Classification-based Prediction of Gadget Time Series Using Machine Learning

by Sahil Chadha

Submission date: 10-Sep-2019 02:12AM (UTC+0100)
Submission ID: 110620089
File name: CHADHA_SAHIL_01105677.pdf (2.87M)
Word count: 15879
Character count: 77367

Classification-based Prediction of Gadget Time Series Using Machine Learning

by

Sahil Chadha (CID: 01105677)

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, 2018-2019

Declaration

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

Signature and date:

In the loving memory of my grandmother,

Ms. Sudha Chadha

Acknowledgements

I would like to thank Deutsche Bank's Electronic Rates Trading team for providing me with the necessary data and infrastructure for pursuing this project. I am extremely grateful to my supervisor Brian Gabillet for guiding me throughout this project.

Also, I would like to thank my supervisor at Imperial College London, Prof. Damiano Brigo, for his important remarks and suggestions whenever needed. I am also grateful to the department of Mathematics and Finance for facilitating this project with Deutsche bank.

I would also like to thank all my family back home. I would like to thank my father for his financial support throughout my education and for being the best life coach. I would also like to thank my mother for very patiently teaching me mathematics, and my sister who has always served as the perfect role model. I am forever grateful to my friends, Sarthak and Nischay for always putting a smile to my face in times of trials and turbulence. Finally, I would like to thank my girlfriend Fer Derbez for always pushing me to achieve higher, even from miles away. This wouldn't have been possible without their support.

Abstract

The goal of this thesis is to use a classification-based approach to predict movements of an Interest Rates security called Gadget. We will focus on four models with increasing complexity, ranging from less than a hundred trainable parameters for Multinomial Logistic Regression, to more than ten thousand parameters as in the case of Long Short Term Memory networks. We will also review some recent research in machine learning about optimisers to achieve faster convergence. Further, we will discuss the results for short-term (2 minutes ahead) and long-term (10 minutes ahead) predictions using appropriate metrics. We will finish by implementing a simple trading strategy and study the profitability of alpha generated by each model. The research from this thesis can potentially be used for market-making purposes by banks, hedge-funds and liquidity providers alike.

Contents

1 Introduction

Until a few years ago, it was believed that financial markets were completely random and any changes in market regimes were impossible to predict. While the extent of noise in financial markets data is relatively high, many recent studies have established trading opportunities in a variety of markets. According to Bank for International Settlements' research [1], today nearly three-quarters of trading in certain asset classes such as FX and Equities is driven by algorithms, many of which use sophisticated machine learning techniques. One of the markets in which trading is still very much voice-driven is the Interest Rates derivatives market. The goal of this thesis is to study the scope of viability of using machine learning algorithms for an Interest Rates product only available in an inter-bank market, called Gadget. The research from this thesis can potentially be used for market-making purposes by banks, hedge-funds and liquidity providers alike.

The aim is to classify the short-term (2 minutes) and long-term (10 minutes) log-returns of 5Y Gadget time series into 5 categories, which will be appropriately defined. A very common approach is to predict the level of the asset log return using techniques such as vector auto-regression, however, Leung et al. [2] provide evidence that classification based methods outperform level based methods in the prediction of the direction of asset movement and trading returns maximization. Keeping this view in mind, all our models will be introduced for the purpose of classification. A similar approach applied by Dixon et al. in [3] using Deep Neural Networks yielded impressive results for predicting 3-class movements in 43 commodities and FX futures.

We will start with the most basic model, Multinomial Logistic Regression which was introduced in statistical literature by J.Engal [4] using a Ridge penalty for regularisation (by Hoerl & Kennard [5]). MLR method is fast, easy to train and provides interpretability of the model parameters. However, financial markets are proved to exhibit non-linear behaviour (see [6]), and the biggest problem with this model is its inability to address non-linearity in data. We will explore alternative models to address this problem.

We will look at a famous ensemble learning technique called Random Forests introduced by Leo Breiman [7] based on the idea of bootstrap aggregation and random feature selection. The idea behind such an approach is simple but extremely effective: a congregation of relatively uncorrelated randomised decision trees outperforms any of the individual randomised trees in the congregation. Effectively, Random Forests can be tuned to minimise the overall variance of single CART decision trees. The element of randomness in these models was a success and served as an inspiration for using randomness in deep learning.

The space of functions represented by Random Forests is relatively rich but it can be further extended by considering a Deep Neural Network, which from Universal Approximation Theorem [8] can theoretically approximate a wide variety of interesting functions. Even though applications of artificial neural networks to time series were well documented long ago (see Kaastra and Boyd [9]), many problems with convergence and over-fitting were not thoroughly researched. With higher computation speed, came the ability to feasibly implement neural networks with millions of parameters to train. The problem of overfitting was solved by applying randomness to DNNs through 'dropouts' introduced by Srivastava et al. [10].

While feedforward neural networks can theoretically represent an extremely rich space of functions, their inability to comprehend sequential data gave rise to Recurrent Neural Networks and their popular extension Long Short Term Memory units introduced by Hochreiter & Schmidhuber [11]. These networks don't only exhibit temporal dynamic behaviour but also fix the infamous vanishing gradient problem. LSTMs have a forget gate which controls the memory of each cell state over time. Such an invention was revolutionary in processing complex time series in speech recognition and language modelling [12] and has been regarded the most commercial AI achievement. We will explore this 'black box' model's use case in finance, where it is extremely important to be able to explain the reason for decisions.

Current research in machine learning is very much focused on developing gradient descent optimisers for faster convergence. We will discuss the problem with two common gradient descent optimisers, Stochastic Gradient Descent [13] and Adam [14], and also apply the latest development, Rectified-Adam [15] to our dataset.

The out-of-sample performance of all the models will be then evaluated using relevant performance metrics, i.e, Precision, Recall and F-score. Furthermore, we shall back-test a simple trading strategy using our models on 2-months data, and compare its profitability against the performance of a random predictor to verify the alpha generated.

2 Basics

2.1 What's 'Gadget'?

Our prediction target throughout this thesis will be the 5-year Gadget, an exclusive Interest Rates product only available at the i-Swaps interbank market. It is a 'proxy' of the asset swap spread of BOBL, which are standardised futures contracts based on a basket of medium-term debt (4.5-5.5 year maturities) issued by the German Federal Government. BOBL futures contracts trade under the symbol FGBM on the Eurex exchange and are settled by delivery. These contracts mature quarterly in March, June, September, and December. At any given point of time, the implied yield of a BOBL future is determined by the yield of the cheapest-to-deliver debt contract in the basket. Since the cheapest-to-deliver bond in the basket is dynamic, the asset swap spread can't be directly calculated. Hence to maintain the product's simplicity, the 'proxy' asset swap spread, which is Gadget, is calculated by taking the floating rate as the 6 month fixed leg of the 5y EURIBOR. Gadget is a difference between two interest rates and so the unit of measurement is basis points. In this thesis, the focus of our analysis will be on modelling this financial instrument empirically. Hence, we will be working in 'p-measure'.

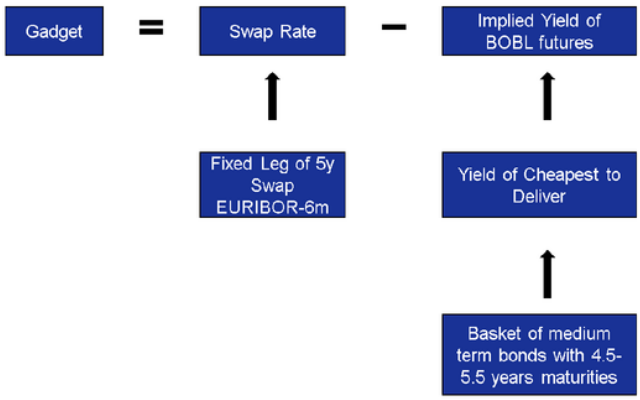

Figure 1: Gadget (5Y) Explanation

2.2 Why use 10 year Gadget?

The justification for using 10 year Gadget time series for predicting the 5-year Gadget is that both financial instruments are based on German interest rate futures of different maturities and hence driven by the same macroeconomic factors. From the following graph, prima-facie it can be said that the two time series move together.

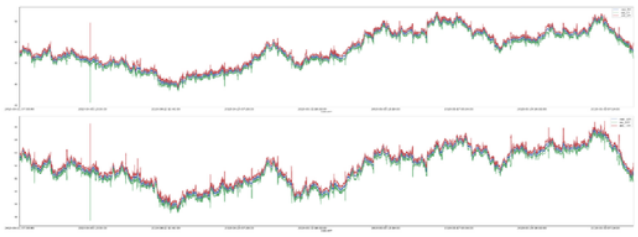

Figure 2: Gadget 5Y (top) & Gadget 10Y (bottom)

This becomes clearer when we look at the correlation between the log returns of the two series on different timescales. As the timescales gets wider, there is a significant correlation between the two.

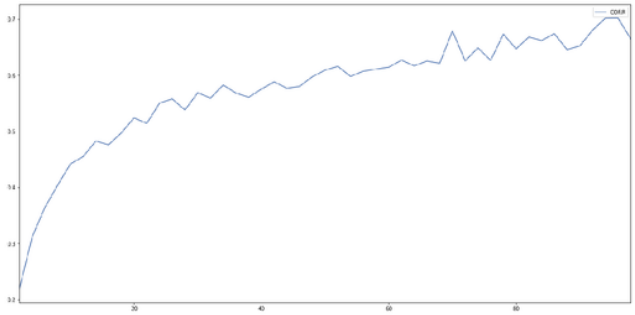

Figure 3: Increasing correlation between Gadget 5Y and Gadget 10Y log returns w.r.t time

2.3 Data Preparation

The precursor to any effective analysis is the data preparation stage. The first step was to concatenate the two asynchronous time series. Just by comparing the lengths of the two time series it was observed that the 10-year Gadget ticks more frequently than the 5-year Gadget (although the latter is more volatile than the former). Also, given that the prediction target is 5-year Gadget, the 10-year Gadget was adapted to the time index of the 5-year using forward filling technique.

Once the time indices were unified, the concatenated tick data-frame was re-sampled to a 2-minute timescale, such that for any given two-minute window, the closing mid of the window was taken to be the mid at that time point. This timescale was chosen keeping in mind our prediction horizon (2 minutes and 10 minutes) and to ensure minimal loss of data points. Also, the decision of working in time units instead of ticks was made keeping in mind the nature of this market.

2.4 Feature Generation

We need to define features that capture the appropriate amount of information about historical mid movements. For this purpose, we use the Exponential Moving Average Lagged and Exponential Moving Average Cross indicators which are both based on Exponential Moving Average (EMA) indicator of the given time series. Mathematically, for some $\alpha \in (0,1)$ and $t \in \{1,...,T\}$, EMA : $\mathbf{N} \times (0,1) \to \mathbf{R}^+$ is defined as:

$$EMA(0,\alpha) = MID(0)$$
$$EMA(t,\alpha) = \alpha * MID(t) + (1-\alpha) * EMA(t-1,\alpha)$$

The parameter α controls the memory in the time series as each data point is now a linear combination of the present and the past mids, with decreasing weights. Clearly, higher α gives more weight to the present data point and implies less memory in the time series. Using this function, we shall now define two types of technical indicators:

- **Exponential Moving Average - Cross Indicators**: For $0 < \alpha_1 < \alpha_2 < 1$ and $t \in \{0,...,T\}$,
$$EMA_{CROSS}(t,\alpha_1,\alpha_2) = EMA(t,\alpha_2) - EMA(t,\alpha_1)$$
- **Exponential Moving Average - Lagged Indicators**: For $\alpha \in (0,1)$ and $i \in \{lag,..,T\}$
$$EMA_{LAGGED}(t,\alpha) = EMA(t,\alpha) - EMA(t-lag,\alpha)$$

Essentially, both the indicators are linear combinations of the data points of the time series and capture past fluctuations of the mid. For the purpose of this thesis, at each time point, to predict the future log return we consider a list of α's which was arbitrarily taken to be $\{0.1, 0.3, 0.5, 0.7, 0.9\}$ and list of lags $\{0, 10, ..., 40, 50\}$, where each lag is in units of 2 minutes. This way we are able to generate a large number of features that might contain some information about the future.

2.5 Data Splitting, Denoising and Scaling

At this point, we split our prepared dataset into training and testing sets. Training and validation were performed for the months of April and May, and testing on June and July.

We only alter the target variable (future log returns) in the training set by fitting a standard scaler to the target variable $\mathbf{y}$, such that:

$$\mathbf{y}_i \mapsto \frac{\mathbf{y}_i - \mu}{\sigma}$$

for $i \in \{0,...,T\}$ where μ and σ are the mean and variance of $\mathbf{y}$. To denoise the peaks, we clip all the entries greater than 3 (so more than 3 standard deviations away from the mean) to 3. We then

transform the log returns back to the original scale by applying an inverse of the transformation defined above. The testing set target is kept untouched throughout this procedure to avoid peeking into the future.

Further, many of the log returns on the training set were found to be equal to 0. We remove 20% of such data points (randomly chosen) to make the distribution of log returns on the training set smoother.

2.6 Data Labelling

Traditionally, researchers go for 3-label classification of movement in financial instruments but for the purpose of this thesis, we considered a 5-label classification. More number of classification bins would mean clearer distinction between future movements and hence allow for more effective decision making for the investor. We label our target variable into 5 labels which are:

- 1 or '- -' : extreme downward, bottom 10 percentile of the training set log returns
- 2 or '-': gentle downward, 10 to 30 percentile of the training set log returns
- 3 or '$\sim$' : not significant, 30 to 70 percentile of the training set log returns
- 4 or '+': gentle upward, 70 to 90 percentile of the training set log returns
- 5 or '++': extreme upward, top 10 percentile of the training set log returns

It is important to note that by doing this, we are introducing a class imbalance in our data set. Labels 1 and 5 will be under represented in our dataset and label 3 will be the most represented. This is desirable because we want extreme movements to be more rare and kept exclusive, but this can also disrupt our machine learning models. We will address this problem in Section 3 when we discuss the models.

3 Models

Now that we have defined the setup of our data, we can go onto discussing the theory behind each model that was implemented.

First we will introduce some notation which will be used throughout this section unless stated otherwise. We can assume that we are given a dataset $\mathcal{D} := \{(\mathbf{x}_t, y_t), \text{t} = 0,...,\text{T}\}$, where $\mathbf{x}_t$ and y_t are our input and output variables respectively. It is assumed that $\mathbf{x}_t$ belongs to $\mathbb{R}^p$, where p is the number of features. Also, since we are classifying into 5 categories, y_t will be a label in the label space $\mathcal{Y} := \{1, 2, 3, 4, 5\}$. Hence, each observation from $\mathcal{D}$ will be a realisation of the stochastic vector (X_t, Y_t) belonging to $\mathbb{R}^p \times \mathcal{Y}$. The model prediction at time t is denoted by $\hat{y}_t$, which also belongs to $\mathcal{Y}$.

