Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2019

---


| | |
|---|---|
| Project Title: | **Recurrent Networks for Factor Models in Time Series** |
| Student: | **Tim Hung Wu** |
| CID: | **01195701** |
| Course: | **EIE3** |
| Project Supervisor: | **Dr. Carlo Ciliberto** |
| Second Marker: | **Professor Yiannis Demiris** |

## Abstract

*This project explores the effectiveness of certain recurrent neural network architectures with long short-term memory units (LSTM RNNs) in the forecasting of stock prices for companies in the STOXX Europe 600 Industrial Goods and Services Index (SXNP). In particular, the project focused on attempting to identify whether certain attributes of a company's fundamentals data, such as book value and annual sales, had significant influences on its stock price. The main emphasis of this project was to implement a complete tool that would provide all the functionalities required to take this machine learning problem from the initial raw dataset to a reasonable final forecast result. As such, the various processes required to develop a solution to a machine learning problem such as data preparation, network structuring, optimisation etc. were studied and analysed. A significant proportion of the project focused on processing the raw data into a suitable and effective format that could be used as the input of a Keras model. Additionally, by tuning various hyperparameters and experimenting with multiple loss functions and classifications methods, attempts were also made to optimize and improve the performance of the model. Finally, the stock weightings provided by the model were used to construct a portfolio to simulate a real world scenario. Its performance was then compared to the naïve diversification heuristic over a time period consisting of unseen test data.*



## Acknowledgements

A special thanks goes to Dr. Carlo Ciliberto of Imperial College London for taking the time to provide expertise about developing and designing a machine learning experiment. Additional thanks goes to Dr. Marco Bianchi and Michele Ragazzi of Giano Capital for providing the financial background knowledge to help bring this project to a real world application as well as their development team for providing the raw data.

## Contents

# 1. Introduction

## 1.1. Project Motivation

Public corporations are required to periodically report various fundamental financial data such as income statements and balance sheets. This is legally required in most countries for tax purposes and also provides key performance indicators for stakeholders and investors into the general financial health of a corporation. The owners of these companies are a mixture of the general public and ownership is fractionally divided via shares of a company's stock. These shares are able to be freely bought and sold and the process of valuing them has become of predominant interest to investors. Consequently, the expectation of how these valuations will deviate and change in the future is interesting as it provides the opportunity for financial gain by taking the profit through a difference in price.

Academic research has demonstrated that some key factors of the reported financial data have historically had high correlation with the returns enjoyed by the stock of a company. An example of a popular methodology is analysing the asset turnover ratio ($sales/assets$), which approximately indicated the efficiency with which a company deploys its assets to generate sales.

The process of automating the identification of features which are able to deliver profitable investment opportunities falls into the domain of systematic factor investing. Increasingly, this has become more popularised because of the exponential technological improvements in data storage and computational power. Additionally, the digitalisation of data allows it to be more standardised and of higher quality, further lowering the limitations boundary for automation.

## 1.2. Problem Definition

Using the SXNP dataset supplied by Giano Capital, this project aims to establish the key fundamentals that affect the returns of a company's stock price. This is achieved by developing algorithms that can assist in predicting the future prices of the stock. The algorithms utilise several inputs such as book value, sales and profit of the specified company to train and learn a function that correlates with the price. The performance is then benchmarked against an industry standard to see if the algorithms have achieved any outperformance compared to traditional models. This problem falls in the scope of machine learning, which has been frequently used to produce solutions in data-driven problems in recent times. Focus will be directed at RNNs with LSTM units. We will choose well established and tested techniques used in literature to extract features from the dataset and attempt to learn patterns in the data. We will then tune the hyperparameters to optimise the model, and evaluate its potential performance in a pseudo-generated real world application.

### 1.2.1 Intended Deliverables

- A cleaned dataset capable of being trained on.
- A piece of software that implements a predictive model.
- Analysis into the performance of the model.
- Understanding of prior research in the area and how to design a machine learning experiment.
- A report and poster summarising the project and its experimental results.

## 1.3. Importance in Industry

Outperformance, defined by generating excess returns above the market average, is termed as alpha ($\alpha$) in the financial industry. This metric is widely used to assess and compare the abilities of actively managed funds. The process from which alpha is generated is termed "signal generation" and determines when a security, such as a stock, should be bought or sold. However, as more people utilise a certain signal, it becomes weaker as it, by definition, tends towards becoming part of the market average. The objective then evolves to be consistently producing new, stronger signals. As mentioned in section 2.1, there are various methodologies people use to generate signals. Although there is no limit to how complex a signals can theoretically be, people tend to focus on just a handful of parameters for ease-of-use and understanding. This begs the

question of whether a machine learning model, with increased numbers of inputs, are able to assist in developing or even create complex methodologies that result in strong signals.

As such, this project is of interest to money managers as a means from which to develop alternative and new solutions to the problem of generating new signals to produce alpha. Additionally, the consequence of being able to produce improved returns are ultimately handed to the investors. This project focuses particularly on the fundamental components of companies, which tends to be in the scope of long term investment. The investors in these areas often include institutional clients such as pension funds meaning that these potential improvements are eventually passed down to working individuals, who do not possess the ability to generate this income on their own for their future retirements.

### 1.4. Software Usage

Python will be the language used for software development given the extensive number of libraries that offer the opportunity for the fast development required for the project. The key packages that will be used are Pandas and Keras, which are data manipulation and machine learning tools respectively. Correctness of the code is a necessity whereas performance is treated as a luxury. The report will be written in LaTeX.

All software development will be within the student's personal Google drive, with external view access granted to only those that are necessary. A web based environment, Jupyter Notebooks[1], will be used in order to develop a clear and structured front end document that can be easily used and interacted with by users. Consequently, the ipynb file format will be used for the source. Data is stored as csv files.

Cloud services will be used for the computational power and elements of data storage. Google Colaboratory offers good support for the Notebook and has previously been used as part of the EE3-25 Deep Learning module and so has a low barrier to entry. Given the fairly small size of the dataset (at approximately 2.5GB), the required computation power required is estimated to be well below the maximum threshold of the environment.

### 1.5. Report Structure

This report begins with a background outline to provide more colour into the problem by highlighting alternate approaches to investing and uses of the LSTM RNN. It will then establish the software deliverables and requirements expected from this project. This will then be followed by a detailed description of the iterative design processes used to construct models for the problem, with comments about how they were modified from the initial specification to build the final solution. Afterwards, the implementation and testing methodology for the software will be explained, with highlights of any interesting challenges. Then, the system will be benchmarked against the self-developed metrics before a final section of evaluation and reflection of the key discoveries and findings learned throughout the project and the direction in which the project could be further developed in the future.

## 2. Background

### 2.1. History of Investing Methodologies

The fundamental question that the project is attempting to answer is what drives stock price and their underlying returns. This has been tackled through a whole range of varied approaches ever since shares of stock were first offered to general public in the early 1600s by the Dutch East India Company[27]. In 1934, Benjamin Graham et al. introduced the first formal framework for security analysis[11]. In his book, he describes the differences of market behaviour in the short and long run. Graham observed that fear, greed and emotions were the main drivers of short-term market fluctuations which can cause discrepancies between the pricing and true value of a company's stock. However, over long periods of time, Graham perceived that a company's fundamentals are the critical price driver, which causes the market pricing of the stock to converge with its true value. These observations formed one of the first formal distinctions between speculation and investment.

The first formal model[34] of analysing stock returns was introduced by Treynor et al. in the 1960s. It was called the Capital Asset Pricing Model (CAPM) and focused on the idiosyncratic and systematic risk of an asset relative to the market. The formula for pricing a security is as follow: $E(R_i) = R_f + \beta_i(E(R_m) - R_f)$ where $E(R_i)$ is the expected return of investment, $R_f$ is the risk free rate, $\beta_i$ is the beta of the investment, $E(R_m)$ is the expected return of the market. In other words, the expected return of a stock can be viewed as a function of its correlation to the market.

An alternative theory[26] about what drivers stock returns was later proposed by Ross in 1976. Arbitrage pricing theory (APT) postures that the expected return of a financial asset can be modelled as a function of various macroeconomic factors. Ross is often accredited with popularising the term "multi-factor models." Crucially, APT performed better than CAPM because it did not explicitly state a set of fixed factors. Instead, factors were assumed to vary across markets and differ over time. The challenge then boiled down to identifying the factors that best describe a specific region at a point in time.

Arguably, the most utilised factors for analysis are fundamental factors, which capture the characteristics of a stock. One of the most well known attempts in this space was derived from Fama and French in the early 1990s. Their model[8] explained US equity market returns from three factors: size, value and correlation as mentioned in the CAPM model. The fundamental objective of factor investing is to achieve enhanced diversification, above-market returns and reduced risk exposure. The numbers of quantifiable factors are vast but typically, are broken down into five predominant categories:

- Value: The stocks usually trade at low prices relative to their fundamental indicators. These are identified via indicators which are typically ratios such as $\frac{bookvalue}{price}$, $\frac{earnings}{price}$ or $\frac{dividend}{price}$. When these ratios are significantly higher than the average company within the same sector, the company can be classified as having value stock. It should be noted that average ratios vary between sectors and so cross-industry comparisons cannot be made.

- Size: Eun et al. demonstrate in their research[7] that small capitalisation stocks tend to exhibit statistically significant returns and diversification than large capitalisation stocks when allocated in international funds. Capitalisation is defined as $capitalisation = no.share * shareprice$. Small-caps are defined to range from valuations of $300 million to $2 billion.

- Momentum: Stocks that have exhibited outperformance in the past tend to follow the same trend in the future. Strategies involving momentum often involve analysing, ranking and allocating to the highest returning stocks within a specified time frame such as one year or three years.

- Quality: These stocks are typically categorised by certain fundamentals such as low debt, average earnings with low standard deviation i.e. high stability and consistent asset growth i.e. the derivative of book value with respect to time is fairly uniform. Common financial metrics used to help identify this include the return on equity ($ROE = \frac{NetIncome}{ShareholderEquity}$) or the debt-to-equity ratio ($\frac{D}{E} = \frac{TotalLiabilities}{TotalShareholderEquity}$.)

- Volatility: Empirical research by UBS[20] suggests that low volatility stocks earn greater risk-adjusted returns than other stocks, with particular outperformance during stressed market environments. These are often ranked and selected by measuring the standard deviation of a stock's return.

## 2.2. Long Short-Term Memory Networks

### 2.2.1 Overview

LSTM networks are derived from RNN models, which were first introduced into literature[29] in 1986 by Rumelhart et al. Rumelhart was a psychologist by training and the initial objective was to provide an alternative framework for understanding cognitive perception. The fundamental problem was to highlight that human perception and learning were formed through temporal experience and so a model was required that could process and partially remember sequential data. Rumelhart mathematically modelled this as $h_t = f_W(h_{t-1}, xt)$ where $h$ is the state, $x$ is the input with respect to the time interval $t$ and $f$ is a function generated by the parameters $W$ as determined by the configuration of the LSTM cell. This was in contrast with more traditional neural networks, such as the multilayer perceptron (MLP), are structured in a feedforward fashion that do not share weights, meaning that there are no cycles formed within the nodes.

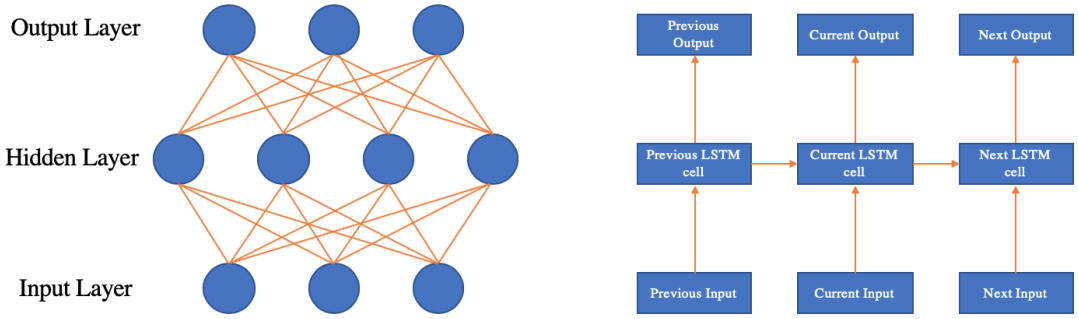



Figure 1: A classical MLP (LHS) simply generates an output with respect to its input whereas in the RNN (RHS), the output is governed by the input and previous input.

The use cases of RNNs have since expanded to multiple, practical domains that heavily use sequential data, such as text generation, visual recognition and time series prediction. Part of the advantages offered by RNNs is the various topologies in which it can be structured. The most vanilla case is with a direct one-to-one mapping. However they often change to many-to-one, many-to-many or one-to-many to suit the task at hand.

