other hand, interpretation went in the opposite direction, and in the final LSTM model, there was hardly any left. Also, computation time increased significantly as shown in Table 4. In Section 4, we investigated whether the increase in model complexity resulted in models capable of producing outstanding predictions or they are simply not suitable for high frequency FX mid price predictions. We did this by considering both a short prediction horizon (2 ticks) and a longer one (50 ticks).

In the case of 2 ticks predictions, there was a clear distinction between the performance of machine learning models and the performance linear models. Based on R^2 scores, predictions made with random forest, XGB and LSTM were significantly better than the ones made with linear models. Especially LSTM proved suitable for the task, as it consistently outperformed the other models for all currency pairs but EUR/SWE where it was equally good to the XGB model.

For 50 ticks predictions, we could identify MARS and XGB as the worst performing models but it was difficult to rank OLS, random forest and LSTM based on predictive performance. The predictions made by these three models tended to be very similar during normal market conditions but could differ significantly during economic news events where price movements are large. There was though no clear pattern of which models were best at getting the predictions right during these periods of time, meaning that we are unable to rank them by their R^2 numbers. If we also consider computation time and interpretability of models as factors as well, we will though argue that choosing a linear model is the best option for this prediction task.

We believe that the difference in model performance in the two prediction tasks is mainly caused by an increased noise to signal ratio and feature importance concentration in the 50 ticks predictions task. When we combine a lot of noise in the data with a high amount model parameters, we easily overfit the data and are forced to reduce model complexity significantly, meaning that the model capabilities cannot be used in full. Additionally, the increased feature importance concentration means that the machine learning models' capability of modelling high-level interactions may become less relevant.

References

Breiman, Leo. Bagging predictors. *Machine Learning*, 24(2):123–140, August 1996. ISSN 0885-6125.

Breiman, Leo. Random forests. *Machine Learning*, 45(1):5–32, October 2001a. ISSN 0885-6125.

Breiman, Leo. Manual on setting up, using and understanding random forests. *Machine Learning*, 45:5–32, 2001b.

Chen, G. A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation. *ArXiv e-prints*, October 2016.

Efron, Bradley, Hastie, Trevor, Johnstone, Iain, and Tibshirani, Robert. Least angle regression. *Annals of Statistics*, 32:407–499, 2004.

Friedman, Jerome H. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1): 1–67, 1991. ISSN 00905364.

Friedman, Jerome H. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001.

Friedman, Jerome H. Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38(4):367–378, February 2002. ISSN 0167-9473.

Gal, Yarin and Ghahramani, Zoubin. A theoretically grounded application of dropout in recurrent neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pp. 1027–1035. Curran Associates Inc., 2016.

Graves, Alex. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, 2011.

Graves, Alex, Jaitly, Navdeep, and rahman Mohamed, Abdel. Hybrid speech recognition with deep bidirectional lstm. In *ASRU*. IEEE, 2013.

Hastie, Tibshirani, Friedman. *Elements of Statistical Learning*. Springer, 2008.

Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In Kremer and Kolen (eds.), *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, November 1997. ISSN 0899-7667.

Hoerl, Arthur E. and Kennard, Robert W. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):80–86, February 1970.

Hornik, Kurt. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4 (2):251–257, March 1991. ISSN 0893-6080.

Kingma, D. P. and Ba, J. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, December 2014.

Louppe, Gilles. *Understanding Random Forests: From Theory to Practice*. PhD thesis, University of Lige, 2014.

Olah, Christopher. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015. Accessed: 2018-09-05.

Schäfer, Anton Maximilian and Zimmermann, Hans Georg. Recurrent neural networks are universal approximators. In *Proceedings of the 16th International Conference on Artificial Neural Networks - Volume Part I*, ICANN'06, pp. 632–640. Springer-Verlag, 2006.

Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15 (1):1929–1958, January 2014. ISSN 1532-4435.

Tibshirani, R. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society (Series B)*, 58:267–288, 1996.

Wilder, K., Geman, D., and Amit, Y. Joint induction of shape features and tree classifiers. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 19:1300–1305, 11 1997. ISSN 0162-8828.

Wolpert, David H. and Macready, William G. An efficient method to estimate bagging's generalization error. *Machine Learning*, 35(1):41–55, 1999.

Zhang, Heping. Maximal correlation and adaptive. *Technometrics*, 36(2):196–201, 1994.

Zou, Hui and Hastie, Trevor. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, Series B*, 67:301–320, 2005.