Further we define the vector representation or 'one-hot encoding' of the label y_t as $v_t \in [0, 1]^5$ (realisation of a stochastic vector Υ_t), such that $v_{t,i} = 1$ if $i = y_t$ and $v_{t,i} = 0$ otherwise. For example, OHE for label $y_t = 1$ would be $v_t = [1, 0, 0, 0, 0]$. This way we are able to get a vector representation for each label. Usually, machine learning models will output a discrete probability estimate $\hat{v}_t \in [0, 1]^5$ over the 5 categories, such that, $\sum_{k=1}^{5} \hat{v}_{t,k} = 1$. For a model prediction $\hat{v}_t$, the label y_t can be simply taken as $\arg\max_{k \in \{1,..,5\}} \hat{v}_{t,k}$.

The goal of all machine learning models in this thesis is to approximate a non-deterministic function $f_{\mathcal{L}} : \mathbb{R}^p \to \mathcal{Y}$, such that the Expected Prediction Error (EPE) is minimised, where,

$$EPE_{Y|X=x,\mathcal{L}}[f_{\mathcal{L}}] = \mathbb{E}_{Y|X=x,\mathcal{L}}[L(Y, f_{\mathcal{L}}(X))],$$

Here $\mathcal{L} \subset \mathcal{D}$ is a carefully chosen 'learning/training set' on which $f_{\mathcal{L}}$ is constructed and $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is called a loss/cost function for the machine learning problem. Alternatively, the loss function can also be defined on the probability estimates, such that, $L : [0, 1]^5 \times [0, 1]^5 \to \mathbb{R}$

Bias-Variance Decomposition for Multi-Classification Problem: Before we go on to define the models, it would be useful to decompose the EPE to find its components. While such a decomposition is widely discussed for regression problems, the expression for classification problems is more complicated. For simplicity consider a zero-one loss function, for the true label y_t and predicted label $\hat{y}_t$ as:

$$L(\hat{y}_t, y_t) := \begin{cases} 0 & \text{if } \hat{y}_t = y_t \\ 1 & \text{otherwise,} \end{cases}$$

Since $f_{\mathcal{L}}$ is a non-deterministic function, we define $\hat{y}_t^*$ for an example $\mathbf{x}_t$ to be the 'optimal prediction' which minimises $EPE[f_{\mathcal{L}}]$ for that example. Then the 'optimal classifier' $f_{\mathcal{B}} : \mathbb{R}^p \to \mathcal{Y}$ is the

model for which $f_{\mathcal{B}}(\mathbf{x}_t) = \hat{y}_t^*$ for every t. We call this the Bayes classifier, and its loss the Bayes rate.

Also, define the 'main prediction' to be the value $y_{\mathcal{B}}$ whose average loss relative to all the predictions in $\mathcal{L}$ is minimum, i.e, it is the Bayes estimator. Under the assumed loss function, this would be the mode (the most frequent prediction) of the predictions. With this setup, we will reproduce the results from Domingos [16], where he introduced the idea of a unified bias-variance decomposition for several loss functions. For a data sample $(\mathbf{x}, y) \in (X_t, Y_t)$ (we omit the subscript t for now for convenience), in a multi-classification problem like ours, consider the following:

$$\begin{aligned}
\mathbb{E}_{Y|X=x}[L(f_{\mathcal{L}}(\mathbf{x}), Y)] &= \mathbb{E}_{Y|X=x}[1(f_{\mathcal{L}}(\mathbf{x}) \neq Y)] \\
&= \mathbb{P}(f_{\mathcal{L}}(\mathbf{x}) \neq Y) \\
&= 1 - \mathbb{P}(f_{\mathcal{L}}(\mathbf{x}) = Y) \\
&= 1 - \mathbb{P}(f_{\mathcal{L}}(\mathbf{x}) = Y | f_{\mathcal{B}}(\mathbf{x}) \neq Y)\mathbb{P}(f_{\mathcal{B}}(\mathbf{x}) \neq Y) \\
&= \mathbb{I}(f_{\mathcal{L}}(\mathbf{x}) \neq f_{\mathcal{B}}(\mathbf{x})) + c_0 \mathbb{E}_{Y|X=x}[\mathbb{I}(Y \neq f_{\mathcal{B}}(\mathbf{x}))]
\end{aligned}$$

where $c_0 = -\mathbb{P}(f_{\mathcal{L}}(\mathbf{x}) = Y | f_{\mathcal{B}}(\mathbf{x}) \neq Y)$ if $f_{\mathcal{L}}(\mathbf{x}) \neq f_{\mathcal{B}}(\mathbf{x})$ and $c_0 = 1$ otherwise, and $\mathbb{I}(.)$ is the indicator function. Similarly one can show that:

$$\begin{aligned}
\mathbb{E}_{\mathcal{L}}[L(f_{\mathcal{B}}(\mathbf{x}), f_{\mathcal{L}}(\mathbf{x}))] &= L(f_{\mathcal{B}}(\mathbf{x}), \mathbf{y}_{\mathcal{B}}) + c_2 \mathbb{E}_{\mathcal{L}}[L(f_{\mathcal{L}}(\mathbf{x}), \mathbf{y}_{\mathcal{B}})] \\
&= \mathbb{I}(f_{\mathcal{B}}(\mathbf{x}) \neq \mathbf{y}_{\mathcal{B}}) + c_2 \mathbb{E}_{\mathcal{L}}[\mathbb{I}(f_{\mathcal{L}}(\mathbf{x}) \neq \mathbf{y}_{\mathcal{B}})]
\end{aligned}$$

where $c_2 = -\mathbb{P}_{\mathcal{L}}(f_{\mathcal{L}}(\mathbf{x}) = f_{\mathcal{B}}(\mathbf{x}) | f_{\mathcal{L}}(\mathbf{x}) \neq \mathbf{y}_{\mathcal{B}})$ if $f_{\mathcal{L}}(\mathbf{x}) \neq f_{\mathcal{B}}(\mathbf{x})$ and $c_2 = 1$ otherwise.

From the above results, we have:

$$\begin{aligned}
\mathbb{E}_{\mathcal{L}}[\mathbb{E}_{Y|X=x}[L(f_{\mathcal{L}}(\mathbf{x}), Y)]] &= \mathbb{E}_{\mathcal{L}}[\mathbb{E}_{Y|X=x}[\mathbb{I}(f_{\mathcal{L}}(\mathbf{x}) \neq Y)]] \\
&= \mathbb{E}_{\mathcal{L}}[\mathbb{I}(f_{\mathcal{L}}(\mathbf{x}) \neq f_{\mathcal{B}}(\mathbf{x})) + c_0 \mathbb{E}_{Y|X=x}[\mathbb{I}(Y \neq f_{\mathcal{B}}(\mathbf{x}))]] \\
&= \mathbb{E}_{\mathcal{L}}[\mathbb{I}(f_{\mathcal{L}}(\mathbf{x}) \neq f_{\mathcal{B}}(\mathbf{x}))] + \mathbb{E}_{Y|X=x}[\mathbb{I}(Y \neq f_{\mathcal{B}}(\mathbf{x}))]\mathbb{E}_{\mathcal{L}}[c_0] \\
&= \mathbb{I}(f_{\mathcal{B}}(\mathbf{x}) \neq \mathbf{y}_{\mathcal{B}}) + c_2 \mathbb{E}_{\mathcal{L}}[\mathbb{I}(f_{\mathcal{L}}(\mathbf{x}) \neq \mathbf{y}_{\mathcal{B}})] + \mathbb{E}_{Y|X=x}[\mathbb{I}(Y \neq f_{\mathcal{B}}(\mathbf{x}))]\mathbb{E}_{\mathcal{L}}[c_0]
\end{aligned}$$

where $\mathbb{E}_{Y|X=x}[\mathbb{I}(Y \neq f_{\mathcal{B}}(\mathbf{x}))]$ doesn't depend on $\mathcal{L}$ in the third-last step, and c_1 can be calculated as:

$$c_1 = \mathbb{E}_{\mathcal{L}}[c_0] = \mathbb{P}_{\mathcal{L}}(f_{\mathcal{L}}(\mathbf{x}) = f_{\mathcal{B}}(\mathbf{x})) - \mathbb{P}_{\mathcal{L}}(f_{\mathcal{L}}(\mathbf{x}) \neq f_{\mathcal{B}}(\mathbf{x}))\mathbb{P}_{Y|X=x}(f_{\mathcal{L}}(\mathbf{x}) = Y | f_{\mathcal{B}}(\mathbf{x}) \neq Y)$$

So for a multi-classification problem considering a zero-one loss function, the EPE is a linear combination of 3 quantities:

- **Bias**: $L(f_{\mathcal{B}}(\mathbf{x}), \mathbf{y}_{\mathcal{B}})$, the loss incurred by the main prediction relative to the optimal prediction. It measures the systematic loss incurred by a learner, and is training set independent.

- **Variance**: $\mathbb{E}_{\mathcal{L}}[L(\mathbf{y}_{\mathcal{B}}, f_{\mathcal{L}}(\mathbf{x})]$, the average loss incurred by predictions relative to the main prediction. It measures the loss incurred due to fluctuations around the central tendency, and is independent of the true value of the predicted variable.
- **Noise**: $\mathbb{E}_{Y|X=x}[L(Y, f_{\mathcal{B}}(\mathbf{x}))]$, the unavoidable component of the loss, incurred independently of the learning algorithm

More intuitively, bias can be considered as the 'stiffness' of the model, and variance can be considered as its 'flexibility'. Usually bias and variance move in opposite directions, and so machine learning models need to be carefully tuned in order to balance the bias and the variance. The above idea will be particularly useful to build the argument for Random Forests.

Coming back to our classification problem, we would ideally want $f_{\mathcal{L}}$ to estimate a probability distribution over the 5 classes from which the class can be easily inferred (defined at the beginning of this section). It is safe to assume that as we improve on the estimation of this probability distribution for a particular training sample, we will also improve in predicting the correct label. So we would like to define our loss function L on $[0,1]^5 \times [0,1]^5$. For this purpose, the conventional choice of the loss function in literature is 'Categorical Cross Entropy' (CCE), defined as:

$$L(\upsilon_t, \hat{\upsilon}_t) := -\sum_{k=1}^{5} \upsilon_{t,k} \log(\hat{\upsilon}_{t,k})$$

The CCE loss function is a measure of distance between two probability distributions and hence a suitable choice of loss function for this problem [17]. Moreover, it is differentiable unlike the zero-one loss function. Even though no explicit solution exists in current literature for bias-variance decomposition of this loss function, the idea of bias-variance trade-off still holds.

Testing set: If we simply take $\mathcal{L}$ as $\mathcal{D}$, it is guaranteed that we will able to come up with $f_{\mathcal{L}}$ which would give us a very low EPE for all $(\mathbf{x}_t, y_t) \in \mathcal{L}$. This is could be easily achieved by fitting a polynomial of order equal to the number of data points in $\mathcal{L}$. Such a model would get increasingly complex with increase in data points and would capture signal as well as noise in the data. The model would perform poorly on any unseen data point, a condition in machine learning known as 'overfitting'. Hence, as described in section 2, we shall take 2 disjoint partitions of our data, such that, $\mathcal{D} = \mathcal{L} \cup \mathcal{T}$, where $\mathcal{L}$ is the training or learning set and $\mathcal{T}$ is the testing set.

Hyperparameters and Regularisation: Generally, as the model complexity increases, bias decreases but variance increases and so the machine learning objective should also take into account the complexity of the model. More formally, we must first choose a class of functions $\mathcal{F}$ and then within that class find the aforementioned function $f_{\mathcal{L}}$. Here each $\mathcal{F}$ is characterised by some parameters (not to be confused with parameters of $f_{\mathcal{L}}$) which control the complexity of $f_{\mathcal{L}}$, called

'Hyperparameters'. Usually, we optimise the set of hyperparameters by trial and error.
We will now a add a complexity or 'regularisation' term to the original error function (CCE). In totality, the machine learning objective to be minimised then becomes:

$$obj(f_{\mathcal{L}}) = L(f_{\mathcal{L}}) + \mathcal{C}(f_{\mathcal{L}}),$$

where $L(f_{\mathcal{L}}) = \sum_{(\mathbf{x},y)\in\mathcal{L}} L(f_{\mathcal{L}}(\mathbf{x}), y)$, and $\mathcal{C}$ controls the model complexity, also called the 'Regularisation term'. The regularisation term is model-type specific, for example, dropouts in Neural Networks and maximum tree depth in Random Forests.

Cross Validation for Time Series Data: Cross-validation is one of the model selection techniques used to tune hyperparameters in machine learning. The technique aims to find the optimal hyperparameters independent of the choice of the training set and hence to remove any selection bias while doing so. For this purpose, we take disjoint partitions of the learning set called folds, such that, $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2 ... \cup \mathcal{L}_K$. The standard algorithm used for cross validation, called K-Fold Cross Validation, iterates through the folds such that in each iteration, all but one folds are used for training and the remaining one is used for testing. This way, for each combination of hyperparameters, the algorithm computes an aggregated loss (for example, average) over the K-folds, and chooses the combination of hyperparameters for which loss is minimised. However, this approach cannot be directly used for time series data because of the serial order of the data points. For time series data, it is assumed for K-folds that $\mathcal{L}_1 \prec \mathcal{L}_2 ... \prec \mathcal{L}_K$, where $\prec$ denotes the ordering of the folds. Hence, because of these serial dependencies, we cannot train a model on $\mathcal{L}_2 ... \cup \mathcal{L}_K$ and then evaluate it on $\mathcal{L}_1$ (we can't train a model on future data, and then evaluate it on some past data).