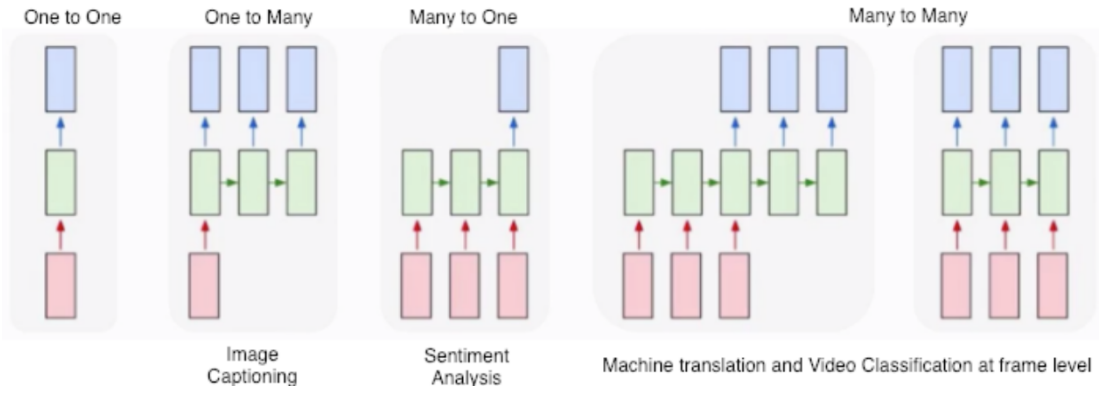



Figure 2: Numerous varying topologies for specific sequential data tasks.

### 2.2.2 The Exploding and Vanishing Gradient Problem in RNNs

One of the issues with RNNs stems from the inherent property of operating with a sequential nature. The time difference between multiple cells often causes issues during backpropogation involving repeated multiplication of the gradient signal. This means that if the partial derivatives of the error gradient are either too large or small, they can explode or vanish

respectively during the procedure. Exploding gradients cause the learning to diverge whereas vanishing ones can cause the learning rate to become very slow and eventually stop, as it tends to 0.

*Proof.* Let $x, y, h, \theta, f$ be the input, output, hidden state, weights and basis function respectively: (Bengio et al.)

$$h_t = \theta f(h_{t-1} + \theta_x x_t)$$
$$y_t = \theta_y f(h_t)$$

The loss function is then:

$$\frac{\partial E}{\partial \theta} = \sum_{t=1}^{S} \frac{\partial E_t}{\partial \theta}$$
$$\frac{\partial E_t}{\partial \theta} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial \theta} \qquad \text{(where } k \text{ are the layers before time } t\text{)}$$

The third term of the chain, $\frac{\partial h_t}{\partial h_k}$, is a product of Jacobians such that:

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} \theta^T diag(f'(h_{i-1}))$$
$$||\frac{\partial h_i}{\partial h_{i-1}}|| \leq ||\theta^T|| ||diag(f'(h_{i-1}))|| \leq \gamma_\theta \gamma_f \qquad \text{( where } \gamma \text{ are the constants that upper bound the two terms)}$$
$$||\frac{\partial h_i}{\partial h_{i-1}}|| \leq (\gamma_\theta \gamma_f)^{t-k}$$

This implies that if $\gamma_\theta \gamma_f > 1$ or very small, the gradient will explode or vanish given a large $t - k$. LSTMs were first introduced by Hockreiter et al.[15] in 1997 and were effective at correcting the vanishing gradient problem. LSTMs use gating, also known as component-wise multiplication, which allows for more control over the gradient flow and thus enables better preservation of long term dependencies. Three gates exist in the LSTM cell forget, input and update, which are mathematically defined as following:

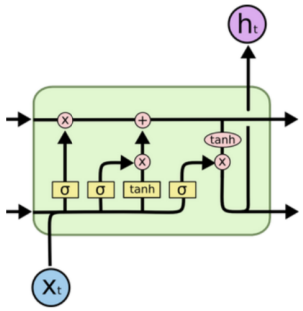



Figure 3: Breakdown of a LSTM cell depicting the input, forget and update gates.

$$\text{Forget gate} : f_t = \sigma(W_f(h_{t-1}, x_t) + b_f)$$
$$\text{Input gate} : i_t = \sigma(W_i(h_{t-1}, x_t) + b_i)$$
$$\text{Update gate} : u_t = \sigma(W_u(h_{t-1}, x_t) + b_u)$$

As explained by Olah[21], the forget gate controls the fraction of how much of the previous cell's information should be retained. Next, a decision needs to be made about what new information should be stored in the current cell. This is achieved by the input gate. Additional to this, a candidate cell, $\tilde{C}$, is produced and scaled appropriately with a $tanh$ layer. By combining these two it is possible to create an update to the state.

$$\tilde{C}_t = tanh(W_C(h_{t-1}, x_t) + b_C)$$
$$C_t = (f_t \times C_{t-1}) + (i_t \times \tilde{C}_t)$$

The block output will then be a filtered version of our newly updated cell state, with the filter being the update gate. The sigmoid layer in the update gate decides what parts of the cell state will be output.

$$y_t = u_t \times tanh(C_t)$$

Although LSTMs offer a method of counteracting vanishing gradients, the problem of exploding gradients still exists. Predominately, there are two main strategies to deal with this: gradient clipping and gradient normalisation, both introduced by Pascanu et al.[22]. Gradient clipping involves involves reducing a gradient's components if it exceeds a specified threshold, $\tau$. The components form part of a matrix defined as $g_{ij}$ such that the updated matrix element values are $\hat{g_{ij}} = max(-\tau, min(\tau, g_{ij}))$. This is then used for the weight update.

Gradient normalisation involves rescaling the gradient rather than reducing its with an absolute value clip. This occurs when the norm $||g||_2$ goes over a threshold $\tau$. The gradient is then readjusted as such: $\hat{g} = g(\tau/||g||_2)$. Reimers et al.[24] explored both of these methods in their paper and concluded that gradient normalisation produced statistically significant performance improvements whereas gradient clipping did not in the context of sequential tagging.

### 2.2.3 Loss Functions

The loss function is important in the feedback mechanism of general neural networks and vary given the task at hand. They typically fall into two categories depending on the problem itself: classification and regression. These domains treat discrete and continuous value labelling problems.

For regression problems, two of the most common loss functions used are mean squared error (MSE) and mean absolute error (MAE). Mathematically, there are defined as following: $MSE = \frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2$, $MAE = \frac{1}{N}\sum_{i=1}^{N}|x_i - y_i|$. The difference in the behaviour of the loss functions can be observed with large errors, with MSE putting much more emphasis on learning and correcting from those data points. Interestingly, Janocha et al.[17] suggested that these loss functions also perform surprisingly well in deep net classification objectives as a means of providing a probabilistic interpretation in terms of expected misclassification.

A popular example of a classification problem is the a binary classifier. These take input data and assign one of two possible output values. An example would be the binary cross-entropy loss function, expressed as $\frac{1}{N}\sum_{i=1}^{N} log(p(y_i)) - (1 - y_i)(log(p(1 - y_i)))$ where $y_i$ is the target label and $p(y_i)$ is the probability of a correct prediction. This error function is then derived from the probability of a correct result, with it being an exponentially larger loss for a weaker predictor.

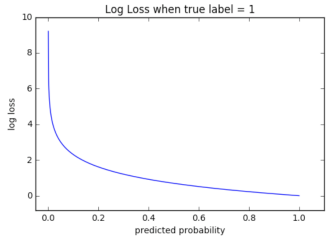



Figure 4: Graph showing the binary cross entropy loss function given the probability of a correct prediction

### 2.2.4 Cross Validation

An issue with LSTM networks arises when using classical cross validation methods. K-fold cross validation involves dividing the data into $k$ folds before assigning the initial fold as the test data and the rest as the training data. After each iteration of training, the test data assignment is then incremented until its position has permeated through all of the positions. However, with sequential data, this method causes disjoints in the training data since training for fold $n + 1$ would involve information from $n - 1$ rather than $n$. In 2015, Bergmeir et al. introduced a new method[4] to handle cross validation for time series data. Their system involved a forward chaining concept such that training data is never interrupted. Although it resolves the issue, the generalisation error is then affected by the change in training data size. Below is a heuristic for the method:

- Divide the data into $k$ folds.
- 1st iteration: Training on fold 1. Testing on fold 2.
- 2nd iteration: Training on fold 1, 2. Testing on fold 3.
- Repeated to $k-1$th iteration: Training on fold 1, 2 ... $k-1$. Testing on fold $k$ (last fold).

An alternative solution[23] was also proposed by Racine. He observed that what was truly needed is independence between the training and test data sets. This can be approximated by removing adjacent folds which will be highly dependent on and correlate with the test fold. This needs to be performed on both sides of the test fold as dependence is symmetric when moving forwards and backwards in time. This approach is known as $hv$ cross validation where $v$ is the test fold and $h$ are the number of observations on each side of the test fold. Below is a heuristic of the method:

- Divide the data into $k$ folds. Assume $h = 1$
- 1st iteration: Testing on fold 1. Fold 2 is discarded. Training occurs on the rest of the folds.
- 2nd iteration: Training on fold 2. Folds 1 and 3 are discarded. Training occurs on the rest of the folds.
- 3rd iteration: Training on fold 3. Folds 2 and 4 are discarded. Training occurs on the rest of the folds.
- Repeated to $k$th iteration Testing on fold $k$ (last fold). Fold $k-1$ is discarded. Training occurs on the rest of the folds.

This does not introduce the generalisation error as in Bergmeir's method. However, it is more complex to measure the $h$ required to approximate a good independence and there is still a consistent inefficiency in data usage given the discarded folds are not trained or tested with. This efficiency is only better than Bergmeir when $h \leq 2 \times (\frac{k^2}{2}) = k^2$. Both methods will be considered during the project to determine which one is optimal.

### 2.2.5 Applications

An exciting application of LSTM networks can be found in one of Google Deepmind's project called Deep Recurrent Attentive Writer (DRAW). In their paper[12], Gregor et al. outline the architecture for a network capable of image generation. They use a variational autoencoder (VAE) as the generative model. This encodes a raw image into a lower dimensionality vector, which forces the encoder to learn the features of the image. LSTMs and attention mechanisms are then used to iterate this encode-decode procedure. This network was used to generate digits after being trained on the Modified National Institute of Standards and Technology (MNIST) database.

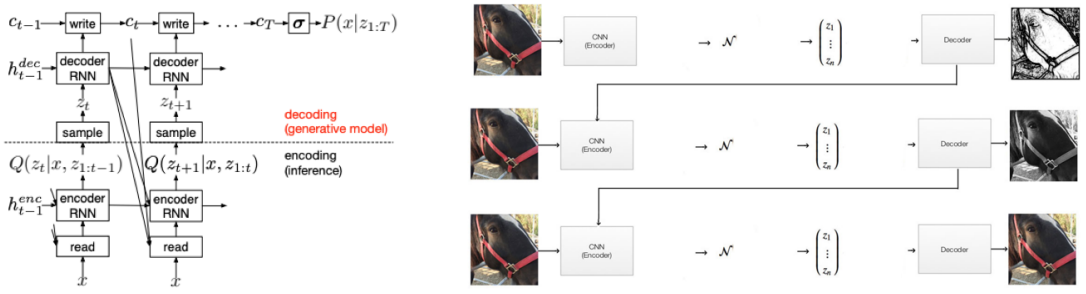



Figure 5: LHS: DRAW network from Google Deepmind, featuring an attention mechanism formed by an encoding and decoding RNN (with the LSTM architecture). RHS: Simulation of how each iteration improves on the previous when using the network.

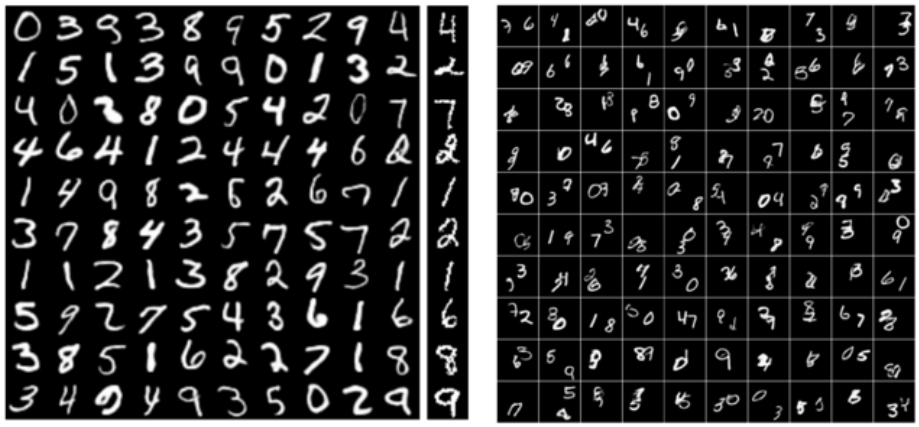


Figure 6: LHS: Single digits generated from the MNIST database, with the right-most column depicting original images from the database. RHS: DRAW's attempts at generating double digit values.

## 2.3. Review of Machine Learning Research in Investing

### 2.3.1 Overview

Machine learning research in finance applications often span two domains; applied research in financial firms often has a more pragmatic approach and a more theoretical approach is usually observed in academic institutions. Man Group, a quantitative investment management firm, have been conducting research[25] within the machine learning domain for decades but have only in the past five years begun actively applying models in a live environment. This is because there have only been enough developments recently in infrastructural support to help scale these models onto a production level. They separate their approaches into two major categories:

- Deep Learning: Models using artificial neural networks which are trained on large data sets to identify special features. The traditional application of these techniques stem from areas such as image recognition but can also be used to learn predictive patterns in financial datasets.
- Natural Language Processing: This mainly involves the automation of analysing written reports, such as a company's financial statements or a shareholder letter. Techniques involve assigning numerical values that measure the sentiment of these reports, which are often correlated with the price. This provides an unbiased and deterministic method for evaluating stocks.