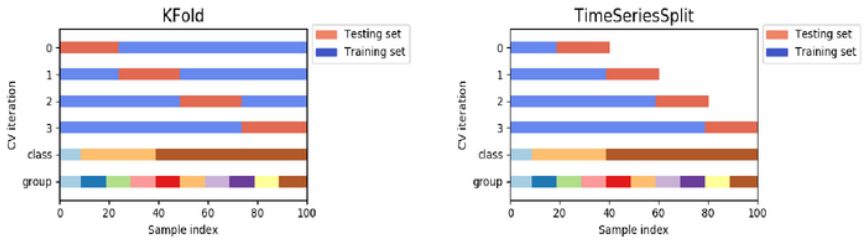

Figure 4: K-Fold Vs. Time Series Split

For the purpose of cross-validation, we shall use a variation of K-Fold, for which we formally define algorithm 1.
In algorithm 1, we assume that $\mathcal{H}$ is the hyperparameter space which is appropriately chosen, we train on $\mathcal{L}^*$ and test on $\mathcal{T}^*$ (different for each k), $f_{\mathcal{L}^*}$ is the model trained on $\mathcal{L}^*$ expressed as a

Algorithm 1 Time Series Cross-Validation

for h `in` $\mathcal{H}$ **do**
  $losses \leftarrow$ `[]`
  for `k in {2,...,K}` **do**
    $\mathcal{L}^* \leftarrow \mathcal{L}_1 \cup ... \cup \mathcal{L}_{k-1}$
    $\mathcal{T}^* \leftarrow \mathcal{L}_k$
    $f_{\mathcal{L}^*} \leftarrow \mathcal{A}(h, \mathcal{L}^*)$
    $losses \leftarrow losses + [L(f_{\mathcal{L}^*}(\mathcal{T}^*))]$
  end for
  $grid(h) \leftarrow max(losses)$
end for
$h_{opt} \leftarrow \arg\min_{h \in \mathcal{H}} grid$

function ($\mathcal{A}$, some algorithm) of the hyperparameters h and $\mathcal{L}^*$. With a slight abuse of notation, we define $L(f_{\mathcal{L}^*}(\mathcal{T}^*))$ to be the loss of a model trained on $\mathcal{L}^*$ and evaluated on $\mathcal{T}^*$. Furthermore, we took the aggregated loss function over K folds to be the maximum of the losses (the worst case scenario) and not average of the losses because in practice, even if the mean loss for a particular combination of hyperparameters is low, the standard error of h defined as:

$$SE(h) = \sqrt{\frac{Var(losses(h))}{K}}$$

could be high, leading to a bad choice of hyperparameters. Here $losses(h)$ is the array of losses for hyperparameter set h over the K folds.

Addressing Class Imbalance: As stated in Section 2, when creating the boundaries for classification, we take unequal bin sizes. This creates a class imbalance which might lead to biased models. The way we address this problem in this thesis is by taking appropriate class weights. Since class '1' and '5' are top 10 and bottom 10 percentiles, classes '2' and '4' each are 20 and class '3' is middle 40 percentile of the training set, misclassification in each class should be penalised in inverse proportions to the size of the class. A simple calculation would dictate class weights of 4 for '1' and '5', 2 for '2' and '4' and 1 for '3'. By doing this, during model training, a misclassification of classes '1' and '5' is penalised 4x more than that for '3' because they are considered as rarer data points. So all throughout our analysis, we are considering a weighted version of CCE. In theory, we will use the variable ω for the array of weights and take a weighted loss function to be L_ω.

Having discussed some background about the general methodology, we will now discuss some methods of constructing f. We will start with a linear model, and then escalate the complexity to training over 10,000 parameters. All models were implemented in Python. For Multinomial Logistic

Regression and Random Forests we used Scikit Learn's [18] implementations *LogisticRegression* and *RandomForestClassifier*, and for Deep Neural Networks and LSTMs we used *Keras*. Note that in this thesis, Deep Neural Network classifier refers to a deep neural network with a feedforward architecture.

3.1 Multinomial Logistic Regression

We will start with a basic model which extends the idea of Logistic Regression to multiple classes, and is also known as Softmax Regression. Considering the setup as defined earlier, the learning function $f_{\mathcal{L}}$ for Multinomial Logistic Regression can be defined by the following simple steps:

- First, consider the affine transformation $\mathbf{z}_t = \theta \times \mathbf{x}_t + b$, where, $\theta \in \mathbb{R}^{5\times p}$ and $b \in \mathbb{R}^5$ so that $\mathbf{z}_t \in \mathbb{R}^5$. Here θ is the weight matrix and b is called the bias of the model.
- To this affine transformation, we apply the 'softmax' function, which is defined as:

$$\text{softmax} : \mathbb{R}^5 \to \mathbb{R}^5$$
$$\text{softmax}_i(\mathbf{z}_t) = \frac{e^{\mathbf{z}_{t,i}}}{\sum_{j=1}^{5} e^{\mathbf{z}_{t,j}}},$$

  where $i \in \{1, 2, 3, 4, 5\}$. This way softmax function transforms vector $\mathbf{z}_t$ into a probability distribution vector, which is the model's prediction $\hat{v}_t$.
- More concisely, we can define the learning function as $\hat{v}_t = f_{\mathcal{L}}(\mathbf{x}_t) = \text{softmax}(\theta\mathbf{x}_t + b)$ and we can now compute the associated Categorical Cross Entropy loss with this prediction, given by $L_\omega(v_t, \hat{v}_\mathbf{t})$. From above, for a given v_t, we can clearly express the loss just as a function of θ and b.

The above procedure is called 'Forward Propagation' where we start with some random parameters θ and b, and compute the associated loss for them. Now the question remains to choose the θ and b which minimise this loss. For this, we use an approach called 'Gradient Descent'. Due to the differentiability of the loss function, we can find its gradient with respect to θ and b, and update both in the direction of minimisation of the loss. Assume for the following that the data is scaled and so $b = \mathbf{0}$, so that the update rule for θ is given by:

$$\theta_{i,j} \mapsto \theta_{i,j} - \eta \frac{\partial L_\omega(\theta)}{\partial \theta_{i,j}},$$

where $i \in \{1, 2, 3, 4, 5\}$, $j \in \{1, ..., p\}$ and η is the step size or learning rate.

The next step is to compute the gradient, which is done using 'Backward Propagation' based on simple chain rule. For a single sample $(\mathbf{x}, y) \in \mathcal{L}$ and v the OHE of y,

$$\frac{\partial L_\omega}{\partial \theta_{i,j}} = \sum_{k=1}^{5} \frac{\partial L_\omega}{\partial \hat{v}_k} \frac{\partial \hat{v}_k}{\partial \mathbf{z}_i} \frac{\partial \mathbf{z}_i}{\partial \theta_{i,j}},$$

but we know

$$\begin{aligned}\frac{\partial \mathbf{z}_i}{\partial \theta_{i,j}} &= \mathbf{x}_j \\ \frac{\partial L_\omega}{\partial \hat{v}_k} &= -\frac{\omega_k v_k}{\hat{v}_k} \\ \frac{\partial \hat{v}_k}{\partial \mathbf{z}_i} &= \frac{\partial}{\partial \mathbf{z}_i}\text{softmax}_k(\mathbf{z})\end{aligned}$$

For the last differential, consider 2 cases:

- **Case 1**: i = k

$$\begin{aligned}\frac{\partial \hat{v}_k}{\partial \mathbf{z}_i} = \frac{\partial}{\partial \mathbf{z}_i}\left(\frac{e^{\mathbf{z}_i}}{\sum_{j=1}^5 e^{\mathbf{z}_j}}\right) &= \frac{e^{\mathbf{z}_i}\sum_{j=1}^5 e^{\mathbf{z}_j} - \left(e^{\mathbf{z}_i}\right)^2}{\left(\sum_{j=1}^5 e^{\mathbf{z}_j}\right)^2} \\ &= \frac{e^{\mathbf{z}_i}}{\sum_{j=1}^5 e^{\mathbf{z}_j}} - \left(\frac{e^{\mathbf{z}_i}}{\sum_{j=1}^5 e^{\mathbf{z}_j}}\right)^2 \\ &= \text{softmax}_i(\mathbf{z})(1 - \text{softmax}_i(\mathbf{z})) \\ &= \hat{v}_i(1 - \hat{v}_i)\end{aligned}$$

- **Case 2**: i ≠ k

$$\frac{\partial \hat{v}_k}{\partial \mathbf{z}_i} = \frac{\partial}{\partial \mathbf{z}_i}\left(\frac{e^{\mathbf{z}_k}}{\sum_{j=1}^5 e^{\mathbf{z}_j}}\right) = -\frac{e^{\mathbf{z}_i}e^{\mathbf{z}_k}}{\left(\sum_{j=1}^5 e^{\mathbf{z}_j}\right)^2} = -\hat{v}_i\hat{v}_k$$

Combining the two cases we get,

$$\begin{aligned}\frac{\partial L_\omega}{\partial \hat{v}_k}\frac{\partial \hat{v}_k}{\partial \mathbf{z}_i} &= \begin{cases}\omega_i v_i(1 - \hat{v}_i) & \text{if } i = k \\ -\omega_k v_k \hat{v}_i & \text{otherwise}\end{cases} \\ &= \omega_k v_k(\delta_{i,k} - \hat{v}_i),\end{aligned}$$

where $\delta_{i,k} = 1$ if $i = k$ and 0 otherwise. Therefore, the overall gradient is calculated to be:

$$\begin{aligned}\frac{\partial L_\omega}{\partial \theta_{i,j}} &= \sum_{k=1}^5 \omega_k v_k(\delta_{i,k} - \hat{v}_i)\mathbf{x}_j \\ &= -\left(\sum_{k=1}^5 \omega_k v_k \delta_{i,k} - \sum_{k=1}^5 \omega_k v_k \hat{v}_i\right)\mathbf{x}_j = -\left(\omega_i v_i - \hat{v}_i \sum_{k=1}^5 \omega_k v_k\right)\mathbf{x}_j,\end{aligned}$$

For the entire training sample $\mathcal{L}$, we get

$$\frac{\partial L_\omega}{\partial \theta_{i,j}} = -\sum_{(\mathbf{x},y)\in\mathcal{L}}\left(\omega_i v_i - \hat{v}_i \sum_{k=1}^5 \omega_k v_k\right)\mathbf{x}_j$$

The above calculations would still hold for unscaled data for which we could find the gradient with respect to the bias in a similar fashion.

Multinomial Logistic Regression with Regularisation: The above algorithm was defined for just weighted CCE, without taking into account the regularisation term. For this model, we will control the model complexity by considering 'Ridge' penalty, defined by:

$$\mathcal{C}(\theta) = \frac{1}{\lambda}\|\theta\|_2^2$$

where λ is the inverse of the overall regularisation strength, and $\|.\|_2$ is the L2 norm. The empirical results for tuning this hyperparameter will be discussed in Section 4.

This model is extremely fast with less than a hundred parameters to train. One big drawback is that although softmax function is a non-linear function, the softmax transformation was applied to an affine transformation of the original data, and so MLR belongs to the family of 'Generalised Linear Models'. These models capture individual predictors' (or features') nonlinear impact on the target variables, but do not take into account the nonlinear interactions between different features. To enable this for our analysis, the rest of the models will have this ability.

3.2 Random Forests Classifier

Random Forests is based on ensemble learning method, where we train several different models on the same training set $\mathcal{L}$ by introducing random perturbations in the learning procedure, and then combine the prediction of each of the individual models to form the prediction for the ensemble. However, before we discuss this concept, we must look at their building blocks - decision trees.

To understand decision trees, consider that for any classification problem with classes $\{c_1, ..., c_K\}$, the true label space $\mathcal{Y}$ defines a partition of the universe ω, such that,

$$\omega = \omega_{c_1} \cup \omega_{c_2} ... \cup \omega_{c_K}$$

where ω_{c_i} is the subset of ω for which the true label is c_i, and $i \in \{1, ..., K\}$. Similarly, we can consider that f, the classifier to be trained, defines a partition on the input space $\mathcal{X}$ of the learning set to give an approximation $\hat{\mathcal{Y}}$ of the true label space $\mathcal{Y}$. This partition is different from the one above and defined as:

$$\mathcal{X} = \mathcal{X}_{c_1}^f \cup \mathcal{X}_{c_2}^f ... \cup \mathcal{X}_{c_K}^f$$

where $\mathcal{X}_{c_i}^f$ is the subset of $\mathcal{X}$ for which $f(\mathbf{x}) = c_i$ for all $\mathbf{x} \in \mathcal{X}_{c_i}^f$. The aim is to then find a partition which matches as closely as the one defined by the Bayes model $f_{\mathcal{B}}$:

$$\mathcal{L} = \mathcal{L}_{c_1}^{f_{\mathcal{B}}} \cup \mathcal{L}_{c_2}^{f_{\mathcal{B}}} ... \cup \mathcal{L}_{c_K}^{f_{\mathcal{B}}}$$

Based on the above, a decision tree can be defined as a model $f : \mathcal{X} \to \mathcal{Y}$ which approximates the partition of the Bayes model by recursively partitioning the input space $\mathcal{X}$ into a set of terminal

subspaces and then assigning constant prediction values $y \in \mathcal{Y}$ to each of the subspaces.

The procedure for defining an optimal f as above is an NP-complete problem and only has a sub-optimal solution. It has been well defined by Breiman et al. [19]. Crucial to their idea is defining an 'impurity measure' associated with each node n of the tree. For the purpose of this thesis, the impurity measure considered is based on Gini index [21], defined for a node n as:

$$i_G(n) := \sum_{j=1}^{K} p(c_j|n)(1 - p(c_j|n))$$

where $p(c_j|n)$ is the probability of getting the label c_j in the samples under the node n. Essentially, the Gini-based impurity measures how often a randomly chosen object $\mathbf{x} \in \mathcal{L}_n$ would be incorrectly classified if it were randomly labeled by a class $c \in \mathcal{Y}$ according to the distribution $p(c|n)$. A lower impurity measure means a purer node and so better predictions for the subset of the input space falling below that node. Starting from a root node, near-optimal decisions can be grown by iteratively dividing nodes into purer nodes until a stopping criterion is met or the nodes cannot be made purer. This is done by taking a greedy approach, where the split at each node is determined by considering the locally maximum impurity decrease of the resulting child nodes. For a binary split $s \in \mathcal{Q}$ dividing the node n into n_L and n_R, the impurity decrease can be defined as:

$$\Delta i(s, n) := i(n) - p_L i(t_L) - p_R i(t_R)$$

where p_L and p_R are the fraction of total samples under node n under the nodes n_L and n_R respectively.

Once a terminal node n_T (also called leaf node) is reached, it is assigned a constant value. For a multi-classification problem like ours, this constant value is taken to be the majority class for the subset $\mathcal{L}_{n_T}$ of the learning set $\mathcal{L}$. Now the question that remains to answer is that when should we stop splitting the nodes, i.e, how to define the 'stopping criterion' for a decision tree?

In G. Louppe's work on Random Forests [20] Proposition 3.1, one can find the proof that the more we split a terminal node in any way, the smaller the training loss gets. Likewise, one can say that the training loss is minimal when terminal nodes can no longer be divided. In particular, it is equal to zero if the tree can be fully developed, that is if terminal nodes can be divided until they all contain exactly one object from $\mathcal{L}$, in absence of any stopping criterion.

However, such a decision tree would overfit on the learning set, and perform badly on the testing set. It would capture noise along with signal, and fail to generalise from the dataset due to too much information. Hence, we need to introduce hyperparameters which define the stopping criterion for

the decision tree. These hyperparameters would control the depth of the tree, where a very deep tree leads to high variance and overfits and an extremely shallow tree has high bias and underfits (recall bias-variance tradeoff). The most common approaches to prevent overfitting in decision trees are by setting n as a terminal node if:

- it contains less than a certain number of samples N_{min}, or
- it reaches a certain maximum depth d_{max}, or
- any further decrease in node impurity is less than a certain threshold β, or
- there is no such split for which both the child nodes meet a certain requirement N_{leaf} for the minimum number of samples. Hence it guarantees a minimum number of samples in every leaf.

Since all these hyperparameters directly or indirectly affect the depth of the tree and control overfitting, we will aim to optimise only one of them, i.e, N_{leaf}. This, we will do using K-Fold cross validation for time series as defined earlier.

Considering the bias-variance decomposition for a multi-classification problem, it is established from [16], that the tolerance for variance will decrease as the number of classes increases, other things being equal. Thus the ideal setting for the bias-variance trade-off may be more in the direction of low variance in problems with higher classes. Keeping this conclusion in mind, we shall aim to decrease the EPE by decreasing the variance and keep the bias constant or even a little higher. We will now show how taking a collection of uncorrelated randomised decision trees gives a model which supersedes all the individual models in the collection. This aggregated model is the Random Forests model.

Let some random variable ξ be the source of randomness in a decision tree $f_{\mathcal{L},\xi}$ trained on the learning set $\mathcal{L}$. Taking into account the randomness from ξ, the bias-variance decomposition for such a decision-tree under zero-one loss function becomes:

$$\mathbb{E}_{\mathcal{L},\xi}[\mathbb{E}_{Y|X=x}[L(f_{\mathcal{L},\xi}(\mathbf{x}), Y)]] = \text{bias}(\mathbf{x}) + c_1\text{noise}(\mathbf{x}) + c_2\text{var}(\mathbf{x}),$$

where:

$$\begin{aligned}
\text{noise}(\mathbf{x}) &= \mathbb{E}_{Y|X=x}[\mathbb{I}(Y \neq f_{\mathcal{B}}(\mathbf{x}))] \\
\text{bias}(\mathbf{x}) &= \mathbb{I}(f_{\mathcal{B}}(\mathbf{x}) \neq \mathbb{E}_{\mathcal{L},\xi}[f_{\mathcal{L},\xi}(\mathbf{x})]) \\
\text{var}(\mathbf{x}) &= \mathbb{E}_{\mathcal{L},\xi}[\mathbb{I}(f_{\mathcal{L},\xi}(\mathbf{x}) \neq \mathbb{E}_{\mathcal{L},\xi}[f_{\mathcal{L},\xi}(\mathbf{x})])]
\end{aligned}$$

and c_1 and c_2 were both defined in the beginning of this section. This new decision tree has both bias and variance higher than the non-random one and by itself is not an improvement. But what

if we take M of these?

Let $f_{\mathcal{L},\xi_1}, f_{\mathcal{L},\xi_2}, ..., f_{\mathcal{L},\xi_M}$ be a collection of decision trees trained on the same learning set $\mathcal{L}$, using random seeds $\xi_1, \xi_2, ..., \xi_M$ which are independent and identically distributed samples. Assume each decision tree $f_{\mathcal{L},\xi_i}$ gives a probability estimate $\hat{p}_{\mathcal{L},\xi_i}(Y = c|X = \mathbf{x})$ for each class $c \in \{c_1, .., c_J\}$. We also define an ensemble model which is a combination of these M decision trees for a classification problem as

$$\psi_{\mathcal{L},\xi_1,...,\xi_M}(\mathbf{x}) = \arg\max_{c \in Y} \frac{1}{M} \sum_{m=1}^{M} \hat{p}_{\mathcal{L},\xi_m}(Y = c|X = \mathbf{x})$$

This is called the 'soft-voting' rule [22], wherein we first aggregate the probability estimates for each class by taking an average of the estimate from each individual tree, and then the ensemble prediction is taken as the class with the highest probability estimate. We will now prove that the ensemble technique improves these probability estimates, thereby improving the prediction as a whole.

For mean and variance of the prediction of some $\mathbf{x}$ by any individual model, $f_{\mathcal{L},\xi_m}$, we let

$$\mu_{\mathcal{L},\xi_m}(\mathbf{x}) = \mathbb{E}_{\mathcal{L},\xi_m}[f_{\mathcal{L},xi_m}(\mathbf{x})] = \mathbb{E}_{\mathcal{L},\xi_m}[\hat{p}_{\mathcal{L},\xi_m}(Y = c|X = \mathbf{x})]$$
$$\sigma^2_{\mathcal{L},\xi_m}(\mathbf{x}) = \mathbb{V}_{\mathcal{L},\xi_m}[f_{\mathcal{L},xi_m}(\mathbf{x})] = \mathbb{V}_{\mathcal{L},\xi_m}[\hat{p}_{\mathcal{L},\xi_m}(Y = c|X = \mathbf{x})]$$

Also, let $\hat{p}_{\mathcal{L},\xi_1,...,\xi_M}(Y = c|X = \mathbf{x})$ be the probability estimate by the ensemble for the class 'c'. Note that the final prediction of the ensemble is just the class with the highest probability estimate. Then it can be shown (from [20]) that for any $\mathbf{x} \in X$,

$$\mathbb{E}_{\mathcal{L},\xi_1,...,\xi_M}[\hat{p}_{\mathcal{L},\xi_1,...,\xi_M}(Y = c|X = \mathbf{x})] = \mu_{\mathcal{L},\xi}(\mathbf{x}),$$

where $\xi \in \{\xi_1, ..., \xi_M\}$. So, an ensemble of M randomised decision trees has the same bias as that for an individual randomised decision tree for predicting the probability estimate.

To calculate the variance of the ensemble, we will first need the correlation between any two randomised decision trees. Consider $f_{\mathcal{L},\xi_1}$ and $f_{\mathcal{L},\xi_2}$, where ξ_1 and ξ_2 are i.i.d as before. Then the correlation can be calculated as follows:

$$\begin{aligned}
\rho(\mathbf{x}) &= \frac{\mathbb{E}_{\mathcal{L},\xi_1,\xi_2}[(f_{\mathcal{L},\xi_1}(\mathbf{x}) - \mu_{\mathcal{L},\xi_1}(\mathbf{x}))(f_{\mathcal{L},\xi_2}(\mathbf{x}) - \mu_{\mathcal{L},\xi_2}(\mathbf{x}))]}{\sigma_{\mathcal{L},\xi_1}(\mathbf{x})\sigma_{\mathcal{L},\xi_2}(\mathbf{x})} \\
&= \frac{\mathbb{E}_{\mathcal{L},\xi_1,\xi_2}[f_{\mathcal{L},\xi_1}(\mathbf{x})f_{\mathcal{L},\xi_2}(\mathbf{x}) - f_{\mathcal{L},\xi_1}(\mathbf{x})\mu_{\mathcal{L},\xi_2}(\mathbf{x}) - f_{\mathcal{L},\xi_2}(\mathbf{x})\mu_{\mathcal{L},\xi_1}(\mathbf{x}) - \mu_{\mathcal{L},\xi_1}(\mathbf{x})\mu_{\mathcal{L},\xi_2}(\mathbf{x})]}{\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x})} \\
&= \frac{\mathbb{E}_{\mathcal{L},\xi_1,\xi_2}[f_{\mathcal{L},\xi_1}(\mathbf{x})f_{\mathcal{L},\xi_2}(\mathbf{x})] - \mu^2_{\mathcal{L},\xi_1}(\mathbf{x})}{\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x})},
\end{aligned}$$

where the simplifications are made because of the i.i.d property of ξ_1 and ξ_2. The correlation ρ measures the strength of the random perturbations introduced by the learning algorithm. If

$\rho(\mathbf{x}) = 0$, it would mean perfectly random models, whereas if $\rho(\mathbf{x}) = 1$, there would be no randomness. Having derived an expression for correlation, the variance of probability estimate prediction is:

$$
\begin{aligned}
\text{var}(\mathbf{x}) &= \mathbb{V}_{\mathcal{L},\xi_1,\ldots,\xi_M}\left[\frac{1}{M}\sum_{m=1}^{M} f_{\mathcal{L},\xi_m}(\mathbf{x})\right] \\
&= \frac{1}{M^2}\left[\mathbb{E}_{\mathcal{L},\xi_1,\ldots,\xi_M}\left[\left(\sum_{m=1}^{M} f_{\mathcal{L},\xi_m}(\mathbf{x})\right)^2\right] - \mathbb{E}_{\mathcal{L},\xi_1,\ldots,\xi_M}\left[\left(\sum_{m=1}^{M} f_{\mathcal{L},\xi_m}(\mathbf{x})\right)\right]\right]^2 \\
&= \frac{1}{M^2}\left[\mathbb{E}_{\mathcal{L},\xi_1,\ldots,\xi_M}\left[\sum_{i,j} f_{\mathcal{L},\xi_i}(\mathbf{x}) f_{\mathcal{L},\xi_j}(\mathbf{x})\right] - \left(M\mu_{\mathcal{L},\xi_1}\right)^2\right] \\
&= \frac{1}{M^2}\left[\sum_{i,j} \mathbb{E}_{\mathcal{L},\xi_i,\xi_j}\left[f_{\mathcal{L},\xi_i}(\mathbf{x}) f_{\mathcal{L},\xi_j}(\mathbf{x})\right] - M^2\mu^2_{\mathcal{L},\xi_1}(\mathbf{x})\right] \\
&= \frac{1}{M^2}\left[M\mathbb{E}_{\mathcal{L},\xi_1}[f_{\mathcal{L},\xi_1}(\mathbf{x})^2] + (M^2 - M)\mathbb{E}_{\mathcal{L},\xi_1,\xi_2}[f_{\mathcal{L},\xi_1}(\mathbf{x}) f_{\mathcal{L},\xi_2}(\mathbf{x})] - M^2\mu^2_{\mathcal{L},\xi_1}(\mathbf{x})\right] \\
&= \frac{1}{M^2}\left[M(\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}) + \mu^2_{\mathcal{L},\xi_1}(\mathbf{x})) + (M^2 - M)(\rho(\mathbf{x})\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}) + \mu^2_{\mathcal{L},\xi_1}(\mathbf{x})) - M^2\mu^2_{\mathcal{L},\xi_1}(\mathbf{x})\right] \\
&= \frac{\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x})}{M} + \rho(\mathbf{x})\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}) - \rho(\mathbf{x})\frac{\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x})}{M} \\
&= \rho(\mathbf{x})\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x}) + \frac{1-\rho(\mathbf{x})}{M}\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x})
\end{aligned}
$$

From above, as the size of the ensemble $M \to \infty$, the variance of prediction $\text{var}(\mathbf{x}) \to \rho(\mathbf{x})\sigma^2_{\mathcal{L},\xi_1}(\mathbf{x})$. Assuming that the randomised trees aren't perfectly correlated (i.e, there is some randomisation strength), and so $\rho(\mathbf{x}) < 1$, the variance of an ensemble is strictly less than the variance of an individual model. However, in practice, shrinking the correlation between trees usually leads to a higher variance of individual trees and so the three quantities, correlation, bias and variance should be balanced by tuning appropriate heuristics. Since, bias and noise, both remain the same, and variance is strictly less than an individual decision tree, we can conclude that the ensemble technique indeed supersedes the individual trees in terms of the EPE.

Moreover, it should be noted that we have proved that the quantity $\hat{p}_{\mathcal{L},\xi_1,\ldots,\xi_M}(Y = c|X = \mathbf{x})$ is a better estimate than $\hat{p}_{\mathcal{L},\xi}(Y = c|X = \mathbf{x})$, for ξ, a random seed. The actual predicted class by the ensemble is the class for which the probability estimate is the highest. Also, for the purpose of minimising the CCE loss, a better estimate of the probability distribution definitely brings down the loss.

More intuitively, the reasons for why ensembles work better than individual decision trees as stated by Dietterich [23] are three fold:

- For a small training set, a learning algorithm can typically find several models in the hypothesis space with the same performance on the training data. Assuming their predictions are

uncorrelated, averaging several models reduces the risk of choosing the wrong hypothesis.

- Many other algorithms rely on some greedy approach to optimisation, and hence fall prey to local optimal solutions. Even though the individual decision trees in an ensemble are formed using greedy algorithms, they all have different starting points and so have a more reliable solution as a whole.

- Finally, having a large number of constituent models where none of the individual models can represent the true function, in fact enrichens the space of representable functions.

The question that remains to answer is - where is the randomness injected in Random Forests?

- **Bagging**: This technique was proposed by Breiman [24] in which each decision tree in the ensemble is trained on a dataset of the same size as $\mathcal{L}$, drawn from $\mathcal{L}$ with replacement. For each draw, the probability of getting selected is the same for all the samples, and so for large data-sets, for the probability of a training sample getting chosen, we can make the following approximation:
$$P(\mathbf{x} \in \mathcal{L}_b) = 1 - (1 - \frac{1}{N})^N \approx 1 - e^{-1},$$
as $N \to \infty$. So each training sample contains 63% of the training samples in $\mathcal{L}$. Since each decision tree is trained on a slightly different training set, this technique has a de-correlating effect on the ensemble.

- **Random Feature Selection**: The other way of injecting randomness is by training each decision tree only for a subset of the training features, introduced by Ho [25] and is also called 'Random Subspace' method. This again has a decorrelating effect on the trees. This way, the output Y can be explained in several ways with trees that are different in structure. Doing this, increases the bias only slightly while the variance shrinks considerably when taken in an ensemble. This method requires us to tune the fraction of total number of features to be considered for each randomised decision tree. Some common choices are, the square root of the total number of features, or 50%, etc.

Recall that we introduced a class imbalance in our data during labelling. In the other three models such a class imbalance is taken into account by a weighted loss function. However, in random forests, weights are applied particularly at two different steps: (i) In tree induction procedure: class weights are used to weight the Gini criterion for finding splits (ii) During soft voting, instead of taking a simple average of the probability estimates for each class from the M decision trees, we take a weighted average, thereby giving higher weights to under-represented classes '1' and '5'.

In this section, we expanded the space of functions that our model is capable of representing and addressed non-linearity beteen input data. Now, we would like to discuss a model with an even

richer space of functions represented. Also, it is worth noting that all the above analysis focuses on zero-one loss, and any probability estimates produced by the model are only a bi-product of it. We would ideally want a model which given a differentiable loss function (for example, CCE), can directly minimise the loss associated with predicting the probability estimate, and thereby also improve the quality of class predictions. This is why we now review Deep Neural Network classifiers (or Multi-layer Perceptron in some literature).

3.3 Deep Neural Network Classifier

Consider our most basic classifier, the MLR model. We trained a network by first taking an affine transformation of the input vector, and then transforming that into a probability distribution by taking a softmax function. If we add atleast one hidden layer after taking the affine transformation and before applying softmax function, we get a Deep Neural Network. According to Universal Approximation theorem [8], doing so, makes the space of representable functions infinitely rich. However, in practicality, defining the appropriate parameters requires careful learning procedures. In this section, we will define a neural network for one hidden layer. However the same idea can be extended to many layers.