The model used in the project falls under the deep learning categorisation and is specified in section 2.2. A few questions come to surface regarding the technical challenges of the implementation. Some are listed below and answered by various approaches recommended in numerous papers:

- Normalisation: Standardising the features of a model is often important because there is a need to compare measurements which have different units.
- Missing Inputs: A way of handling incomplete datasets is needed. Methodologies often fall into the two categories of either regenerating or masking the missing values.
- Encoding Additional Information: Adding human derived formulas such as $\frac{sales}{assets}$ can be seen as generating human bias or assisting the model with classification depending on perspective.
- Cross Validation (Section 2.2.4): A problem with time series structures is the autocorrelation of data points. This means that typically cross validation methods such as leave-one-out introduce issues by missing information in the series and leaking information from the future.

### 2.3.2 Initial Usage

One of the earliest published papers[2] regarding the subject matter was written by Ahmadi in 1990. It was primarily focused on sourcing an alternate approach to generating a more comprehensive system to analyse factors in the APT model. This was done as following: Given a $N \times T$ matrix, $r_{N,T}$ is defined as the return of asset $N$ at time period $T$. For this matrix, there is a smaller one of $K \times T$, where $f_{K,T}$ represents the $K$th factor score for time period $T$. Thus, the return of a given asset is: $R_N = E_N + b_{N,1}F_1 ... + b_{N,K}F_K + e_N$ where $E_N$ is the expected return, $e_N$ is a random zero mean error term and $b_{N,K}$ are the estimated coefficients for each factor. Using the coefficient estimates, it is possible to employ another variable, $\lambda$, to determine whether it is statistically significant and thus whether the factor is significant. This is done with the following equation: $E_N = \lambda_0 + b_{N,1}\lambda_1 ... + b_{N,K}\lambda_K$.

The problem Ahmadi wanted to resolve was the heavy dependency on the sample that factor analysis relied on. There was a significant chance that, given 42 groups of 30 stocks each, different significant factors would be discovered in each group. He therefore employed a neural network so that, rather than optimising and identifying the statistical significance in each feature individually, the patterns stem from analysing the whole vector space. Ahmadi provides some details about the network, highlighting that the inputs used were the unemployment and inflation rates, rate of returns of assets, rate of change in the stock market and the change in gross national product (GNP). He also specifies that the inputs were normalised, two hidden layers offered the best optimal performance and the learning plateaued after 5000 iterations.

### 2.3.3 Formalisation of Experimentation

As machine learning research became more formalised, the structure of experimentation was more defined. Kim published a paper[16] in 2003, focused on understanding how support vector machines (SVMs) fared at predicting the Korea Composite Stock Price Index (KOSPI). Performance, $P$, was measured using the following equation:

$$P = \frac{1}{m}\sum_{i=1}^{m} R_i \qquad \text{(i = 1, 2, ... m)}$$

where $R_i$ is the prediction result for the $i$th trading day as defined by:

$$R_i = \begin{cases} 1, & \text{if } f(x) = y \\ 0, & \text{otherwise} \end{cases}$$

where $f(x)$ is the prediction and $y$ is the actual value.

He tested the structure with both the polynomial and Gaussian radial basis functions for the kernel. Most importantly, he benchmarked the performance against two other models: a three layered backpropogation network (BP) and case based reasoning model (CBR) using nearest neighbour. However, there are no specific details about the architectures of these models.

The best prediction performances of SVM, BP, and CBR (hit ratio: %)

| | SVM | BP | CBR |
|---|---|---|---|
| Training data | 64.7526 | 58.5217 | |
| Holdout data | 57.8313 | 54.7332 | 51.9793 |

Figure 7: Kim's results of SVM compared to BP and CBR.

### 2.3.4 Methodologies

An interesting alternative approach[3] is demonstrated by Alberg et al. who tackle the problem from a different perspective and adapts the classical problem from predicting prices to predicting how the fundamentals of a company will change. These

predictions could then be used in turn to indirectly predict the price at that future time point. Alberg then backtested this idea, accessing existing fundamentals data as the 'future predicted fundamental', equating its price and evaluating the performance of the model. The paper goes into great depth about the set up of the experiment. A key area of interest is the methodology used to establish the binary output classifier:

$$f(x) = \begin{cases} 1, & \text{if } x > \frac{n+1}{n}th \text{ term} \\ -1, & \text{if } x < \frac{n+1}{n}th \text{ term} \end{cases}$$


Figure 8: (LHS) The overview of how the model functions. (RHS) Backtesting results from Alberg with various factor models between 2000-2016. Clairvoyance x-axis equates to the time period difference between the current and future time in the model.

Rather than predicting the prices directly, which could rely on a multitude of factors outside just fundamental data, the model is structured so that it is interested in learning about what makes a company outperform the median return. A weakness of this classifier is that, in economic downturns, the median return could still be negative and so the model is not learning what fundamentally drives 'good' returns in an absolute sense. It also does not predict the degree to which the chosen stocks will outperform. However, this is in interesting structure for this project given the relative nature of the evaluation metric selected as mentioned in section 4.7.

Additionally, normalisation of the inputs is carried out with two methods. The first involves ranking the performance of stocks against a certain ratio, such as $\frac{BookValue}{MarketCapitalisation}$. The percentiles of their rankings are then inputted alongside the raw scores in order to produce a more comparable set of features between various stocks. The second involves transforming all data points into a specified domain for ease-of-use to the learning algorithm whilst maintaining the same relative fundamental values between companies. Alberg achieves this by dividing the input feature by the L2 norm of all the features ($x_i \rightarrow \frac{x_i}{||x||}$). This is in contrast with two of the more popular methods of normalisation: z-score ($x_i \rightarrow \frac{x_i - \mu_x}{\sigma_x}$) where $\mu$ and $\sigma$ are the mean and standard deviation and min-max scaling ($x_i \rightarrow \frac{x_i - x_{min}}{x_{max} - x_{min}}$).

### 2.3.5 Related Works

Groundbreaking work can often be produced and found in competitions. The primarily example for time series datasets was the M4 competition held in June by Makridakis of the University of Nicosia. This was the forth competition of a series which focussed on forecasting for 100,000 time series datasets, spanning different time intervals and domains. Although not directly related to investing, there are various similarities and use of applications. A key objective of the task was to evaluate the performance of machine learning models compared to other statistical methodologies. Makridakis et al.[19] published their findings after the competition. Below are a few of their major findings in the report:

- Combination of methods performed well in the M4 competition - 12 of the 17 most accurate methods were combinations of mostly statistical approaches.

- The best performing approach utilised both statistical and ML features. This method produced the most accurate forecasts as well as the most precise Prediction Intervals and was submitted by Smyl, a data scientist at Uber. According to competition error metric, Symmetric Mean Absolute Percent Error (sMAPE), it was approximately 10% more accurate than the competition benchmark. In the previous M3 competition, the winning method by Makridakis and Hibon was 4% more accurate than the same benchmark.
- The second most accurate method was a combination of seven statistical methods and one ML one submitted jointly by Spain's University of A Coruña and Australia's Monash University.
- The six pure ML methods submitted performed poorly, none of them being more accurate than the benchmark.

Smyl et al. published an overview of their methodology online[33]. In it, he highlights the use of using both neural networks and triple exponential smoothing as part of the forecasting algorithm. This was introduced into literature by Winters[35] in 1960 and involves a weighted moving average, trend and seasonal component. Mathematically, they are described as following:

$$\text{Level} : l_t = a(x_t - s_{t-L} + (1-a)(l_{t-1} + b_{t-1})$$
$$\text{Trend} : b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$$
$$\text{Seasonal} : s_t = \gamma(x_t - l_t) + (1-\gamma)s_{t-L}$$
$$\text{Forecast} : x_{t+m} = l_t + mb_t + s_{t-L+1+(m-1)modL}$$

where $a$, $\beta$ and $\gamma$ are selected coefficients and $L$ is the seasonal length.

Level offers a smoothed version of $x$, Trend observes the local linear trend and Seasonal evaluates the cyclical general nature of the data. The components can be both additive where $b = x_t - x_{t-1}$ or multiplicative where $b = \frac{x_t}{x_{t-1}}$ but multiplicative predictors are seen to be more stable in practice. Smyl used a multiplicative seasonality coefficient. This approach works well independently, but there is no need to assume that the trend component should be linear. Additionally, since each trend is fitted to a respective series, there is no opportunity for cross-series learning. Multiple steps ahead, nonlinear forecasting can be achieved via a neural networks trained on all series.

The general architecture that Smyl relied on was the dilated RNN, first introduced by Chang et al. in 2017. This model helps efficiently resolve issues in multi scale problems without increasing the dimensionality of the network significantly. Multi scale problems require a balance between evaluating the relationship between neighbouring cells whilst also integrating a general, globalised learning context. This is resolved in the model by structuring the cells so that they have dilated skip connections as shown in figure 9.

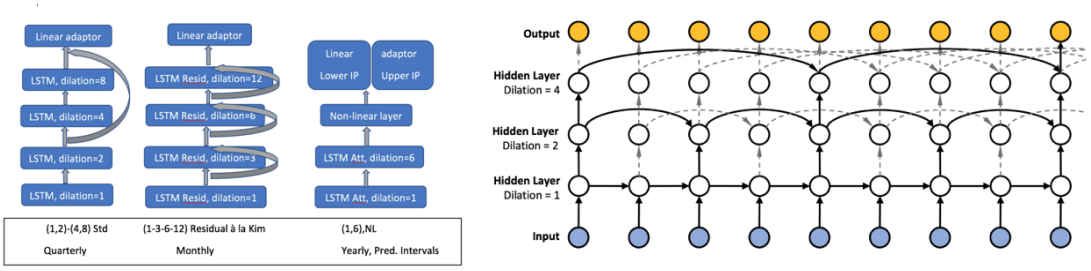



Figure 9: LHS: Different architectures that Smyl used with respect to time horizon of data. RHS: General structure of 3 layered dilated RNN.

### 2.3.6 Summary

Finally, in 2019, a summary of the literature[14] that had been written about financial prediction applications using machine learning was produced by Henrique et al. They constructed a table comprehensively detailing the various attributes and methods used in the papers they reviewed. This provides a very general overview into how the academic field is generally designing experiments for this problem domain.

- Assets Evaluated: Typically stocks or indices. These fall into the same region as the problem tackled by this project. Interestingly, many index models were focused on predicting the prices of the index itself whereas stocks typically offered more dynamic evaluation metrics such as selecting good stocks and calculating their returns.
- Predictive Variables: Predominately technical analysis or prices, with a few fundamentals and one text and news. It is understandable that most models which predict price use technical analysis indicators given that the perceived relationship between these is fairly strong. Technical analysis was designed for price analysis whereas fundamental data looks at the valuation of stocks, which is a slightly different task. However, this project focuses on a long term investment horizon and so is hoped that fundamental features will be more significant in this time domain. The text and news cases were used to analyse sentiment, which often affects price on a very short time horizon.
- Output Prediction: Mainly prices, return or direction, with a couple targeting volatility. Price and returns are the ideal goal but it is interesting to see that a significant number of models use alternate methodologies. Both direction and volatility offer a less demanding output objectives and so could achieve better results overall if trained correctly. This project will analyse a variety of output predictors and compare their performances.
- Methodology: Mostly neural networks, with some SVMs and fuzzy logic. This project uses a RNN with LSTM units and so falls under the set of neural networks. The objective is to explore how LSTM models fare in a different domain, given their primary applications being in image, audio and text generation/recognition.
- Performance Measure: A mixture of return, Sharpe ratio, mean absolute error (MAE) and mean squared error (MSE). MAE and MSE are generally used when classifying the test error of a price prediction. However, the project aims to deliver a real world grounding and so metrics such as Sharpe ratio will be used during the final evaluation. It is very likely that MAE and MSE will be analysed during the experimentation stages.

# 3. Requirements Capture

The key objective for the project is to deliver a tool that will assist in answering the fundamental problem of whether it is possible to predict price changes in stocks based on the fundamental's data of a company. This primary objective is then divided into sub-sectional physical deliverables, as briefly mentioned in section 1.2.1. Fallback plans were established for each deliverable to prioritise certain essential functionalities if problems in software development and debugging cause significant time delays.

## 3.1. Tasks

1. Data Preprocessing: Using methods that have been researched and understood, the datasets supplied by Giano Capital will be cleaned and formatted in an appropriate fashion for use as the inputs of the LSTM model. This is a vital element of the project and it can be argued that the quality of model performance is critically dependent on this deliverable. However, it is important to observe the schedule as this is the initial deliverable and going overtime would put the other components in jeopardy. In a worst case scenario, priorities will be made to certain fundamentals that are theorised to be the most important and the others discarded to reduce workload.

2. Building and Testing Benchmark: The design for a suitable industry-standard benchmark is mentioned in section 4.7. Going forwards, the objective will be to build the tools necessary to compare the benchmark with the output data from the model. Similar to the fallback plan in Data Preprocessing, efforts will be solely concentrated on the key benchmark criteria (the most important being absolute returns) if there are heavy time constraints.