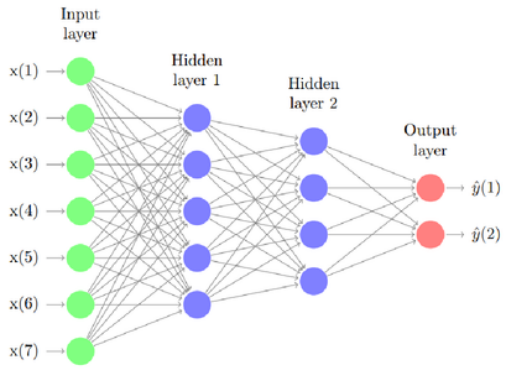

Figure 5: A feedforward network with architecture $7 \times 5 \times 4 \times 2$, taken from [3]

For Deep Neural Networks with p features, size of hidden layer s and number of classes K, the definition of the learning function $f_{\mathcal{L}}$ changes to following:

$$\mathbf{h}_t = \sigma(\theta^{(1)}\mathbf{x}_t + b^{(1)})$$
$$\hat{v}_t = \text{softmax}(\theta^{(2)}\mathbf{h}_t + b^{(2)}),$$

where $\theta^{(1)} \in \mathbb{R}^{s \times p}$, $\mathbf{h}_t, b^{(1)} \in \mathbb{R}^s$, $\theta^{(2)} \in \mathbb{R}^{K \times s}$, $b^{(2)} \in \mathbb{R}^K$. The function σ is a popular activation

function defined as:

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

In the above, we apply it elementwise to a vector in $\mathbb{R}^s$. For such a neural network we shall say that the architecture is $p \times s \times K$, and the set of parameters to be trained is $\{\theta^{(1)}, \theta^{(2)}, b^{(1)}, b^{(2)}\}$. We initialise all the parameters to random values and calculate the associated CCE loss at the output layer, as defined for MLR.

The process for error minimisation using backpropagation in Deep Neural Networks is similar to what we defined for Multinomial Logistic Regression. The only difference is that now we have an extra layer, with the sigmoid activation function. The derivative for the sigmoid function can be easily found to be:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Using the same chain rule as we did for Multinomial Logistic Regression, we can compute the gradient of the weighted CCE loss with respect to all the parameters and carry out the training process.

However, model training is neural network is not as direct as for Multinomial Logistic Regression where there were only a few parameters to train. In many practical applications neural networks may contain millions of trainable parameters. This would make the learning process computationally very expensive as at each update step, we would be training over all the data points available. For this purpose, training of DNNs is carried out in randomly chosen 'batches' of the training data. The batch size is considered as a hyperparameter to be tuned. The randomness due to training in batches also solves the problem of the algorithm getting trapped in the local minima. We also define the number of 'epochs' (another hyperparameter) as the number of times we iterate through a cycle of forward and backpropagation.

Now that we are dealing with a way higher number of trainable parameters than before, we are more exposed to the risk of our model overfitting. This calls for a need to regularise the network, even better using randomisation as seen for Random Forests. This can be done by using a technique called 'dropouts' introduced in [10], wherein some randomly chosen connections (or weights) between the two layers where we apply the dropout are set to zero. The fraction (called dropout ratio) of connections set to zero falls between 0 and 1, and is a hyperparameter to be tuned. A higher dropout ratio means a higher strength of regularisation, and hence a higher bias. Note that the concept of dropouts in Neural Networks is analogous to the random feature selection technique in Random Forests, where we trained different decision trees on a randomly chosen subset of the feature space.

This is a 'black-box' approach to modelling time series. Given the high number of parameters it

is extremely difficult for one to explain certain decisions made by the model. This is one of the biggest limitations of this approach, and also a reason for it gaining vast criticism.

3.4 LSTM Classifier

So far our approach has been to train the models using input features that were defined in Section 2. These are Exponential Moving Average based indicators that contain a certain amount of memory from the past. Our choice of lags and alphas was rather arbitrary and was kept sparse. This way of time series prediction can be a bit restrictive as we consider only a subset of information from the past to predict the future. Ideally, we would like the model to learn the weights associated with log returns at different time points from the past to predict the future instead of using exponential moving averages which already assume decaying weights. This gives motivation for Long Short Term Memory units introduced by Hochreiter & Schmidhuber [11] that can take sequential data such as time series data as input and model complicated patterns in them. We will first review Recurrent Neural Networks (which LSTM is a type of), and then address the problems with RNNs which make them a bad choice for modelling long range dependencies. Then we will review how LSTMs fix this problem.

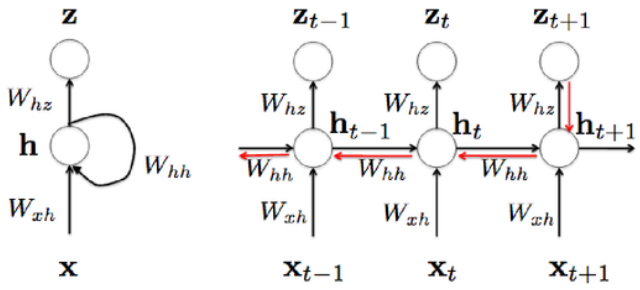

Figure 6: RNN expressed as a feedforward network

Before we dive into the theory, there are some important remarks to be considered. For the purpose of this model, we do not use the feature generation method as described in Section 2. Instead, in the implementation of LSTMs, we consider Gadget 5 year and Gadget 10 year sequential log returns with lookback period (or step size) of 50 timescale units. So, each training sample consists of series of 50 past log returns of 5 year and 10 year Gadget data, and the corresponding label for the future log return of Gadget 5-year.

As seen in figure 6, we see that an RNN is essentially a feedforward network in a loop, and so has a

training process similar to that of DNN. However, for DNNs, the hidden layer (or hidden variable) is only a function of $\mathbf{x}_t$, and so doesn't contain any memory of the past hidden states. What makes RNNs different is that for every time point t, the hidden state is a non-linear function of both, the input $\mathbf{x}_t$ and the hidden state at previous time point $t-1$, and hence recursively is a function of all previous hidden states. It is common to take this function as $tanh$, so that the output of the recurrent layer at time t, $\mathbf{h}_t$ of the RNN model can be defined as:

$$\mathbf{h}_t = tanh(W\mathbf{h}_{t-1} + U\mathbf{x}_t + \mathbf{b}^{(1)})$$
$$\hat{v}_t = \text{softmax}(O\mathbf{h}_t + \mathbf{b}^{(2)})$$

where $tanh$ (defined below) is applied element-wise,

$$tanh(x) := \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$\hat{v}_t$ is the model prediction as defined earlier, and the trainable model parameters are:

- W has dimensions $n \times p$
- U has dimensions $n \times n$
- O has dimensions $K \times n$
- $\mathbf{b}^{(1)}$ and $\mathbf{b}^{(2)}$ are bias vectors used to translate the non-linear transformations, and have dimensions $n \times 1$ and $K \times 1$ respectively.
- $\mathbf{h}_t$ is time t output with dimensions $n \times 1$ and $\hat{v}_t$ is the prediction with dimensions $K \times 1$.

In the above, n is the step-size in the implementation of LSTMs, taken to be 50 in this thesis, p is the number of features, taken to be 2 as explained earlier, and K is the number of nodes in the output layer, which is 5 for the purpose of this thesis. Note that, figure 6 was taken from Chen ([26]), where W_{hh} is W, W_{xh} is U, W_{hz} is O and $\mathbf{z}_t$ is $\hat{v}_t$.

All the trainable parameters of the model are first initialised with some random values, and the associated loss is calculated by forward propagation as in the case of Multinomial Logistic Regression and Deep Neural Networks. This is followed by gradient descent for loss minimisation at the output node, which is done by a variation of backpropagation algorithm called 'Backprogagation Through Time' or BPTT. This is thoroughly discussed in [26] and will be only reviewed in this thesis.

For ease of notation, let $\mathbf{a}_t = W\mathbf{h}_{t-1} + U\mathbf{x}_t + \mathbf{b}_h$. Then using chain rule we get that,

$$\frac{\partial L_\omega}{\partial O_{i,j}} = \sum_{t=1}^{N} \frac{\partial L_\omega}{\partial \hat{v}_t} \frac{\partial \hat{v}_t}{\partial O_{i,j}}$$
$$\frac{\partial L_\omega}{\partial b_i^{(2)}} = \sum_{t=1}^{N} \frac{\partial L_\omega}{\partial \hat{v}_t} \frac{\partial \hat{v}_t}{\partial b_i^{(2)}}$$
$$\frac{\partial L_\omega}{\partial W_{i,j}} = \sum_{t=1}^{N} \sum_{k=1}^{t} \frac{\partial L_\omega}{\partial \mathbf{a}_{k,i}} \mathbf{x}_{t,j}$$
$$\frac{\partial L_\omega}{\partial U_{i,l}} = \sum_{t=1}^{N} \sum_{k=1}^{t} \frac{\partial L_\omega}{\partial \mathbf{a}_{k,i}} \mathbf{h}_{t,l}$$
$$\frac{\partial L_\omega}{\partial \mathbf{b}_i^{(1)}} = \sum_{t=1}^{N} \sum_{k=1}^{t} \frac{\partial L_\omega}{\partial \mathbf{a}_{k,i}},$$

where,

$$\frac{\partial L_\omega}{\partial \mathbf{a}_{k,i}} = \frac{\partial L_\omega}{\partial \hat{v}_t} \frac{\partial \hat{v}_t}{\partial \mathbf{h}_{t,i}} \frac{\partial \mathbf{h}_{t,i}}{\partial \mathbf{h}_{k,i}} \frac{\partial \mathbf{h}_{k,i}}{\partial \mathbf{a}_{k,i}}.$$

and $i, l \in \{1, ..., d\}$ and $j \in \{1, ..., p\}$. In the above gradient, $\frac{\partial \mathbf{h}_{t,i}}{\partial \mathbf{h}_{k,i}}$ is calculated iteratively using the relation:

$$\frac{\partial \mathbf{h}_{t,i}}{\partial \mathbf{h}_{k,i}} = \prod_{r=k}^{t-1} \frac{\partial \mathbf{h}_{r+1,i}}{\partial \mathbf{h}_{r,i}} \tag{3.1}$$

Once, we have the gradients for all the parameters, a gradient descent approach (as explained previously) can be used to minimise the loss.

In theory, RNNs being a subset of neural networks, can approximate any function (from [8]). However in practice, RNNs fail to capture long-range dependencies. Since $\mathbf{h}_t$ is expressed as a *tanh* function, its range is from -1 to 1 and the range of its derivative is from 0 to 1. In equation 3.1, as $t - k \to \infty$, the gradient $\frac{\partial \mathbf{h}_{t,i}}{\partial \mathbf{h}_{k,i}}$ will go to zero. This is called the 'Vanishing Gradient Problem' and leads to loss of learning capabilities for capturing long term dependencies. The similar problem is faced if we use the sigmoid function for this purpose. If we attempt to fix this problem by replacing the activation function with a function with gradient larger than 1, we might face an 'Exploding Gradient Problem' as the gradient in question would explode to high values, disrupting the learning process.

LSTMs are a variation of RNNs invented to fix the drawbacks of Recurrent Neural Networks, particularly, vanishing gradient problem. As shown in figure 7 (top), in vanilla RNNs, the hidden state is just a *tanh* function of linear combination of input and previous hidden states. In LSTM, this relation is more complicated and is controlled by structures called 'gates'. Crucial to LSTM is a conveyor belt like structure called the 'cell state' $\mathbf{c}_t$, which is passed on from cell to cell (top flow in LSTM, figure 7). Note that in the figure, pink nodes signify element-wise operations

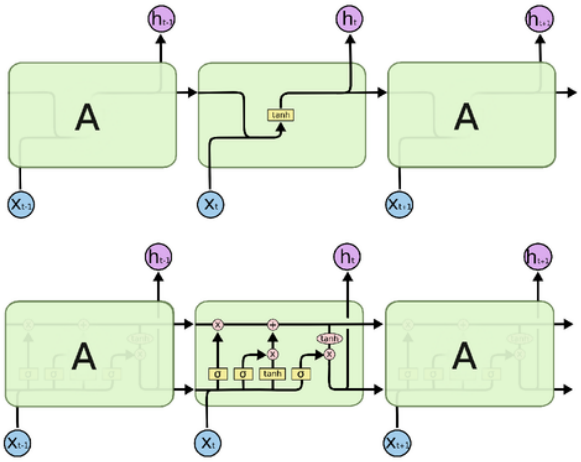

Figure 7: Difference between RNN(top) and LSTM(bottom), taken from [27]

and the yellow nodes are the activation functions. We have already defined *tanh*, and σ is the sigmoid function. Since the range of the sigmoid function is between 0 and 1, they can be used to regulate the information flow. The cell state holds the memory of the network and addition of any new information entering the network is regulated at each time point by gates. This removes LSTM's dependence on iterative multiplication of gradients. The gate mechanism in LSTM can be characterised by the following equations:

- **Forget gate**: This gate controls the amount of information that is remembered from the cell state. Corresponding to each number in the cell state c_{t-1}, it outputs a number between 0 and 1 (range of the sigmoid function) to regulate its information flow. The related equation is:

$$\mathbf{f}_t = \sigma(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + \mathbf{b}_f)$$

- **Input Gate Layer and Temporary Cell State**: These two gates regulate the new information that is to be stored in the current cell state. First, the input gate layer decides which values will be updated:

$$\mathbf{i}_t = \sigma(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + \mathbf{b}_i),$$

  and then we define a temporary cell state, $\tilde{\mathbf{c}}_t$, which regulates the extent to which they are updated:

$$\tilde{\mathbf{c}}_t = tanh(W_{\tilde{c}} \mathbf{x}_t + U_{\tilde{c}} \mathbf{h}_{t-1} + \mathbf{b}_{\tilde{c}}),$$

- **Cell State Update**: The cell state at time t can now be defined by a linear of combination of (i) memory from the previous cell state and (ii) information extracted from the current

input. So, the cell state update equation is:

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tilde{\mathbf{c}}_t$$

- **Hidden state**: The hidden state output $\mathbf{h}_t$ is a function of the cell state $\mathbf{c}_t$. We first define the output gate as

  $$\mathbf{o}_t = \sigma(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + \mathbf{b}_o),$$

  which regulates what in the current cell state should be output, and then multiply it with a scaled version of the current cell state $\mathbf{c}_t$ as follows:

  $$\mathbf{h}_t = \mathbf{o}_t \circ tanh(\mathbf{c}_t)$$

- **Prediction**: Finally, the prediction $\hat{v}_t$ is a function of $\mathbf{h}_t$. This could be through another hidden layer, or directly through a softmax function.

This gives the set of trainable parameters for LSTM to be $\{W_f, U_f, \mathbf{b}_f, W_i, U_i, \mathbf{b}_i, W_{\tilde{c}}, U_{\tilde{c}}, \mathbf{b}_{\tilde{c}}, W_o, U_o, \mathbf{b}_o\}$ with appropriate dimensions. Hence, the recurrent layer of LSTMs has four times more parameters to train than the recurrent layer of vanilla RNNs. This makes the possible space for prediction functions much more vast than any other models we have previously reviewed. However, with this greater degrees of freedom, comes a greater risk of overfitting the data. This also means that LSTMs require a higher degree of regularisation and more powerful optimisers with adaptive learning rates as compared to DNNs. In the next section we will review some developments in this segment, where the learning rate is made adaptive with momentum for faster convergence and for avoiding falling into local minima.

3.5 A Note on Optimisers

Training time and convergence of models for both, DNNs and LSTMs, are largely impacted by the choice of optimiser. In this thesis, we will discuss 3 optimisation algorithms existing in current literature: Mini-batch Gradient Descent, Adaptive Moment Estimation (Adam) and the newest development, Rectified-Adam (RAdam). The theory and equations for Mini-batch Gradient descent and Adam are well explained in [28] and RAdam was introduced at Microsoft (see [15]).

Starting with the very basic optimiser, Mini-batch Gradient Descent algorithm performs parameter updates in small batches of the training sample. This way it greatly reduces the variance of the parameter updates as compared to the case where updates are made individually for each training sample (as in Stochastic Gradient Descent). It also ensures speed and tractability relative to the case where the whole dataset is considered for every epoch (Batch Gradient Descent). The idea

of training in batches also enables parallel computing (atleast for non-recurrent models) and takes up less RAM as the whole dataset doesn't have to considered at once. Note that in computer applications, Mini-batch Gradient Descent is usually referred to as Stochastic Gradient Descent, and so the two terms will be used interchangeably in this thesis as well.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t) \tag{3.2}$$

While the above optimiser led to a fast convergence in the case of Deep Neural Networks, recurrent neural networks such as LSTMs require more powerful approaches to gradient descent. As discussed in [28], the concept of having a fixed learning rate for training doesn't yield good results if the data is sparse and the features have very different frequencies. To address this problem, Adam uses an adaptive learning rate mechanism for each parameter based on (1) exponentially decaying average of past gradients (estimate of the first moment or the mean), and (2) exponentially decaying average of past squared gradients (estimate of the second moment or the uncentered variance). Hence, the update rule becomes:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta L(\theta_t) \\ \nu_t &= \beta_2 \nu_{t-1} + (1-\beta_2)(\nabla_\theta L(\theta_t))^2 \\ \theta_{t+1} &= \theta_t - \tfrac{\eta}{\sqrt{\nu_t}+\epsilon} m_t \end{aligned} \tag{3.3}$$

According to the authors of Adam, suitable values for β_1, β_2 and ϵ are 0.9, 0.999 and 10^{-8} respectively. For empirical purposes, m_t and ν_t are replaced by their bias corrected versions:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{\nu}_t = \frac{\nu_t}{1-\beta_2^t}.$$

However, Liu et al. [15] successfully identified the problem of high variance of the adaptive learning rate as the fundamental cause of convergence to suspicious local optima in Adam and other adaptive learning rate algorithms. They pointed out that the adaptive learning rate has large variance in the early stage of model training due to lack of training samples being used. To fix this problem, they proposed R-Adam which rectifies the variance issue and has a reduced susceptibility to the choice of the hyperparameter η due to improvement in robustness of model training.

R-Adam improves by bringing consistency in the variance of ν_t. Let $g_t := \nabla_\theta L(\theta_t)$, then for a generic adaptive learning rate optimiser, ν_t is set to $\psi(g_1, ..., g_t)$. For Adam, this function is

$$\psi(g_1, ..., g_t) := \sqrt{\frac{1-\beta_2^t}{(1-\beta_2)\sum_{i=1}^t \beta_2^{t-i} g_i^2}}$$

It can be shown that $\psi^2(g_1, ..., g_t)$ approximately subjects to a scaled inverse chi-square distribution with ρ_t degrees of freedom, where ρ_t is due for approximation. From Theorem 1 in [15], we have that

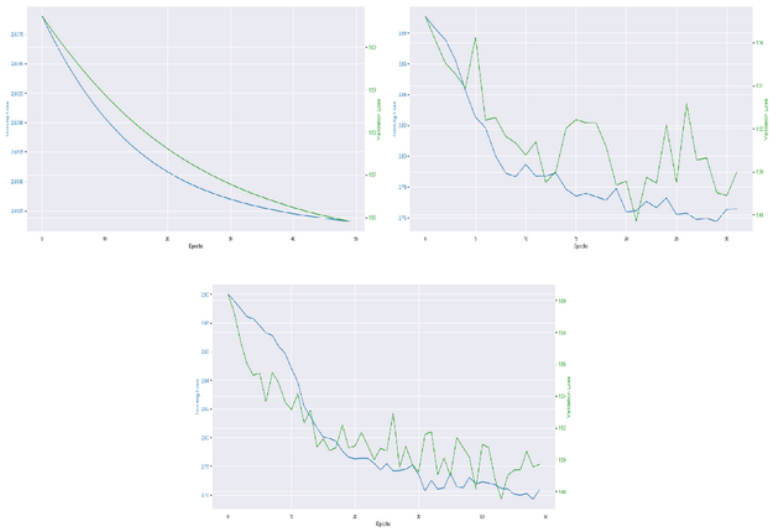

Figure 8: LSTM training with architecture 50 x 0.5 x LSTM(50) x 0.5 x 5 for 2 minute look-forward prediction using (i) MB-GD (ii) Adam (iii) R-Adam

$Var(\psi(.))$ decreases monotonically with increase in ρ_t, which is exactly what we want. Further, the authors prove that $\rho_t \approx f(t, \beta_2)$, where f is defined as

$$f(t, \beta_2) := \frac{2}{1-\beta_2} - 1 - \frac{2t\beta_2^t}{1-\beta_2^t}$$

Clearly, for a fixed β_2, ρ_t is monotonically increasing in t, and so the maximum value of ρ_t is

$$\rho_\infty := \frac{2}{1-\beta_2} - 1$$

and so this is the number of degrees of freedom for which the variance of $\psi(.)$ is minimum. Based on this, a rectification term r_t for all t can be found such that we have, $Var(r_t\psi(.)) = Var(\psi(.))_{\rho_t=\rho_\infty} = \kappa$, some constant. In the paper, this term is approximated to be

$$r_t := \sqrt{\frac{(\rho_t - 2)(\rho_t - 4)\rho_\infty}{(\rho_\infty - 2)(\rho_\infty - 4)\rho_t}}$$

Due to the constraints on ρ_t in r_t, the rectified adaptive learning rate term $r_t\nu_t$ is only applied to the parameter update when $\rho_t > 4$. This way R-Adam ensures that the variance of adaptive learning rate is consistent over the learning period leading to better convergence.

When the 3 discussed optimisers were used for training an LSTM network on Gadget dataset for 50 epochs, the learning curves as shown in figure 8 were observed. While mini-batch Gradient Descent is stable, the convergence to minima is slow as the loss only decreases to 1.56 as opposed

to below 1.5 in the other two. For Adam, even though the optimiser can be considered strong in terms of bringing the loss down, there are major fluctuations in validation loss indicating poor convergence. This is further improved by R-Adam which succeeds in bringing the loss down with a relatively more stable learning curve.

4 Empirical Results

Having discussed the underlying theory for each model, we will now discuss the hyper-parameter tuning and performance of these techniques for predicting Gadget 5-year. Two prediction horizons were considered: short-term (2 minutes) and long-term (10 minutes). All the results are out-of-sample, for the months of June and July, covering 9900 data points in total. Training and validation was done on approx. 10000 data points for the months of April and May.

4.1 Model Evaluation:

For each model, we will report the Expected Prediction Error (EPE) based on a Categorical Cross Entropy loss, where the EPE will be an average of 10 simulations. Since EPE is only a measure of the goodness of estimate of the probability distribution, the predictive power of the classifiers will also be evaluated using the following classification metrics derived from the confusion matrices for each class:

- **Precision**:
$$\frac{TruePositive}{TruePositive + FalsePositive}$$
- **Recall**:
$$\frac{TruePositive}{TruePositive + FalseNegative}$$
- **F1-Score**:
$$2 * \frac{Precision * Recall}{Precision + Recall}$$

More intuitively, for each class, Precision measures the number of times the classifier correctly predicted that class out of the total number of times the classifier predicted that class, and hence is a measure of how 'reliable' a model is for predicting a particular class. Recall measures the number of times the classifier correctly predicted that class out of the total number of times that class occurred in the dataset. and hence, measures how 'opportunistic' a model is towards a particular class. F1-score is simply the harmonic mean of Precision and Recall.

Note that it may be tempting to consider accuracy as the base metric, but accuracy doesn't give a true measure of a classifier's performance when there is class imbalance. For example, a trivial model predicting '3' in this case everytime would be a highly accurate one but mostly redundant.

4.2 Loss and Average F1-score Comparison

In figure 9 we see that in both short-term and long-term cases, all the four models have a loss less than the random model. Hence, we can conclude that all the four models beat the random

	Random	MLR	RF	DNN	LSTM
2-minutes	1.6	1.5720	1.5602	1.5708	1.5599
10-minutes	1.6	1.5738	1.5730	1.5736	1.5727

Figure 9: CCE loss on Test set averaged over 10 simulations

model when it comes to predicting the probability distribution over the 5 classes (defined to be $\hat{v}$). Moreover, we observe that this loss is more for 10-minute predictions than for 2-minute predictions, which confirms that it is easier to predict nearer into the future.

In terms of ranking of the models, we observe that LSTM brings down the loss of prediction the most, followed by the Random Forest model. We don't observe a significant difference between the performance of DNN and Multinomial Logistic Regression as the loss for both the cases is similar.

	Random	MLR	RF	DNN	LSTM
2-minutes	0.2	0.27	0.27	0.26	0.27
10-minutes	0.2	0.19	0.22	0.20	0.25

Figure 10: Average F1-score over 5 classes

In figure 10, for 2-minute predictions, we observe that averaged across the 5 classes, each model performs 35% over the baseline of 0.2. However, for prediction horizon of 10 minutes, the reduced loss doesn't necessarily translate to a higher predictive power. In the next sub-section, we will explore this in greater detail.

4.3 Hyperparameter Tuning and Predictions

- **Multinomial Logistic Regression**: The hyperparameter λ in the regularisation term is calculated using a 5-fold Time Series Cross Validation as shown in Algorithm 1. It is clear from the diagram below how first with increase in λ (decrease in regularisation) the loss decreases monotonically until an optimal point, and increases after that. It is at this point that the bias and variance of the model are balanced to give the minimum loss. Any further decrease in regularisation leads to increase in model variance.

  Something worth noting for comparing the 2 minute and 10 minute cases is the difference between the optimal λ in the two. For 2-minute predictions, the optimal point is $\lambda = 0.45$ and for 10-minute predictions, $\lambda = 0.01$. Since λ is inversely related to regularisation strength,

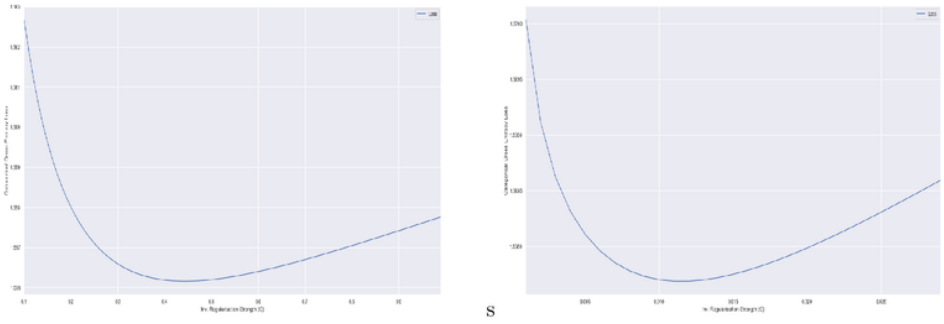

Figure 11: Hyperparameter λ tuning for 2 mins (L) and 10 mins (R) prediction

we can conclude that the model needs stronger regularisation for the 10-minute case. This is in line with our intuition, since for the same data, the signal to noise ratio will decay as the prediction horizon gets longer and so the same data would be more prone to overfitting for long term predictions. This can also be attributed to market efficiency, wherein any long-term alpha would be discovered and quickly exploited.

Given in figure 12 is the classification report for Multinomial Logistic Regression for 2 minutes prediction. On the rows, we have the actual label, and on the columns, we have the predicted label. For example, from the matrix below, the model correctly predicted '5' as '5' 438 times, and incorrectly predicted '5' as '1' 87 times.

Actual \ Predicted	1	2	3	4	5	Support
1	324	448	239	281	171	1463
2	259	764	516	579	328	2446
3	174	623	444	555	306	2102
4	201	651	488	662	426	2428
5	87	269	223	396	438	1413
Precision	0.31	0.28	0.23	0.27	0.26	
Recall	0.22	0.31	0.21	0.27	0.31	
F1-Score	0.26	0.29	0.22	0.27	0.28	

Figure 12: MLR Classification Report for 2 minutes prediction

From the classification report we can see that MLR improves the F1-score over the baseline of 0.2 for all the 5 classes. Further it can also be observed that the model understands the hierarchy of the classes. For example, against a prediction of class '1' (see the column under '1'), we observe decreasing frequencies of prediction along the column, i.e, the model understands that '1' is closer to '2' than to '5'. Likewise, under the prediction class '3', this

distribution of frequencies is more or less symmetric. Notice that we observe a negligible improvement of F1-score under class '3'. While this can be seen as a room for improvement, we will see in the next section that this doesn't hurt the investor because we don't trade when the model signals '3'.

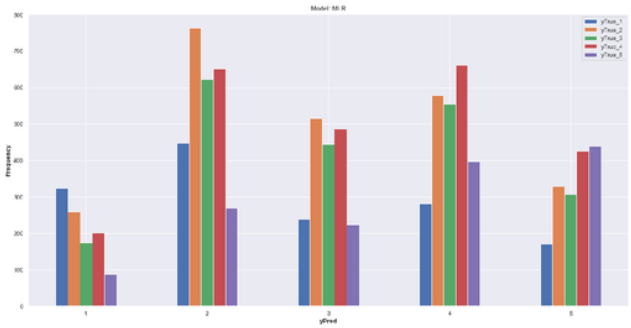

Figure 13: MLR Performance for 2 minutes prediction

Another way of visualising alpha generated is the bar chart in figure 13. On the x-axis, we have the signal generated by the model, and for each class prediction, we have the actual underlying distribution of frequencies. We see a strong skew in frequency distributions of classes '1' and '5', and none at all for '3'.