3. Implementing and Testing Model Architecture: Official documentation about the Keras API alongside example notebooks from the EE3-25 Deep Learning module will provide the basis for developing a well structured and coherent architecture. Initially, a simplistic baseline model will be constructed and tested in order to provide a fundamental level from which to develop a better and more sophisticated model. Parameters will be experimented with by applying similar logic to the research methods discussed in section 2.3. If time constraints become an issue, a simpler model will be developed and tested instead, with the baseline model being used in the absolute worst case scenario.

4. Integration Testing and Debugging: The prior three deliverables form components of the system and so will need to be tested when combined to see if the behaviour and results are as intended. This deliverable is flexible as testing can be performed to a wide degree of formality. A rigorous framework will require the construction of test datasets and their intended outputs, accounting for all edge cases that may occur. Consequently, significant time may need to be spend debugging and restructuring the codebase to account for these edge cases. The primary focus is for the development of a robust rather than an entirely correct system. The fallback plan is therefore to solely focus on accounting for errors that may occur in the Giano dataset as ultimately, this is what the model will use to perform analysis.

5. Hyperparameter Optimisation: To improve the performance of the model with respect to the benchmark, various modifications can be applied to areas of the algorithm such as the training method, network depth, learning rate optimisers etc. A significant proportion of time has been allocated to this since it involves the lengthy process of retesting and reevaluating the results and performance given every adjustment made. This final deliverable is arbitrarily long, given the fact that analysis of components can be inexhaustibly extensive and so it will be flexible in the sense that as much experimentation will be performed as allowed for by time.

# 4. Design and Analysis

## 4.1. Overview

Figure 10 depicts the intended diagrammatic pipeline of the tool that was developed. Each of the blocks consists of various functions combined to achieve the intended result. A behavioural driven development (BDD) philosophy was used to develop these subsystems, meaning that functions were implemented based on what was needed and so there was no detailed initial plan of the functions. The following subsections highlight each of the individual blocks. Program flow diagrams can be found in section 5.

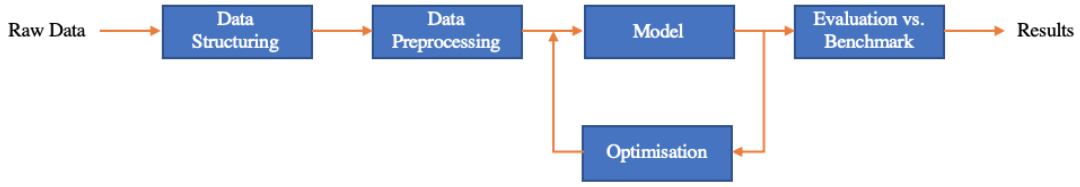



Figure 10: High-level overview of the tool pipeline design

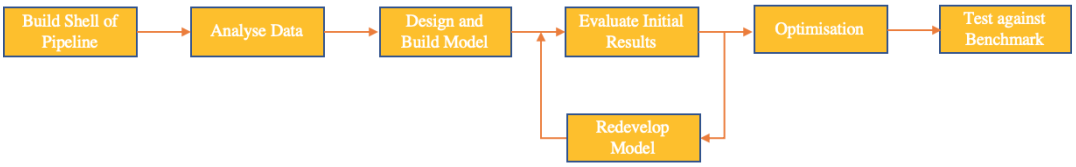



Figure 11: Overview of the processes involved in iterative designing and testing of models for the problem

## 4.2. Data Structuring

### 4.2.1 Data Description

The data is structured in lists of features. These are separated into 5 categories, two of which will be useful to this project. These will be *Fundamentals* and *Prices*. *Fundamentals* is data about the company itself, such as book value or sales whereas *Prices* is data regarding how the behavioural characteristics of the stock itself trades on the exchange. Each company is labelled with their unique ticker. Example: CNHI:IM which is the CNH Industrial NV.

*Fundamentals* is divided into two categories: Last Twelve Months (LTM) and Next Twelve Months (NTM). LTMs are the historical data points for the past year of a specific feature whereas NTMs are the predictions of those features for the next year. LTM contains 16 features and NTM contains 31 features. We are working with a subset of the Euro Stoxx 600 Index (SXXP) and so will only focus on companies that are categorised as Industrial Goods and Services. The Index ID for this is SXNP. There are 203 companies that fit this category. For each feature, there are 6313 data points, starting from 1995-01-03 (Y-D-M).

*Prices* is divided into two categories: Local and Euros which indicates the denomination that the values of the stocks are presented in. Because Local introduces an external factor of currency value fluctuation into the problem, it will be ignored and Euros will be used for its standardised format. Euros contains 14 features related to the technical indicators of the stock. Again, there are 203 companies for each of these features and the number of datapoints is 6313, starting from 1995-01-03 (Y-D-M).

The other 3 categories are *Mapping*, *Signals* and *Indices*. *Mapping* provides the keys of the dataset and so can be used to select a specific subset that is of interest. This is what is used to filter out the original number of companies (1184) to the Industrials subset (203). *Signals* provides 3 sets which looks at the investor sentiment. *Indices* provides the pricing data of worldwide indices.

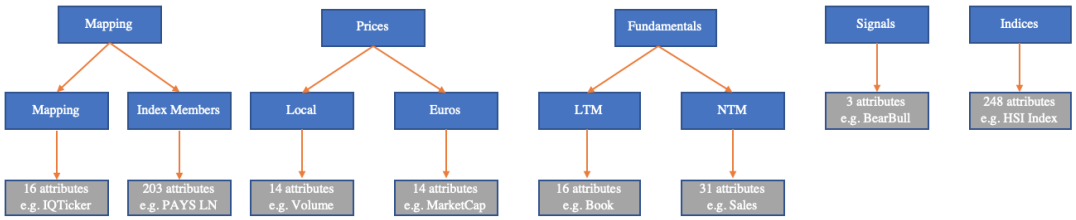



Figure 12: Diagram overview of the database hierarchy.

#### 4.2.2 Data Formatting

Initially, sample formats were distributed in csv formats. However, when the entire dataset was actually provided, the file was stored in a .rds file because the firm mainly operated in R. By studying the data storage structure as described in section 4.2.1, it was possible to extract the relevant information into individual csv files. These were then read into dataframes in the Jupyter notebook where the pipeline was built.

A few issues arose due to a the deprecated state of various aspects of the initial dataset. It was found that some datasets had not been filtered down to the 203 companies that formed part of the Industrial Goods and Services sector but still contained 1184 companies that formed the entire STOXX Europe 600 Index. Additionally, some attributes did not have a date key. It was therefore necessary to standardise all of the feature datasets with the same date index and companies. The company tickers were then anonymised simply for ease of use in terms of readability and iterating through tickers.

### 4.3. Data Preprocessing

#### 4.3.1 Data Transformation

A number of techniques were employed to manipulate the data into a suitable state for the machine learning algorithm. These were different for input and output variables. A min-max scaler, as previously defined in section 2.3.4, was used in order to sort each input feature into a suitable range between 0 and 1. The choice was mainly based on speed of implementation, with it already built into the *scikit-learn* library. Concerns were raised about whether a min-max scaler was appropriate for each of the attributes but it was decided that evaluation of the performance consequence of the scaler would be evaluated after a functional learning algorithm was implemented.

The output was first changed from the raw *price_close* nature to *returns*, which essentially looked at the multiplicative differences between a time interval and the previous time interval: $(\frac{x_t}{x_{t-1}})$. The outputs were then binarised, with +1 equating to a positive return and 0 indicating equal or negative returns. Although this was known to be a weak learner, the literature review suggested it was better to use this than focus on a regressive problem which may not produce any useful results. If more time was available, there was the suggestion that other binary learners could also be used, such as +1 equating to returns higher than the median across the companies and 0 being lower or equal to.

There was then a choice of selecting the time step. Although daily change was perceived to be highly susceptible to noise, the justification for its use came from the fact that it was deemed unviable to sacrifice a large proportion of the dataset. For example, assuming a complete time series, weekly intervals would reduce the number of datapoints from 6313 to approximately 1260. It was decided that, given enough time, a dilated RNN may be implemented to evaluate whether generalisation and noise reduction could be improved.

The use of time step multiplicative difference in the output raised questions about whether this would also be useful in the input variables, since most problems analysing a derivative evaluate the change in y given the change in x. Section 4.3.2 answers why this was not used. Additionally, the absolute values of a company's fundamental values have been used to predict the future stock price.

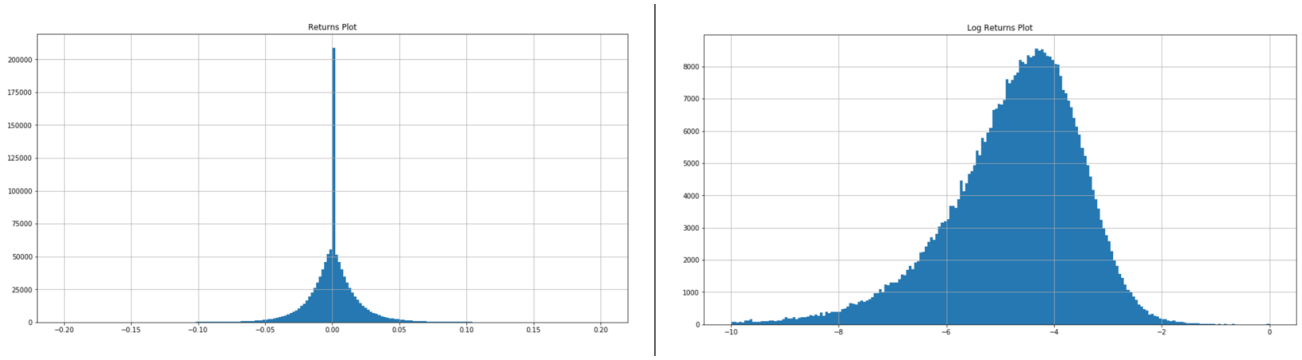



Figure 13: Histogram plots of returns (LHS) and log returns (RHS) distributions.

### 4.3.2 Sparsity

The major concern within the project stemmed from the quality and completeness of the dataset. Initially, the features and outputs were evaluated to see whether how much of the dataset was complete. Figure 14 shows that *Fundamentals* data alone was relatively poor and so it was decided to supplement the model with *Technical* inputs as well to improve performance.

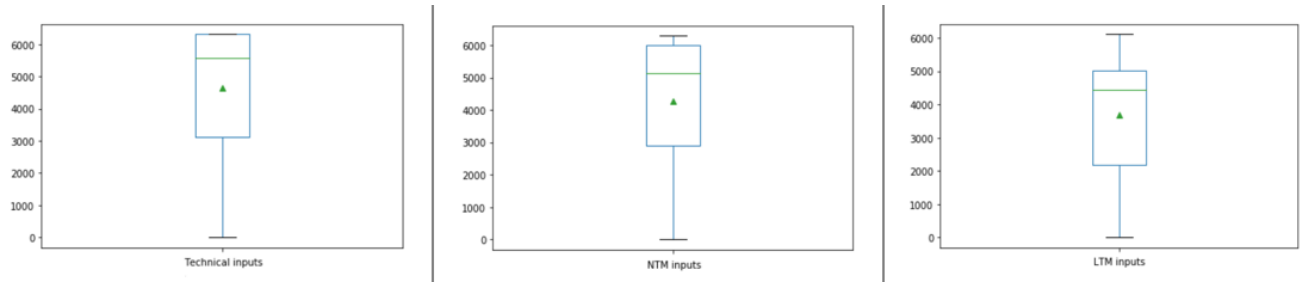



Figure 14: Box plot indicating the approximate average underlying count numbers for each type of input. Green triangles indicate the mean counts.

As mentioned in section 4.3.1, one idea revolved around the concept that in order to have some directional predictive power, the delta of the attributes will be measured over rather than the absolute values i.e. $\frac{x_t}{x_{t+1}}$. This means that the underlying assumption is that the price of stock changes given changes in the *Fundamentals* and *Price* features. Analysis was done on evaluating whether this was a viable transformation. It was discovered that because fundamental data is often quarterly reported, deltas that were not either NaN or 0 were very few in sample size and so the idea was abandoned.

Multiple solutions were considered in resolving how to handle missing values. These fell into three main categories: deletion, masking and generating values. As experienced with the initial architecture, deletion results in a non standardised sequential input which causes very poor performance by the LSTM as the dataset loses its Markovian property. In order to decide whether a masking or prediction approach should be used, the amount of data was also considered. Given the short duration of the project and limited resources, a huge number of input points was time consuming to train. Therefore, some further analysis of the data was carried out.

Most of the input data showed significant sparsity on dates before the turn of the century. It was therefore decided to reduce the datasets down further from beginning in 1995 to 2000. This decision was also supplemented by the idea that the temporal correlation in a company over a long period of time is likely to be much weaker and so removing very old data may even improve the learning performance of the model.