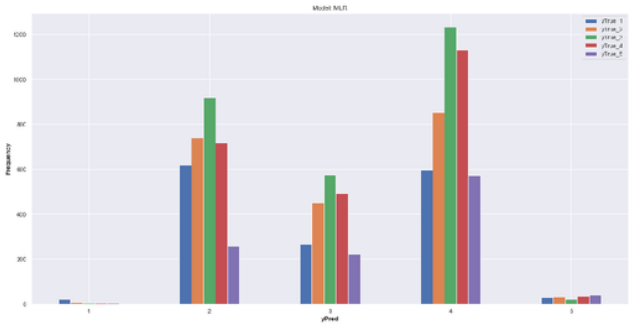

Figure 14: MLR Performance for 10 minutes prediction

A similar bar chart is presented for 10-minute predictions in figure 14. We observe that the desired skew of frequencies is now much fainter in most classes, signifying diminishing alpha. Also, we observe that the performance of the model towards predicting extreme classes '1' and '5' is severely affected, which is intuitive, as extreme movements are the hardest to predict.

- **Random Forest**:

  In Random Forest classifier, we focused on 3 hyper-parameters:

  – **Number of estimators** (or trees) in the forest (parameter M): As established in section 3, as the number of estimators in the forest increases, the prediction variance decreases whereas the bias remains the same. Although, having a large number of estimators only makes the model more efficient and doesn't lead to overfitting, it is computationally expensive. Usually, the loss goes down with increase in number of estimators until a critical point, after which it plateaus, as illustrated below. For our analysis we took $M = 500$, as it was fast enough (3 seconds) for our dataset and guaranteed the loss plateau.

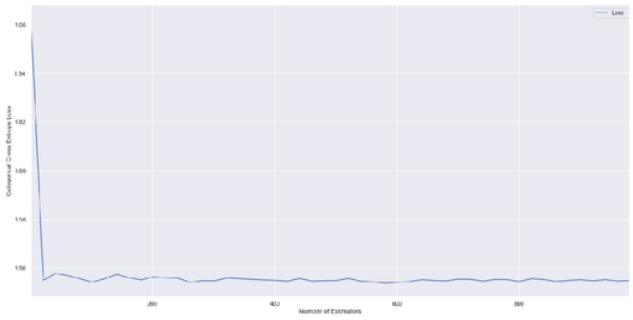

Figure 15: Loss Vs. Number of Estimators (elbow point at M=25) for 2-minute predictions

  – **Maximum Features**: as discussed in Section 3, as this value increases, the strength of randomisation for the random forests decreases, and hence the correlation between individual decision trees increases. More concisely, the higher this number is, the lower is the variance for the ensemble. However, a high value for this number can also lead to a high bias and so it needs to be carefully tuned. This is an ensemble specific hyperparameter.

  – **Minimum Samples per Leaf Node**: this parameter was explained in Section 3 as a stopping criterion for the tree to grow. The more we allow the individual trees to grow (low minimum samples per leaf node), the more prone they are to overfitting, and have a high variance. This is a tree specific hyperparameter.

  To select the optimal hyperparameters for Random Forests, we use a technique called Grid Search, where the grid contains a value of metric of interest (categorical cross entropy loss), against every hyperparameter combination in the hyperparameter space. In this thesis, we

shall consider a 2-dimensional grid (for maximum features and minimum samples per leaf node). Also, apart from the stopping criterion defined, we let any other hyperparameters be as liberal as possible. For example, there is no limit on the maximum depth of the tree, minimum impurity split, and minimum samples split, maximum leaf nodes, etc. are all set to *none*. Also, we use bagging as explained in Section 3 for training the random forests. We fix the number of estimators during cross validation to $M = 100$.

For an appropriate hyperparameter space of the form (minimum samples leaf $\times$ maximum features),

$$\mathcal{H} := \{1, 50, 10, 40, 80, 100, 200, 500\} \times \{0.1, 0.2, 0.4, 0.8, 1, sqrt\}$$

the grid after 5-fold cross validation for 2-minute prediction is given in figure 16.

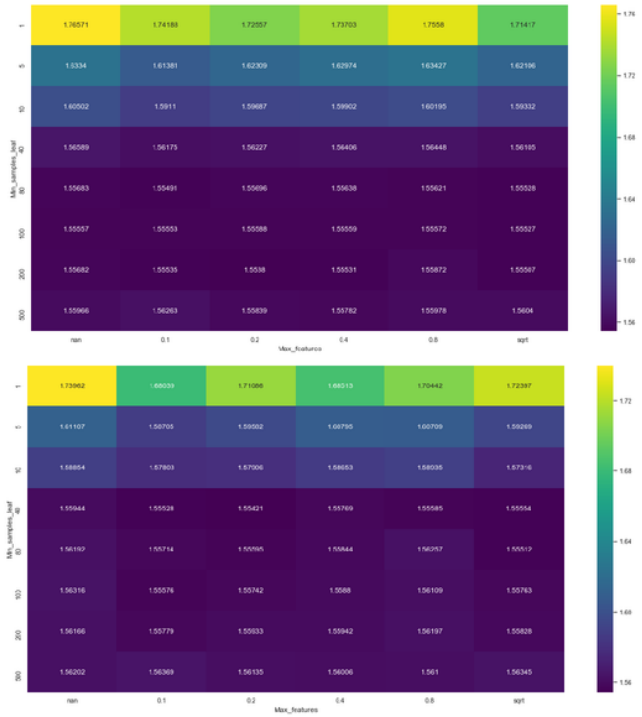

Figure 16: Random Forests Grid Search for 2 minutes (top) and 10 minutes (bottom)

From the grid, it can be seen that for 2-minute prediction the optimal value for the hyperparameters minimum samples leaf and maximum features is 80 and 0.1 respectively. Surprisingly, we see a similar result for 10-minute predictions which can be potentially attributed

to the fact that for increasing time horizons, the correlation between 5Y and 10Y Gadget increases (see figure 2.2), and that the RF model is the best at exploiting it.

Note that the 'NaN' column means that there is no randomisation in the feature selection process for Random Forests. Similarly, for minimum samples leaf set to 1, there is no stopping criterion for the leaves in the individual decision trees. We can also observe, that for any fixed value of maximum features hyperparameter, the loss first decreases until some point, and then increases with increase in minimum samples split. This can again be explained by the concept of bias-variance trade-off.

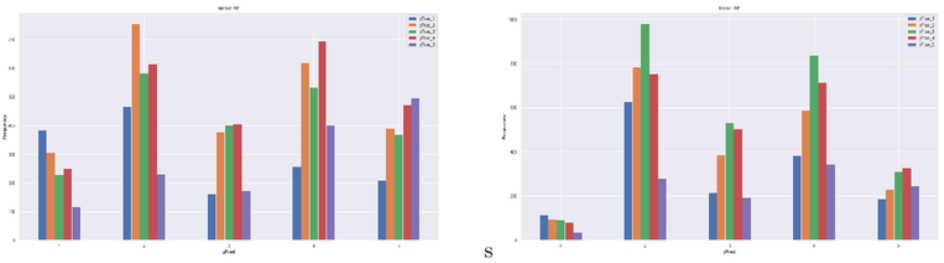

Figure 17: RF Performance for 2 minutes (L) and 10 minutes (R)

Just as for the MLR 2-minute case, we observe the desired skew of frequencies for each label in RF as well (figure 17). One stark difference this time is in the 10-minute case, where we begin to observe improvements in classes '1' and '5'. This proves RF's credibility over MLR for longer term predictions. The classification report for this model, along with the next two models, is given in the appendix.

- **Deep Neural Network**:

  For the DNN classifier, the hyperparameters to be tuned were the model architecture, number of epochs, learning rate, batch size, and dropout ratios. The choice of optimiser was already established to be Stochastic (Mini-batch) Gradient Descent with a learning rate of 10^{-3} (see Section 3, A Note on Optimisers).

  The model architecture was chosen based on some simple rules of thumb for network training. We fixed the number of hidden layers to 2 to capture a higher degree of non-linearity and then followed a simple approach. The number of nodes in the hidden layers was always kept less than the number of nodes in the input layer (number of features), and more than the

number of nodes in the output layer (five, in our case). Also, the number of nodes in the first hidden layer was kept more than or equal to the number of nodes in the second hidden layer. For each configuration, the model was trained, and the learning curve was observed for 500 epochs and a minimal batch size of 16. If the training loss was seen to diverge from the validation loss (a sign of overfitting in neural networks), the problem was solved by adding more nodes in the hidden layers, and considering regularisation (dropouts, as explained in Section 3) between different layers.

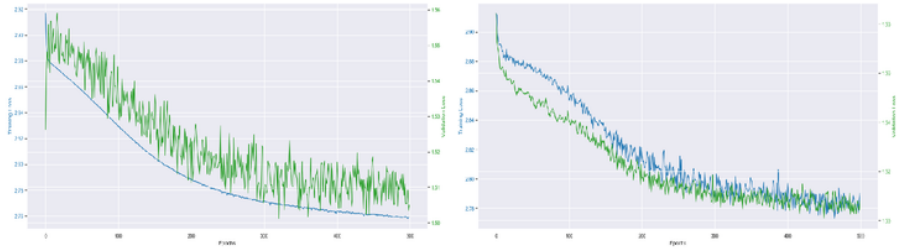

Figure 18: DNN 2 minute prediction without (L) and with (R) dropout ratio of 0.2 for architecture $73 \times 36 \times 36 \times 5$

The effect of dropouts can be clearly seen in figure 18, where we first considered a deep neural network for 2 minute predictions, without any dropouts and then applied a dropout ratio of 0.2 between the two hidden layers, for the model architecture $n_{inputs} \times n_{inputs}/2 \times n_{inputs}/2 \times 5$, using sigmoid activation function on each hidden layer and softmax function for the output layer. This general architecture translated to $73 \times 36 \times 36 \times 5$ and $4,145$ trainable parameters in total for our problem. The training and validation time recorded for this architecture was 500 seconds ($\approx$ 1 second per epoch).

Note that in figure 18, the training loss and the validation loss are on different scales (blue for training, green for validation). This is because during training, the model loss is calculated taking the class weights into account, however during validation/testing, no class weights are applied.

Analogous to the MLR model, where for 10-minute prediction we needed a higher regularisation strength, for DNNs also, it was observed that the training curve is better when a higher dropout ratio of 0.5 is applied, with the same architecture as above.

Close to the performance of MLR 10 minute case, where the model didn't perform well for the extreme cases, we see a similar phenomenon occuring in the DNN classifier. This is also

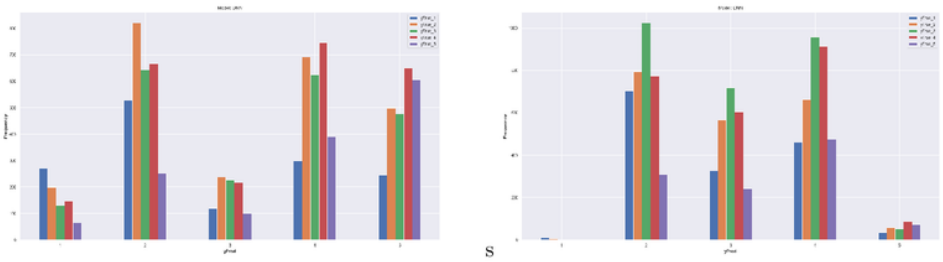

Figure 19: DNN Performance for 2 minutes (L) and 10 minutes (R)

reflected by its average F1-score, which is only 0.2 and close to that of MLR.

- **Long Short Term Memory**:

  The same parameters were tuned for LSTMs using a similar approach. The optimal architecture for LSTMs was found to be $n_{inputs} \times \text{LSTM}(n_{inputs}) \times n_{inputs} \times 5$, with a dropout of 0.5 between the recurrent layer and the hidden layer, and also between the hidden layer and the output layer. This amounted to 13,405 trainable parameters in total. Notice that this time, we applied a higher amount of regularisation to the network as compared to the case of DNN. An explanation for this is that the number of trainable parameters is much higher in this model as compared to the last one and hence it is more prone to overfitting. In fact, for this model, the number of trainable parameters is more than the number of training samples, which is indeed a reason why it needed much stronger regularisation. It can be observed from figure 20 that not applying regularisation gave an extremely volatile training procedure which diverges to a very high value of CCE loss by the end of the training process.

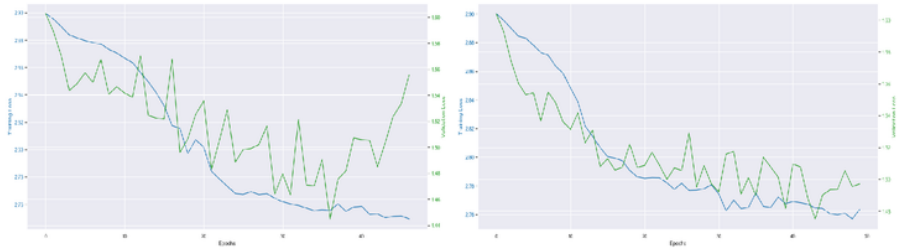

Figure 20: LSTM 2 minute prediction without(L) and with(R) dropout ratio of 0.5 around the non-recurrent hidden layer

Furthermore, for LSTMs, the choice for optimisers was Rectified-Adam, as discussed in Section 3. Due to robustness of training and lower susceptibility to learning rate in R-Adam,

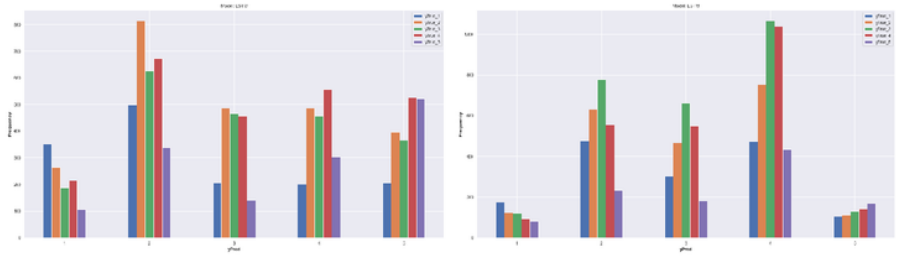

Figure 21: LSTM performance for 2 minutes (L) and 10 minutes (R)

the learning rate was taken to be 10^{-3}. Since the optimiser used was stronger than SGD, the number of epochs was reduced to 50 with a batch size of 64.

5 Trading Strategy

Another way of confirming models' alpha generation is assessing their performance when used in a trading strategy. This way we can test its profitability and also study the behaviour of models in different market conditions. The trading strategy implemented was kept simple to make the models' predictive power more explicit. The strategy closes out all the long positions and opens two short positions if the label is 1, and closes out all the long positions and opens one short position if the label is 2. Likewise, the strategy closes out all the short positions and opens two long positions if the label is 5, and closes out all the short positions and opens one long position if the label is 4. The strategy simply holds on to the inventory if the label is 3.
In calculating the cumulative PnL, the following assumptions are made:

- We start with 0 basis points.
- There is always sufficient cash available to maintain the brokerage account margin.
- No restrictions on the minimum or maximum holding period and overnight positions.
- We trade mid-to-mid, transaction spread is ignored.
- We ignore any slippage effects - it is assumed that a market order gets filled immediately.
- No operational risk measures are deployed, such as placing stop-loss orders

Along with the four models that we have trained, we will also implement the strategy in a 'Random' scenario, where we will put a random number generator in place to predict the outcome.

We will consider the following trade metrics to compare results across models:

- Net Profit: Basis points captured at the end of two months
- Total Trades: Important to know if the model is trading too much or too scarcely.
- Long Positions %: Generally we would want number of long and short positions to be roughly the same. Extreme percentage of long positions would depict model's bias towards a certain kind of signal, and hence any profitability would be just a fluke.
- Profit Factor: Total spread won on winning trades over total spread lost on losing trades.
- Profitable (%): Percentage of profitable trades
- Profit per Trade: Total Spread gained over total number of trades. It is a measure of how effectively the model trades.

5.1 Trading every 2 minutes:

The first striking observation from the figures in 22 is that all the four models have alpha, and in a mid-to-mid scenario show a consistent net gain of basis points at the end of the two months. The random number generator has some profit but it is safe to assume that it is just a fluke. Moreover, for all the models, the number of long and short positions is roughly the same which suggests that they are not biased towards any one particular kind of order. The inventory track confirms that none of the models is repeatedly buying or selling in large amounts, which would be an unusual behaviour. There are some spikes in the inventory track which can be visually mapped with periods of successive drops and hikes in the spread in the top figure, which is a desirable feature. For all the models, the percentage of profitable trades is close to two-thirds, which suggests that we gain more often than we lose. Also, the profit factor being way higher than 1 suggests that we win more than we lose.

It can be observed that the DNN classifier has an edge over the other three models throughout the trading period. The first argument for this behaviour could be that this model trades more frequently than the other three. However, in terms of efficiency of trading (profit per trade), it still ranks the highest which confirms its top rank. While we can comment that the DNN performs the best over specifically these two months, we cannot extrapolate this statement until we have a much larger dataset. Moreover, due to the black box approach of this model. it can't be explained very well why it reacts to certain market conditions in some way.

Sharing the second position are the MLR and RF classifiers. The performance for both is consistently very close to each others'. However we can observe that the RF classifier trades way more frequently than the MLR classifier, shrinking the former's profit per trade to 0.05 against 0.054 of the latter. Finally, we have the LSTM, with the lowest efficiency in trading, and performs consistently the worst out of the four.

We observe that the ranking of efficiency of trading for the four models is not in line with how much they bring down the CCE loss, or even their F1-scores. One of the reasons for this can be that in our trading strategy, we clear the inventory for all opposite positions before taking a certain position. Due to this reason, DNN might be performing the best because it consecutively gives similar predictions and hence has a tendency to accumulate inventory. This is evidenced by the inventory track where the inventory for DNN (shown with a dotted line) has the highest peaks. So one can comment that the profitability of the high profitability of DNN is not solely due to its predictive power, but also due to its tendency to take positions of similar nature consecutively.

	Random	MLR	RF	DNN	LSTM
Net Profit	52.75	431.12	425.8	498.9	364.32
Total Trades	7902	7942	8488	8954	8098
% Long	0.49	0.52	0.52	0.58	0.5
Profit Factor	1.14	2.98	2.94	3.15	2.42
% Profitable	0.44	0.66	0.72	0.686	0.66
Profit per Trade	0.007	0.0543	0.0502	0.0557	0.0449

Figure 22: Top to bottom: (i) 5y Gadget in June and July (ii) PnL (iii) Inventory (iv) Trade metrics

5.2 Trading every 10 minutes:

	Random	MLR	RF	DNN	LSTM
Net Profit	-41.40	92.68	110.52	111.07	100.47
Total Trades	1578	1563	1606	1481	1538
% Long	0.50	0.57	0.52	0.49	0.56
Profit Factor	0.78	1.98	1.88	2.36	2.05
% Profitable	0.45	0.66	0.69	0.67	0.64
Profit per Trade	-0.0262	0.0593	0.0688	0.0750	0.0653

Figure 23: Top to bottom: (i) PnL (ii) Inventory (iii) Trade metrics

As before, we observe profitability over the random model for all the four models. This time, we do lesser number of trades as we only trade every 10 minutes (so only every tenth prediction for test data is used). The trading efficiency (profit per trade) is similar to before, or even higher, however the profit factor falls. All the reasoning from the 2-minute case directly applies to this scenario as well.

6 Further Research and Conclusion

In this thesis, we looked at a machine learning problem in quantitative finance both from a theoretical and practical perspective. We defined the methodology for a time series classification problem with class imbalance. The four models we looked at were increasing in complexity, but decreasing in interpretability. Looking at the empirical results, we observe that the average loss on the test set decreased as we took more complicated models. However, this improvement in test loss didn't bring any considerable improvement in predicting the actual class. For 2 minute predictions, the average F1-Score over the 5 classes in all the models was 0.27, which is 35% above the baseline of 0.2. For 10 minute predictions, this dropped to close to 0.2 for all models but LSTMs, which had an average of 0.25. The alpha generated was confirmed when we implemented a mid-to-mid trading strategy, in both 2 and 10 minute cases. Moreover, it was observed that the Deep Learning classifier performed consistently the best in both prediction horizons due to its tendency to accumulate inventory, which shed some light on how profitability not only depends on the predictive power, but also the type of strategy executed. Since for short-term predictions all the models are equally reliable and adding complexity did not help, we can go with the simplest choice which is the MLR model. However, it should be noted that no feature generation method was used for LSTMs. It was only fed past log returns, and it learnt how to weight past observations, to give equally good or even better results than the other models.

This project was undertaken at Deutsche bank's electronic Rates trading team and was aligned with the bank's endeavour for market-making the Gadget. Having proved short-term and long-term alpha in this market, the models produced in this thesis can be coupled with other trading signals for a careful 'mid' selection with a bid-ask spread built around it. Moreover, since this market is fundamentally driven by macroeconomic factors, we suspect that the alpha can be boosted if we also use alternative data such as prices of German futures contracts (Bobble, Bund, Schatz), prices of German government bonds (varying maturities), sentiment data, etc. We could also use some market volatility index and observe its effect on the Gadget spread. We also believe that as the amount of available data points for this product increases, we would be able to increase the prediction horizon even further, and expect different models to show differences in predictive power.

We believe that the models can be further improved using a cost-sensitive approach [29]. Even though in the last section we witnessed that the models learn the hierarchy of the classes, we can still define the loss function in a way that the models penalise different misclassifications in different ways. For example, misclassifying a '1' as '2' wouldn't hurt the investor much, but misclassifying it as a '5' would be a blunder. This can be taken into account by defining a custom CCE loss function (different from a class weighted CCE loss function).

One of the key challenges in machine learning is model interpretability. While we can easily interpret the parameters for a Multinomial Logistic Regression model, it is very difficult to comment about the parameters of an LSTM with more than 10,000 parameters to train. There are some techniques in current literature like using Surrogate models relying on local linearity of models, and Shapley values from cooperative game theory [30] which use a combinatorial approach to explain the decision of a neural network. However, a major breakthrough in this domain still awaits and so model interpretability in machine learning remains to be an open problem.

References

[1] Nagel, Joachim Penna, Raul. (2016). Markets Committee Electronic trading in fixed income markets. 10.13140/RG.2.2.33023.66724.

[2] M. Leung, H. Daouk, and A.Chen. Forecasting stock indices: A comparison of classification and level estimation models. *International Journal of Forecasting*, 16(2):173190, 2000.

[3] Dixon, Matthew Klabjan, Diego Bang, Jin. (2017). Classification-Based Financial Markets Prediction Using Deep Neural Networks. *Algorithmic Finance*. 10.2139/ssrn.2756331.

[4] Engel, J. (1988), Polytomous logistic regression. *Statistica Neerlandica*, 42: 233-252. doi:10.1111/j.1467-9574.1988.tb01238.x

[5] Hoerl, Arthur E. and Kennard, Robert W. Ridge regression: Biased estimation for non-orthogonal problems. *Technometrics*, 12(1):8086, February 1970.

[6] Hsieh, David, Nonlinear dynamics in financial markets: Evidence and implications, *Financial Analysts Journal*, Vol. 51, July-August 1995, pp. 5562.

[7] Breiman, Leo. *Random Forests*. Machine Learning, 45(1):532, October 2001a. ISSN 0885-6125.

[8] Hornik, Kurt. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251257, March 1991. ISSN 0893-6080.

[9] L. Kaastra and M.S.Boyd. Forecasting futures trading volume using neural networks. *Journal of Futures Markets*, 15(8):953970,1995.

[10] 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):19291958, January 2014. ISSN 1532-4435.

[11] Hochreiter, Sepp and Schmidhuber, Jurgen. Long short-term memory. *Neural Computation*, 9(8): 17351780, November 1997. ISSN 0899-7667.

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

[13] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, Vol. 22, No. 3. (Sep., 1951), pp. 400-407.

[14] Diederik P. Kingma and Jimmy Lei Ba. Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations*, pages 113, 2015.

[15] Liyuan Liu, Haoming Jiang, Pengcheng He, WeizhuChen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han.2019a. On the variance of the adaptive learning rate and beyond.arXiv preprint arXiv:1908.03265.

[16] P. Domingos, A unified biasvariance decomposition and its applications, *Proceedings of the 17th International Conference on Machine Learning, San Francisco*. 2000, pp. 231238.

[17] X. Wu and J. M. Perloff. GMM estimation of a maximum entropy distribution with interval data. *Journal of Econometrics*, 138:532?546, 2007.

[18] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[19] Breiman, Leo, Jerome Friedman, Richard Olshen, and Charles Stone. 1984. Classification and regression trees. *Boca Raton: CRC Press*, ISBN 9780412048418.

[20] Louppe, Gilles. 2014. Understanding Random Forests: From Theory to Practice. ArXiv Preprint: 1407.7502

[21] C. Gini. Variabilit e mutabilit. Reprinted in Memorie di metodologica statistica (Ed. Pizetti E, Salvemini, T). *Rome: Libreria Eredi Virgilio Veschi*, 1, 1912.

[22] Zhou, Zhi-Hua. (2012). Ensemble Methods: Foundations and Algorithms. 10.1201/b12207.

[23] T. G. Dietterich. Ensemble methods in machine learning. *In Multiple classifier systems, pages 115*. Springer, 2000b.

[24] Breiman, Leo. Bagging predictors.Machine Learning, 24(2):123140, August 1996. ISSN 0885-6125.

[25] T. K. Ho. The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions*, 20(8):832844, 1998.

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

[27] Olah, Christopher. Understanding LSTM networks. http://colah.github.io/posts/ 2015-08-Understanding-LSTMs/, 2015. Accessed: 2019-09-09.

[28] Ruder, Sebastian.An overview of gradient descent optimization algorithms.arXiv preprint-arXiv:1609.04747 (2016).

[29] Ling, C.X. & Sheng, V.S. 2010. Cost-sensitive learning. *In Encyclopedia of Machine Learning*. Springer, 231235.

[30] Lundberg, Scott and Su-In Lee. A unified approach to interpreting model predictions. ArXiv abs/1705.07874 (2017)

A Appendix: 2-minute classification reports

Actual \ Predicted	1	2	3	4	5	Support
1	324	448	239	281	171	1463
2	259	764	516	579	328	2446
3	174	623	444	555	306	2102
4	201	651	488	662	426	2428
5	87	269	223	396	438	1413
Precision	0.31	0.28	0.23	0.27	0.26	
Recall	0.22	0.31	0.21	0.27	0.31	
F1-Score	0.26	0.29	0.22	0.27	0.28	

Actual \ Predicted	1	2	3	4	5	Support
1	383	466	163	257	209	1463
2	307	757	377	620	391	2446
3	228	582	402	533	369	2102
4	250	616	405	696	472	2428
5	116	232	171	402	497	1413
Precision	0.30	0.29	0.26	0.28	0.26	
Recall	0.26	0.31	0.19	0.29	0.35	
F1-Score	0.28	0.30	0.22	0.28	0.30	

Actual \ Predicted	1	2	3	4	5	Support
1	271	529	119	299	245	1463
2	198	821	236	696	499	2446
3	131	642	227	625	477	2102
4	147	667	216	748	650	2428
5	64	253	100	389	607	1413
Precision	0.33	0.28	0.25	0.27	0.24	
Recall	0.19	0.34	0.11	0.31	0.43	
F1-Score	0.24	0.31	0.15	0.29	0.31	

Actual \ Predicted	1	2	3	4	5	Support
1	353	498	205	200	207	1463
2	264	814	486	486	396	2446
3	188	627	466	456	365	2102
4	215	674	456	557	526	2428
5	106	339	141	304	523	1413
Precision	0.31	0.28	0.27	0.28	0.26	
Recall	0.24	0.33	0.22	0.23	0.37	
F1-Score	0.27	0.30	0.24	0.25	0.30	

Figure 24: Classification Reports for 2 minutes prediction : (i) MLR (ii) RF (iii) DNN (iv) LSTM

B Appendix: 10-minute classification reports

Actual \ Predicted	1	2	3	4	5	Support
1	20	621	267	595	29	1532
2	8	739	451	853	32	2083
3	6	921	575	1233	19	2754
4	6	718	491	1132	33	2380
5	6	257	223	572	41	1099
Precision	0.43	0.23	0.29	0.26	0.27	
Recall	0.01	0.35	0.21	0.48	0.04	
F1-Score	0.03	0.28	0.24	0.33	0.07	

Actual \ Predicted	1	2	3	4	5	Support
1	114	629	216	384	189	1532
2	96	783	388	588	228	2083
3	92	982	532	839	309	2754
4	83	753	504	713	327	2380
5	37	279	192	344	247	1099
Precision	0.27	0.23	0.29	0.25	0.19	
Recall	0.07	0.38	0.19	0.30	0.22	
F1-Score	0.12	0.28	0.23	0.27	0.21	

Actual \ Predicted	1	2	3	4	5	Support
1	10	702	325	460	35	1532
2	4	793	566	663	57	2083
3	1	1024	719	958	52	2754
4	1	773	604	915	87	2380
5	1	310	241	475	72	1099
Precision	0.59	0.22	0.29	0.26	0.24	
Recall	0.01	0.38	0.26	0.38	0.07	
F1-Score	0.01	0.28	0.28	0.31	0.10	

Actual \ Predicted	1	2	3	4	5	Support
1	174	477	303	474	104	1532
2	123	631	465	752	112	2083
3	120	777	661	1066	130	2754
4	93	555	550	1039	143	2380
5	80	234	180	435	170	1099
Precision	0.29	0.24	0.31	0.28	0.26	
Recall	0.11	0.30	0.24	0.44	0.15	
F1-Score	0.16	0.27	0.27	0.34	0.19	

Figure 25: Classification Reports for 10 minutes prediction : (i) MLR (ii) RF (iii) DNN (iv) LSTM

Classification-based Prediction of Gadget Time Series Using Machine Learning

GRADEMARK REPORT

FINAL GRADE

/0

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55