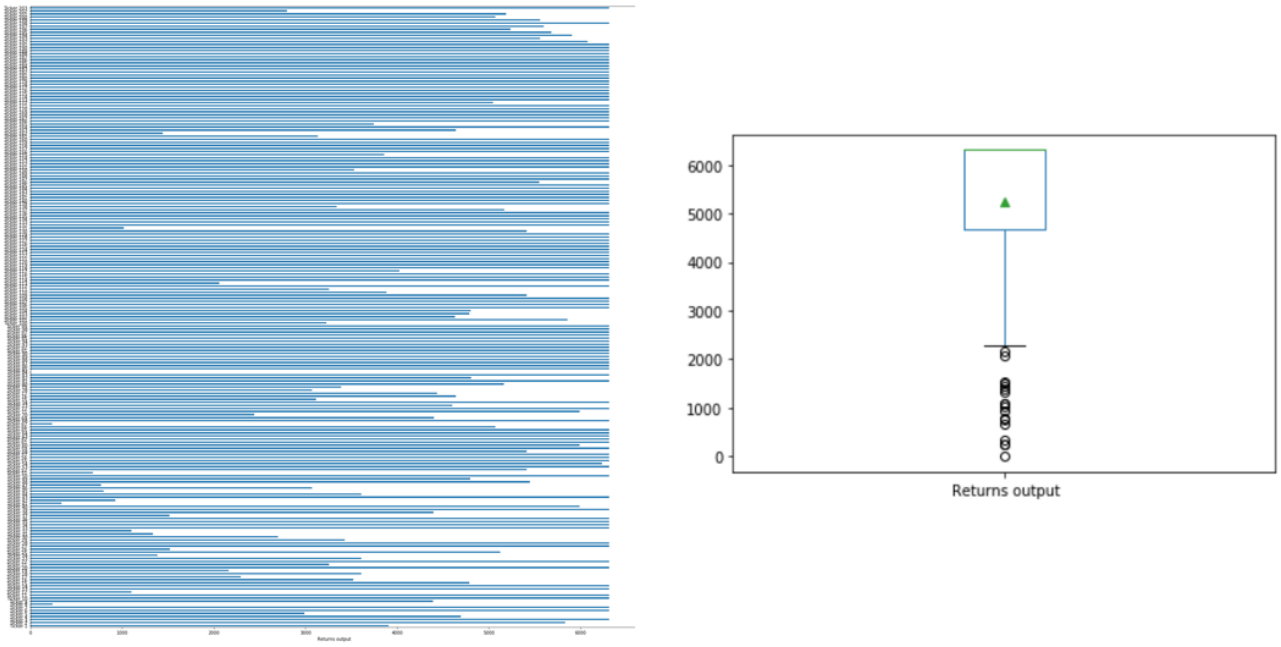



Figure 15: LHS: Individual ticker breakdown of datapoint counts. RHS: Boxplot of returns output count.

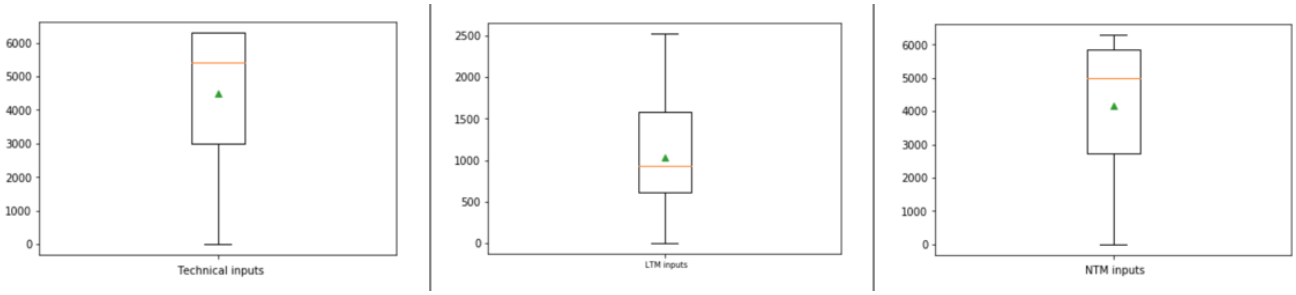



Figure 16: Box plot indicating the approximate average underlying delta counts for each type of input. Green triangles indicate the mean counts. As seen, LTM is particularly lower than the others.

| Total Number of Data Points | Data Points Used | Input Vectors with No Missing Values |
|---|---|---|
| 1,281,539 (100%) | 1,018,451 (79%) | 337,471 (26%) |

Table 1: Percentages of used and complete input vectors with respect to the entire dataset.

## 4.4. Architectural Design

### 4.4.1 Overview

The design of the architecture was implemented in an iterative process, starting initially with the most basic baseline design consisting of one hidden layer. The idea was to evaluate the results of the designs and then modify or redesign them to improve the performance and accuracy. In order to identify the sets of models that could be used, the initial process involved identifying the inputs, outputs and target function that were suitable for the fundamental problem.

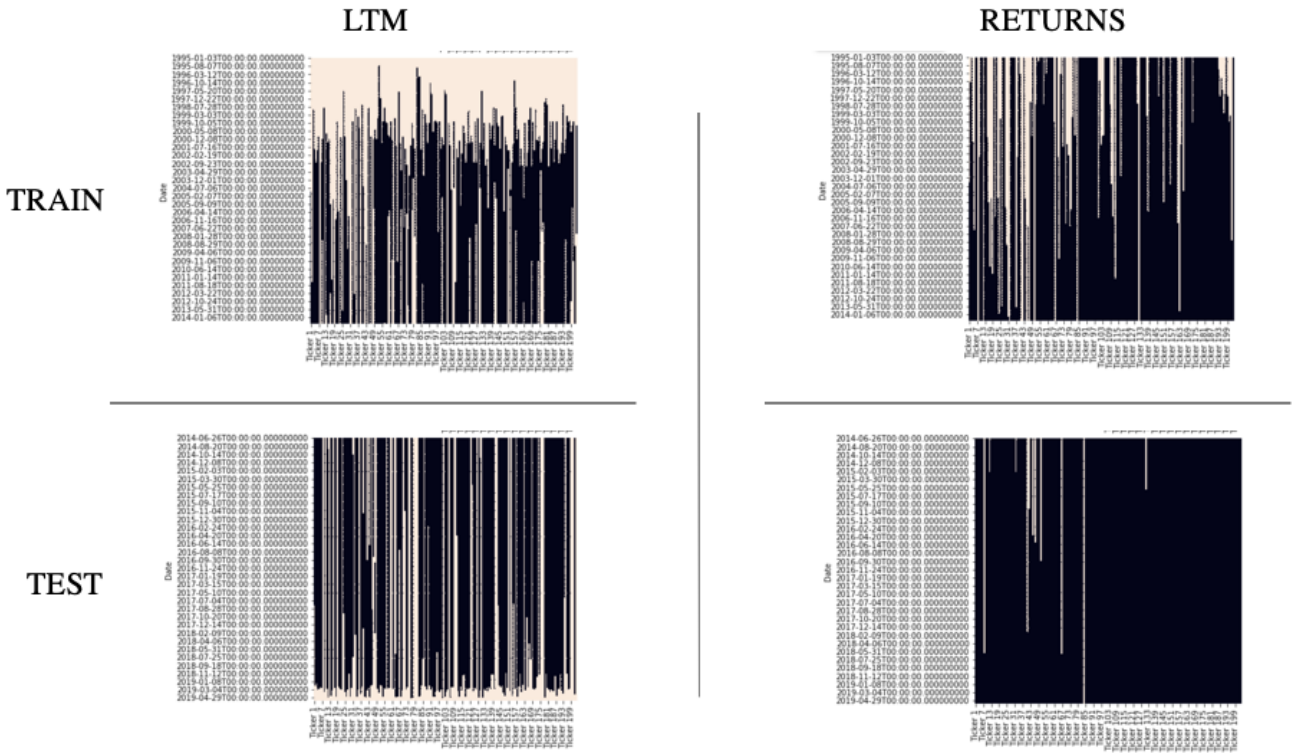



Figure 17: Heatmaps of nan values (depicted in white) of a few datasets.

### 4.4.2 Inputs and Outputs

Not all the features of the dataset were appropriate for the task and so some were discarded. Of the original 61 features, 21 were selected. There were several reasons for discarding some attributes: for example, a section of the NTM category was labelled as original (e.g. EpsOriginal and Eps) and this was decided to be too raw a format for usage. It would also have caused duplications in the input vector, unintentionally giving some attributes double the weighting. Additionally, for the same reason, a choice was made regarding using Earnings before Interest and Tax (EBIT) or Earnings before Interest, Tax, Depreciation and Amortisation (EBITDA). In industrial sectors, companies are often much more capital intensive and so the former is often employed since depreciation and amortisation unintentionally captures a portion of past capital expenditures.

Below is a list of descriptions for the 20 input attributes. These form a vector that is passed into the model:

- LTMs and NTMs:
  - Book Value: A company's net value as calculated by its total tangible assets on the balance sheet.
  - Dividends: A distribution of part of the company's earnings to its shareholders.
  - EBIT: This is evaluated by looking at the revenue of a company and subtracting the cost of goods sold and operating expenses.
  - Earnings per Share: This ratio assesses how profitable a company is by calculating the net income minus dividends over the outstanding number of shares.
  - Free Cash Flow: The amount of cash a company generates after accounting for capital expenditures.
  - Relative Price to Book Ratio: The ratio of market price per share over the book value per share relative to other companies within the same sector.
  - Sales: The total sum of prices paid by customers for products.

- Prices
  - Enterprise Value: Equivalent to the sum of the market capitalisation and total debt minus cash equivalents available to the company.
  - Market Capitalisation: The stock price multiplied by the number of outstanding shares.
  - Price High: The highest price a company's stock reaches within the trading day.
  - Price Low: The lowest price a company's stock reaches within the trading day.
  - Price Open: The initial price of a company's stock when the trading day begins.
  - Volume: The total amount of company stock traded within the trading day.

As specified in section 4.4.4, multiple designs for outputs were iterated through, the most focused on is highlighted below. Our model focused on predicting both a target output label and the next input vector : $x_t \rightarrow y_t, x_t \rightarrow x_{t+1}$. The target output label was defined as following:

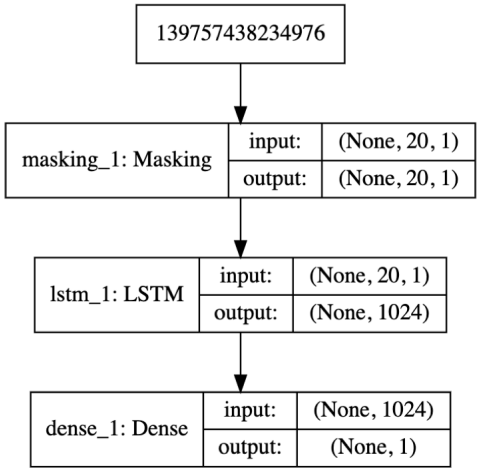
- Price Close: The end price of a company's stock at the finish of the trading day.

### 4.4.3 Loss Function

Although the idea of designing a specific loss function for the problem was considered, there were concerns that it may cause more issues. Therefore, it was decided to first attempt the model with a standard binary cross entropy loss function as described in section 2.2.3.

### 4.4.4 Design Iterations

The initial model used was the most fundamental LSTM possible, with one hidden layer. An additional masking layer was also added in case it was decided that NaN values would be masked instead of predicted or deleted. Afterwards, a stacked LSTM with multiple hidden layers and dropouts between to avoid overfitting was also tested to see if a deeper model could improve the performance of the model.




| Layer (type) | Output Shape | Param # |
|---|---|---|
| masking_1 (Masking) | (None, 20, 1) | 0 |
| lstm_1 (LSTM) | (None, 1024) | 4202496 |
| dense_1 (Dense) | (None, 1) | 1025 |

Total params: 4,203,521
Trainable params: 4,203,521
Non-trainable params: 0

Figure 18: LHS: Architectural overview of the initial one hidden layer model. RHS: Model summary with number of parameters.

One of the problematic issues was the model's capability of handling the missing values within the data. In order to reduce the dataset into a practically trainable size, it was chosen to delete missing values input vectors and output values. However,

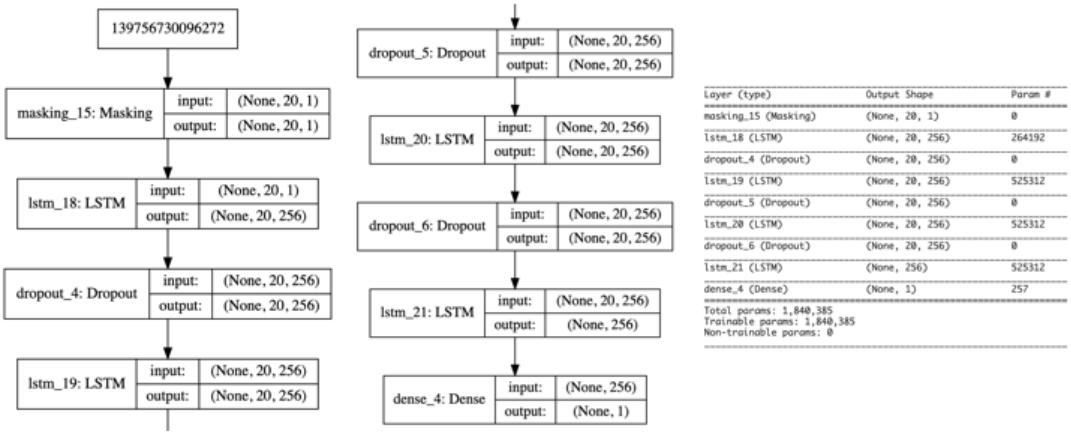



Figure 19: LHS: Architectural overview of the stacked LSTM model. RHS: Model summary with number of parameters.

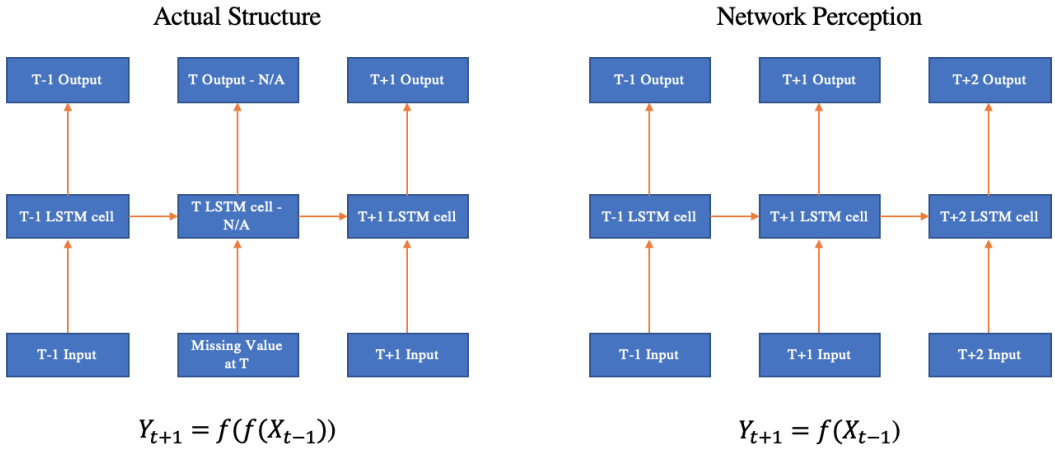
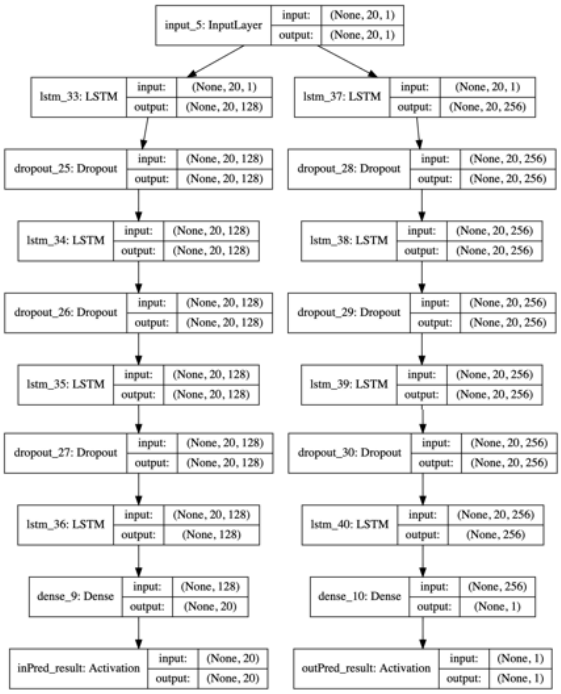



Figure 20: Demonstration of errors introduced into output function of LSTM by missing values.

this had the consequence of causing time skips in the model. This meant that the memory cell was heavily biasing much older information and not achieving accurate prediction or performance.

In order to resolve this, a new architecture was designed where the model predicted both the next input vector and an output value. This would then be able to fill in any missing values and rectify the issue experienced with the initial model. The model would first be trained on the longest continuous time series data available without any missing data so that both predictive models have had an opportunity to train and learn. After each iteration of a series, the model's hidden states will be reset when training begins on a different series that is either unrelated to the previous by time or company. The necessity for predicting the next input values stems from the concern that if this was not implemented, the series would not have enough data to provide actual learning. By using prediction, it would be possible to concatenate two series with minor amounts of missing values into a single, more substantial series. This would only function with relatively small gaps as the error variance increases with increasing numbers of continuous input predictions and this is likely to be detrimental to both the performance of the output prediction and the model's learning.




| Layer (type) | Output Shape | Param # | Connected to |
|---|---|---|---|
| input_5 (InputLayer) | (None, 20, 1) | 0 | |
| lstm_33 (LSTM) | (None, 20, 128) | 66560 | input_5[0][0] |
| lstm_37 (LSTM) | (None, 20, 256) | 264192 | input_5[0][0] |
| dropout_25 (Dropout) | (None, 20, 128) | 0 | lstm_33[0][0] |
| dropout_28 (Dropout) | (None, 20, 256) | 0 | lstm_37[0][0] |
| lstm_34 (LSTM) | (None, 20, 128) | 131584 | dropout_25[0][0] |
| lstm_38 (LSTM) | (None, 20, 256) | 525312 | dropout_28[0][0] |
| dropout_26 (Dropout) | (None, 20, 128) | 0 | lstm_34[0][0] |
| dropout_29 (Dropout) | (None, 20, 256) | 0 | lstm_38[0][0] |
| lstm_35 (LSTM) | (None, 20, 128) | 131584 | dropout_26[0][0] |
| lstm_39 (LSTM) | (None, 20, 256) | 525312 | dropout_29[0][0] |
| dropout_27 (Dropout) | (None, 20, 128) | 0 | lstm_35[0][0] |
| dropout_30 (Dropout) | (None, 20, 256) | 0 | lstm_39[0][0] |
| lstm_36 (LSTM) | (None, 128) | 131584 | dropout_27[0][0] |
| lstm_40 (LSTM) | (None, 256) | 525312 | dropout_30[0][0] |
| dense_9 (Dense) | (None, 20) | 2580 | lstm_36[0][0] |
| dense_10 (Dense) | (None, 1) | 257 | lstm_40[0][0] |
| inPred_result (Activation) | (None, 20) | 0 | dense_9[0][0] |
| outPred_result (Activation) | (None, 1) | 0 | dense_10[0][0] |

Total params: 2,304,277
Trainable params: 2,304,277
Non-trainable params: 0

Figure 21: Overview of the architecture of the multi output model.

## 4.5. Training and Testing

The fundamental objective of finding an optimal split for the training and test sets is to minimise the tradeoff between the parameter variance during training and the performance statistic variance during testing. Both of these variances increase with reduced data. However, it is important to remember that the predominant concern is with the absolute number of instances for each category rather than the percentage splits. Since there are a large number of samples available in this problem, there has been less focus on the optimising these percentage divisions.

Although multiple ideas of cross validation were explored in section 2.2.4, it was decided that there was enough data available to save for a hold out approach. Additionally, there were concerns in terms of time constraint about spending significant time analysing selecting an appropriate size for folds so that the independence assumption of neighbouring folds could be conserved. It was therefore decided to discard cross validation methodologies.

A rule of thumb employed in many models is to start by following the Pareto principle and divide the training and test splits into 80/20 respectively. The training data is then usually separated into training and validation in the same 80/20 ration again. Since there is a large amount of data available, it was decided to reduce the amount allocated to training for time saving purposes and so the final splits decided for training, validation and testing was 60/20/20.

## 4.6. Hyperparameter Optimisation

Hyperparameters were initialised based on some rules of thumb that were discovered during the literature review. It would then be improved upon through adjustment and tuning before being reevaluated to see if any performance boost was achieved. The intended method would be to perform a grid search using the toolkit available in the *scikit-learn* library. Unfortunately, the performance of the model was fundamentally poor and so its responsiveness to optimisation insignificant. This section analyses how the hyperparameters were initialised.

Ruder analyses the advantages and disadvantages of various gradient descent optimisation algorithms in his 2017 paper[28]. He highlights that if the input data is sparse, it is likely the best results would be achieved using an adaptive learning-rate

method. This performs better that learning rate schedules such as time or step decay because the parameters are updated relative to their presence or availability. Learning rate schedules offer the same learning update to all parameters regardless of whether it is missing or not. Additionally, it is observed that adaptive methods tend to have fast convergences and are less likely to get stuck in saddle points during gradient descent.

Some of the algorithms considered included Adadelta, RMSprop and Adaptive Moment Estimation (Adam). Adam[18] was ultimately selected as Ruder assesses it to be better than Adadelta and RMSProp due to its component that mimics momentum, additionally highlighting that its bias-correction helps outperform other methods towards to end of optimisations as gradients become sparser. Adam's weight updated is calculated as such: $W_{t+1} = W_t - \frac{\eta}{\sqrt{v_t+\epsilon}} \odot m_t$ where $m_t, v_t$ are the beta 1 and 2 decay parameters and $\epsilon$ is to prevent division by 0.

As mentioned by Brownlee[5], the activations functions for LSTMs are typically $tanh$ for the cell state and sigmoid for the node output. These are also the default parameter settings in Keras. ReLU applications to the node outputs are typically considered infeasible as they tend to result in exploding gradients due to the potential large values. Although there has been research into designing stable ReLU driven LSTM networks through appropriate weights initialisation, the project will focus on using the default activation functions.

```
Epoch 1/30
269975/269975 [==============================] - 3307s 12ms/step - loss: 8.0394 - acc: 0.5012
Epoch 2/30
269975/269975 [==============================] - 3290s 12ms/step - loss: 8.0394 - acc: 0.5012
Epoch 3/30
269975/269975 [==============================] - 3300s 12ms/step - loss: 8.0394 - acc: 0.5012
Epoch 4/30
269975/269975 [==============================] - 3280s 12ms/step - loss: 8.0394 - acc: 0.5012
Epoch 5/30
```

Figure 22: Screenshot indicating an issue with no decreasing loss function during training due to an incorrectly selected activation function.

The selection of the batch size mainly revolved around computational time efficiency as the high complexity of the model and size of dataset made iterations very taxing to run. It was therefore decided to initially begin with large batch sizes of 1024 as there are suggestions that powers of 2 can improve computational performance by aligning virtual processors onto the physical processors of the GPU, which are often a power of 2. Smaller batches are often considered better for test accuracy as they provide more frequent updates to the model parameters. However, the counterargument is that very small batches are more susceptible to noise and therefore not converge on the correct minima thus increasing convergence time.

Another hyperparameter to consider was the dropout values for deep networks to avoid overfitting. Gal et al.[9] published a paper regarding a variational structure of dropouts which included applying them on the feedforward connections to the next cell. During their experimentation, they derived that an optimal dropout probability for their medium network consisting of approximately 1300 cells was 0.2 and so this was the value used for this project. It is likely that an increase in both depths and numbers of cells is likely to result in a higher optimal dropout value.

It is hard to specify particularly the number of LSTM layers which would be optimal. The number depends on factors such as what the task itself is and the properties of the dataset. There is limited literature exploring this hyperparameter in a financial data analysis context but Sak et al.[30] evaluates that approximately 5 layers is optimal for acoustics modelling. In an ideal environment, there would be enough time to test increasing numbers of layers but due to time constraints, it was decided to settle with 4 layers as the deep model.

The number of hidden cells per layer was decided by following Hagan's rule of thumb[13]. He specifies that the upper bound for hidden neurons that won't result in overfitting is: $N_h = \frac{N_s}{\alpha \times (N_i + N_o)}$ where $N_h, N_i$ and $N_o$ are the hidden, input and output neurons respectively and $N_s$ and $\alpha$ are the number of samples and scaling factor which usually lies between 2-10. This helps limit the number of free parameters in the model with respect to the degrees of freedom in the input data. From this, it is possible to derive and approximate the number of hidden cells, given the sample size:

$$N_h = \frac{N_s}{\alpha \times (N_i + N_o)}$$
$$1190 = \frac{250,000}{10 \times (20 + 1)}$$

### 4.7. Benchmark Design

Our selected benchmark performance metric will be derived from the naive diversification choice heuristic. This means that for a given universe of $N$ stocks, a portfolio will be constructed by allocating a $\frac{1}{N}$ weighting to each stock. The interest here is in the efficiency of such a portfolio compared to those which are optimised to favour certain stocks achieved by overweighting them such that their allocations are greater than $\frac{1}{N}$. DeMiguel et al. conducted investigations[6] regarding 14 mean-variance optimisation models and found that there was no statistically significant improved performance relative to $\frac{1}{N}$ diversification. As such, it is of interest for investment managers to construct mathematical models that can outperform the simplistic model often utilised by less sophisticated investors for its ease-of-use.

In order to evaluate this, the output values will be ordered and a theoretical portfolio constructed with respect to the percentage of the total sum of the outputs ($weighting(i) = \frac{output_i}{\sum_{i=1}^{N} output_i}$). It should be noted that the architecture of the network will be devised such that a larger output is better e.g. implies a higher return, lower volatility etc. Although this portfolio layout is by no means optimised for best performance, it succinctly describes and roughly equates to what stocks the model has selected to over and underweight. It is then possible to evaluate this portfolio with respect to three basic performance measure and compare and contrast to those of the naive diversification heuristic.

- Absolute Return: This is the most basic method of evaluation - observing how much return is generated by a portfolio over a given period of time. The simplicity of the measure allows it to be easily implemented and direct comparisons can be made between the model generated and benchmark figures. It is calculated by $100 \times \frac{V_{end} - V_{start}}{V_{start}}$ where $V$ is the value of the portfolio and $start$ and $end$ are the beginning and finish times of the investment. A higher absolute return indicates better performance.

- Sharpe Ratio: First introduced by William Sharpe[31] in 1966, this is a commonly used method of measuring risk adjusted return. Since a more risky investment should in theory offer a greater potential reward, a more insightful method of evaluation is judging whether the returns received are in line with the risk taken in a given portfolio. A simplified version will be used that does not account for the risk free rate to reduce complexity in calculations: $SimpleSharpeRatio = \frac{R_p}{\sigma_p}$ where $R_p$ and $\sigma_p$ are the return and its standard deviation of the portfolio respectively. A higher Sharpe Ratio indicates better performance.

- Portfolio Turnover: This measures how often the stocks within the portfolio have to be bought and sold to rebalance and maintain the model's strategy. Although it does not theoretically affect the performance, the transaction costs of buying and selling in a real world scenario can significantly affect a portfolio's profitability. Therefore a lower turnover is preferred. It is calculated with the following equation: $turnover = \frac{min(s_b, s_s)}{NAV}$ where $NAV$ is the Net Asset Value and $s_b$ and $s_s$ are the cost of stocks bought and sold respectively.

As work progressed on the project, it was realised that there was a large inconsistency between translating between the model results and applying it to a real world scenario. The project therefore relied heavily on calibration using model evaluation metrics such as accuracy, precision and recall. These are calculated in the following methods:

- Accuracy: $\frac{TruePos + TrueNeg}{TruePos + TrueNeg + FalsePos + FalseNeg}$. This evaluates the number of correct predictions as a proportion of the total predictions made.

- Precision: $\frac{TruePos}{TruePos + FalsePos}$. This accounts for the number of correctly predicted positive labels with respect to the set of predicted positive labels.

- Recall: $\frac{TruePos}{TruePos + FalseNeg}$. This looks at the number of correctly predicted positive labels with respect to the entire set of actual positive labels.

It was then decided to simply plot the distribution of predicted higher return values and analyse whether there was a distribution change when compared with the total returns distribution, as previously depicted in figure 13.

# 5. Implementation

## 5.1. Data Structuring

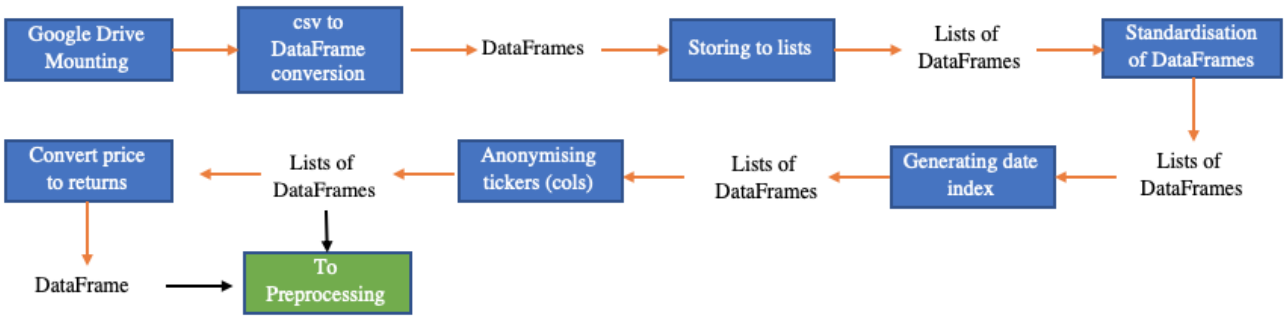



Figure 23: Program flow diagram of data structuring processes.

Given the large number of csv files, there was a necessity to store the resulting dataframes in a systematic method so that functions that needed to be performed on method could be done so efficiently. This was achieved by dividing the respective csv files first into input and target variables. Of course, only one file was needed for the target variable, being the close price dataset. The files themselves were stored on a Google Drive account, at the file path as shown in listing 1.

Listing 1: Importing csv file to dataframe.

```
price_close = pd.read_csv('/content/drive/My␣Drive/thw116_FYP/data/price_close.
    csv')
```

However, the sample data provided was not an accurate representation of the number of attributes actually available in the dataset and so a structure was required to store them. It was decided to store these dataframes in lists, based on the categories that they were originally allocated in the raw .rds file.

Listing 2: Formation of lists of dataframes according to type of input.

```
ltm_book = pd.read_csv('/content/drive/My␣Drive/thw116_FYP/data/ltm_book.csv')
ltm_div = pd.read_csv('/content/drive/My␣Drive/thw116_FYP/data/ltm_div.csv')
.......................
ltm_inputs = [ltm_book, ltm_div, ltm_ebit, ltm_ebitda, ltm_ebitda, ltm_eps,
    ltm_fcf, ltm_pbook, ltm_sales]
ntm_inputs = [ntm_book, ntm_div, ntm_ebit, ntm_ebitda, ntm_ebitda, ntm_eps,
    ntm_fcf, ntm_pbook, ntm_sales]
tech_inputs = [price_high, price_low, price_open, volume, enterprise_val]
```

This structuring of the dataframes made functions far easier to implement across all the data as it only required iteration through the lists. Part of the initial datasets were not standardised, with differing numbers of companies so it was necessary to standardise them before performing any data preprocessing. Additionally, tickers for companies were anonymised so they could be iterated through conveniently and a *datetime* index for each dataframe was created, as shown in listing 3.

Listing 3: Function to create datetime index for dataframes.

```
def time_index_generator(dataframe):
  dataframe['Date'] = pd.to_datetime(dataframe['Date'], dayfirst=True)
  dataframe.set_index('Date', inplace=True)

for i in range(0,len(ltm_inputs)):
```

```
    time_index_generator(ltm_inputs[i])
```

Finally, since it was desired to analyse the problem of predicting whether prices would go up or down, the target dataset of closing price was converted to a more appropriate format of returns by taking the difference between neighbouring rows.

### 5.2. Data Preprocessing

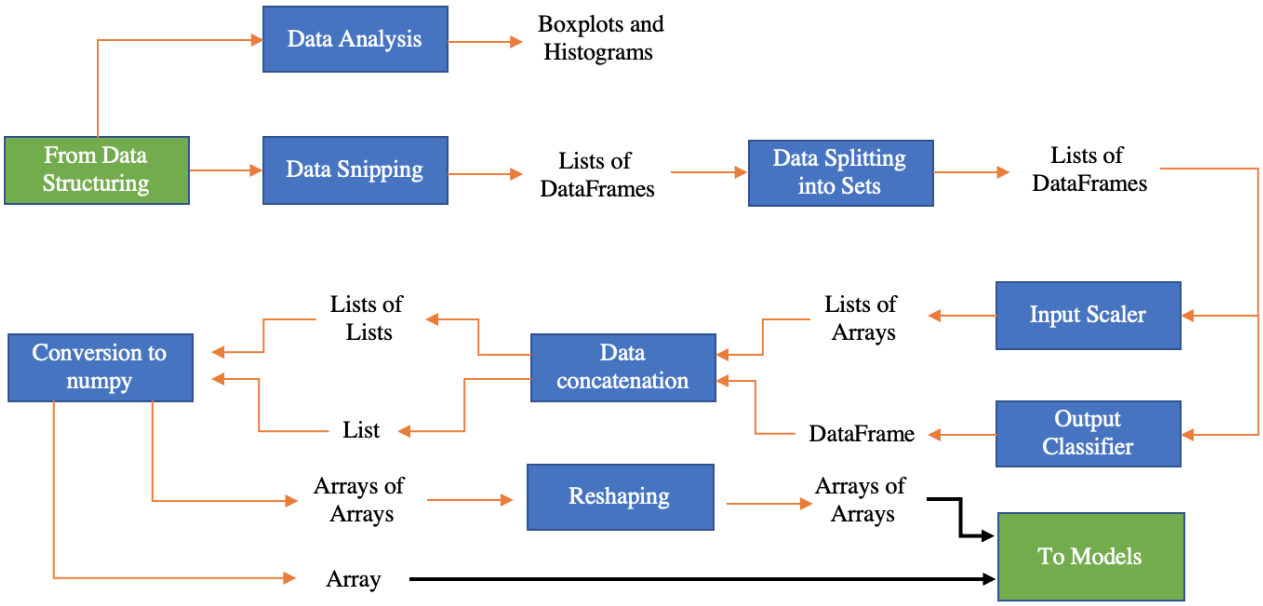



Figure 24: Program flow diagram of data preprocessing processes.

Before any processing was performed on the data, some of the inherent properties were analysed to establish what could or could not be done. This involved lots of data visualisation, with boxplots and histograms describing attributes of the data. Analysis of this is discussed in section 4.3. Basic *matplotlib* functionalities were used and a small function to analyse the useful datapoints available in the derivative of datasets with respect to time, as shown in 4.

Listing 4: Identifying the available datapoint counts in differences of $x_t$ and $x_{t-1}$.

```
def data_change(dataframe, column):
  noChange_count = 0
  NaN_count = 0
  for i in range (0, len(dataframe.index)):
    if dataframe[column].diff().iloc[i] == 0:
      noChange_count += 1
    if dataframe[column].isna().iloc[i]:
      NaN_count += 1

#   print('Total rows: ', len(dataframe[column]))
#   print('The number of rows where no change occurs: ', noChange_count)
#   print('The number of rows which are NaN: ', NaN_count)
#   print('Useful datapoints:', (len(dataframe[column])-noChange_count-
    NaN_count))

  return len(dataframe[column])-noChange_count-NaN_count
```

The data was the reduced in size and all old time series before 01/01/2000 was removed following the analysis in section 4.3.2. The next step involved dividing the output into the training, validation and test splits as specified in section 4.6. This was achieved by dividing the index of the dataframe accordingly and appending each newly created dataframe to a list, similar to before. The lists were then categorised by set and subcategorised by type of data. This meant that $3 \times 3 = 9$ lists were formed for the inputs. The target variable was simply divided into 3 dataframes.

Listing 5: Functionality to split original entire dataset into allocated training split size.

```
def training_set(train_percentage, dataframe):
  train_size = int(train_percentage*len(dataframe.index))
  train_set = dataframe[:train_size]
  return pd.DataFrame(train_set)

training_split = 0.6

ltm_trainInputs = []
ntm_trainInputs = []
tech_trainInputs = []

for i in range(0,len(ltm_inputs)):
  ltm_trainInputs.append(training_set(training_split, ltm_inputs[i]))

for i in range(0,len(ntm_inputs)):
  ntm_trainInputs.append(training_set(training_split, ntm_inputs[i]))

for i in range(0,len(tech_inputs)):
  tech_trainInputs.append(training_set(training_split, tech_inputs[i]))

returns_trainOutput = training_set(training_split, returns)
```

After dividing the data into sets, the input data was scaled into a range between 0 and 1 for easier processing by the model. It was important to do this after the splitting in order to avoid any data contamination and look ahead bias. The scaling function itself was taken from the *scikit-learn* library. One of the consequences of this was that it converted the data from a dataframe into *numpy* arrays.

Listing 6: Functionality to scale the input attributes between 0 and 1.

```
def input_scaling(dataframe):

  # This is the MinMax Scaling function
  sc = MinMaxScaler(feature_range = (0, 1))
  scaled_input_dataframe = sc.fit_transform(dataframe) # This is now an n-
      dimensional array type

#   transformer = Normalizer().fit(dataframe)
#   Normalizer(copy=True, norm='l2')
#   transformer.transform(X)

  return scaled_input_dataframe

np.warnings.filterwarnings('ignore', r'All-NaN␣(slice|axis)␣encountered')

for i in range(0,len(ltm_inputs)):
  ltm_trainInputs[i] = input_scaling(ltm_trainInputs[i])
```

```
    ltm_valInputs[i] = input_scaling(ltm_valInputs[i])
    ltm_testInputs[i] = input_scaling(ltm_testInputs[i])

  for i in range(0,len(ntm_inputs)):
    ntm_trainInputs[i] = input_scaling(ntm_trainInputs[i])
    ntm_valInputs[i] = input_scaling(ntm_valInputs[i])
    ntm_testInputs[i] = input_scaling(ntm_testInputs[i])

  for i in range(0,len(tech_inputs)):
    tech_trainInputs[i] = input_scaling(tech_trainInputs[i])
    tech_valInputs[i] = input_scaling(tech_valInputs[i])
    tech_testInputs[i] = input_scaling(tech_testInputs[i])
```

The output classification function translated the raw returns into the binary classifier as specified in 4.3.1. There was some difficulty in implementing a classifier using the median returns of a row as this required fairly taxing individual cell comparisons, which was computationally very slow. Due to time constraints, it was decided to use the simpler classifier where negative or 0 return equated to 0 and positive return equated to +1. This avoided the need for individual cell comparisons as there was no longer a need to change the comparison value for each row. The use of an unmodified returns dataset was also needed for analysing the model results.

Listing 7: Function to convert the raw output target to a binary label.

```
def output_classifier(type ,dataframe):

  # No adjustments
  if type == 'raw':
    scaled_dataframe = dataframe

  # Binary classifier where return>0 is +1, and return<0 is 0 labels
  if type == 'binary':
    pos_returns = dataframe.values > 0
    neg_returns = dataframe.values <= 0
    scaled_dataframe = pd.DataFrame(np.select([pos_returns,neg_returns], [1,0],
        default='NaN'), index=dataframe.index, columns=dataframe.columns)

  return scaled_dataframe

returns_trainOutput = output_classifier('binary', returns_trainOutput)
returns_valOutput = output_classifier('binary', returns_valOutput)

raw_testOutput = output_classifier('raw', returns_testOutput)
returns_testOutput = output_classifier('binary', returns_testOutput)
```

After preprocessing across the dataset, it was necessary to reformat the data into an input form acceptable to the Keras model architecture. Therefore, the input vectors had to be created by selecting each of the relevant attributes for a datapoint for a company at a specific time. This was achieved by iterating through the lists of dataframes and appending these attributes into a *input_unit* list which represented a vector input, before appending all of these vectors into another list for the relevant training, validation or test sets.

Listing 8: Formation of the input vectors for the model.

```
trainInput = []
trainTarget = []
```

```
valInput = []
valTarget = []

testInput = []
testTarget = []
rawtestTarget = []

# Check that the indices are of the same length
if len(ltm_trainInputs[0]) != len(returns_trainOutput):
  print ('training length', len(ltm_trainInputs[0]))
  print ('target length', len(returns_trainOutput))
  assert False, "Incompatible dataframe index lengths!"

# Training Set
for company in range(0, len(ltm_trainInputs[0][:1][0])):

  for time_unit in range(0, len(ltm_trainInputs[0])):
    input_unit = []

    for ltm_attribute in range(0, len(ltm_trainInputs)):
      input_unit.append(ltm_trainInputs[ltm_attribute][time_unit:time_unit
          +1][0][company])

    for ntm_attribute in range(0, len(ntm_trainInputs)):
      input_unit.append(ntm_trainInputs[ntm_attribute][time_unit:time_unit
          +1][0][company])

    for tech_attribute in range(0, len(tech_trainInputs)):
      input_unit.append(tech_trainInputs[tech_attribute][time_unit:time_unit
          +1][0][company])

    trainInput.append(input_unit)

for company in range(0, len(returns_trainOutput.columns)):
  for time_unit in range(0, len(returns_trainOutput.index)):
    trainTarget.append(returns_trainOutput[anon_tickers[company]][time_unit])
```

These lists were then converted back into *numpy* arrays. This provided the use of *reshape* functionality only accessible to arrays as well as the fact that they are more memory and computationally efficient. Keras requires a Since this is the immediate step before inputting into the architecture, there is no more need for the flexibility offered by lists.

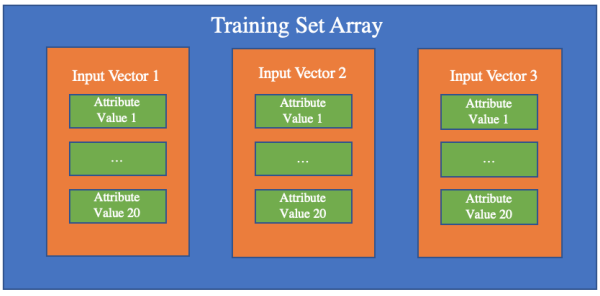



Figure 25: Diagram of reshaped input data structure for Keras.

Listing 9: Reshaping the input *numpy* arrays to the correct format.

```
# Conversion to numpy array for improved memory, performance and functionality
trainInput, trainTarget = np.array(trainInput), np.array(trainTarget)
valInput, valTarget = np.array(valInput), np.array(valTarget)
testInput, testTarget = np.array(testInput), np.array(testTarget)

trainInput = np.reshape(trainInput, (trainInput.shape[0], trainInput.shape[1],
    1))
valInput = np.reshape(valInput, (valInput.shape[0], valInput.shape[1], 1))
testInput = np.reshape(testInput, (testInput.shape[0], testInput.shape[1], 1))
```

## 5.3. Architecture

### 5.3.1 Single Output Model

For the single output model, the format for building the model was defined as a function that took in the parameters, train_X, train_Y, data_dim, which are the training data, test data, and dimensionality of the input training vector ($20 \times 1$) respectively. Various Keras functionalities are then called to define the model. Initially, Sequential() is called to establish an empty model using the Sequential API. A *Sequential* model is a linear stack of layers which can be created using the .add() method.

The first layer of the model must take the input_shape as a parameter. This is required for the model to provide it information on the format of the input vectors. The argument input_shape defines a 2 dimensional shape tuple. The first parameter of input_shape defines the width of the vector (i.e. the number of attributes/columns) and the second defines the time step size (also known as window size). When displaying the model, as shown in figure 19, the *None* parameter of the input shape indicates the batch size and means that any positive integer can be expected for that dimension, since batch dimension is not initialised at this point.

The two main layer types used in the architecture are Dropout and LSTM. Dropout is classed as a *Core* layer in Keras whereas LSTM falls under *Recurrent*. Dropout layers simply apply dropout to the inputs of their layers i.e. the outputs of previous layers. This is achieved by randomly a fraction of input units, specified by the parameter 'rate', to 0 during the update when training.

LSTM layers also require a few parameters to configure. As the name suggests, they are responsible for implementing the LSTM cell architectures. The parameter 'unit' specifies the number of cells in the layer. In stacking architectures, it is important to set the return_sequences parameter to *True* for all but the last layer. This means that all of the hidden state outputs are returned and passed to the next layer as opposed to just the hidden state of the final cell.

The final Dense layer is simply a fully connected layer than transforms the output of the previous cell into the intended output shape for the model. In this case, since only a single value is predicted, the 'units' parameter specifies the output dimensionality i.e. the number of nodes in the dense layer.

After the model is defined, the learning process must be configured by using the .compile() method. There are two arguments that must be specified which are the choice of optimiser and loss function. Optionally, a list of metrics may also be parsed through as an argument to provide evaluation of the model. Finally, the model is trained using the .fit() method. This takes as arguments *numpy* arrays representing the input and target data, as well as the number of epochs to train for and size of batch size. This generates a *History* object, which stores a record of the training losses and metrics after each epoch.

Listing 10: Model creation, compilation and fitting functionality of the single output model. The initial single hidden layer model is commented out.

```
data_dim = trainInput.shape[1]
timesteps = trainInput.shape[0]

# Sample Code
# model parameters:
```

```
def create_model(train_X, train_Y, data_dim):
  lstm_units = 1024

#   print('Build baseline binary model...')
#   model = Sequential()
#   model.add(Masking(mask_value=0., input_shape=(data_dim, 1)))
#   model.add(LSTM(lstm_units))
#   model.add(Dense(1, activation='sigmoid'))
#   model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['
    accuracy'])

  print('Build stacked binary model')
  lstm_units = int(lstm_units/4)

  model = Sequential()
  model.add(Masking(mask_value=0., input_shape=(data_dim, 1)))
  model.add(LSTM(lstm_units, return_sequences=True))
  model.add(Dropout(rate=0.2))
  model.add(LSTM(lstm_units, return_sequences=True))
  model.add(Dropout(rate=0.2))
  model.add(LSTM(lstm_units, return_sequences=True))
  model.add(Dropout(rate=0.2))
  model.add(LSTM(lstm_units))
  model.add(Dense(1, activation='sigmoid'))
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['
      accuracy'])

  return(model)

baseline_model = create_model(trainInput, trainTarget, data_dim)
print(baseline_model.summary())
SVG(model_to_dot(baseline_model, show_shapes=True).create(prog='dot', format='
    svg'))

history = baseline_model.fit(trainInput, trainTarget, epochs = 30, batch_size =
     1024, verbose = 1)
```

### 5.3.2 Multi output Model

By redeveloping the model into a multi output structure, the *Sequential* API was no longer usable. Hence, the model was initiated by calling the *Functional* API instead. Rather than structuring the object build inside a function as with the single output method, a class was defined to instantiate the model. This then allowed class methods to be developed to neatly define each individual branch of the two predictors.

Layers are connected together by pairing rather than using the .add() method in the single output model. This allows for more flexibility in terms of connecting layers in a non linear fashion.

Listing 11: Structuring of the multi output model, with two branches focused on the separate tasks of predicting the target variable, $y$ and the next input vector, $x_{t+1}$.

```
class fypNet:
  @staticmethod
```

```
    def build_input_predictor(inputs, data_dim, lstm_units):
      lstm_units = int(lstm_units/8)

      inPred = Masking(mask_value=-1., input_shape = (data_dim, 1))(inputs)

      inPred = LSTM(lstm_units, return_sequences=True)(inputs)
      inPred = Dropout(rate=0.2)(inPred)
      inPred = LSTM(lstm_units, return_sequences=True)(inPred)
      inPred = Dropout(rate=0.2)(inPred)
      inPred = LSTM(lstm_units, return_sequences=True)(inPred)
      inPred = Dropout(rate=0.2)(inPred)
      inPred = LSTM(lstm_units)(inPred)
      inPred = Dense(20)(inPred)

      result = Activation('softmax', name= 'inPred_result')(inPred)

      return result

    @staticmethod
    def build_output_predictor(inputs, data_dim, lstm_units):
      lstm_units = int(lstm_units/4)

      outPred = LSTM(lstm_units, return_sequences=True)(inputs)
      outPred = Dropout(rate=0.2)(outPred)
      outPred = LSTM(lstm_units, return_sequences=True)(outPred)
      outPred = Dropout(rate=0.2)(outPred)
      outPred = LSTM(lstm_units, return_sequences=True)(outPred)
      outPred = Dropout(rate=0.2)(outPred)
      outPred = LSTM(lstm_units)(outPred)
      outPred = Dense(1)(outPred)

      result = Activation('sigmoid', name= 'outPred_result')(outPred)

      return result

    @staticmethod
    def build(data_dim, lstm_units):

      input_shape = (data_dim, 1)
      inputs = Input(shape = input_shape)

      inputBranch = fypNet.build_input_predictor(inputs, data_dim, lstm_units)
      outputBranch = fypNet.build_output_predictor(inputs, data_dim, lstm_units)

      model = Model(inputs= inputs, outputs= [inputBranch, outputBranch])

      return model
```

Listing 12: Initialisation, compilation and training implementation for the multi output model

```
lstm_units = 1024
num_epochs = 10
initial_lr = 1e-3
```

```
batch_size = 32

# initialize our fypNet multi-output network
model = fypNet.build(data_dim, lstm_units)

losses = {'inPred_result': 'mean_absolute_error','outPred_result': '
    binary_crossentropy'}

# initialize the optimizer and compile the model

print('Compiling_model...')
# opt = Adam(lr=initial_lr, decay=initial_lr/epochs)
model.compile(optimizer='adam', loss=losses, metrics=['acc'])

model.fit(trainInput,{"inPred_result": trainInTarget, "outPred_result":
    trainTarget}, validation_data=(valInput, {"inPred_result": trainValTarget, "
    outPred_result": valTarget}), batch_size = batch_size, Stateful=False,
    epochs=num_epochs, verbose=1)
```

### 5.4. Evaluation Metrics

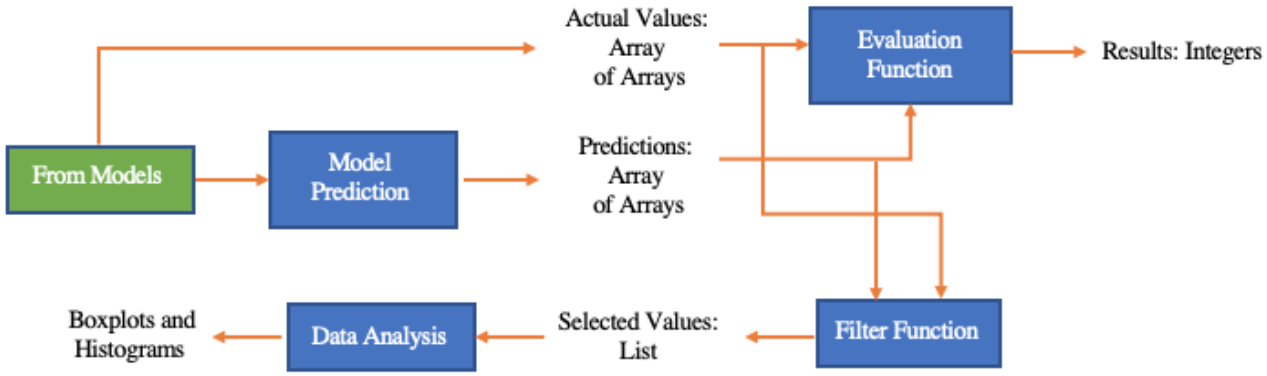



Figure 26: Program flow diagram of evaluation processes.

The final section of the code relates to actually evaluating the predictions of the result. For this, it was first required to generate the predictions from the unseen test data input. Since labels were binary, the Keras function of *predict_classes* was more suitable as this would generate the appropriate labels. Given more time, the *predict* values would be analysed to evaluate whether the continuous probabilities showed signs of an underlying relationship that higher returns correlated with higher probabilities.

Listing 13: Keras predict functions to generate predicted labels from test data.

```
stack_pred = trained_stack.predict(testInput, verbose=1)
print(stack_pred)

stack_classPred = trained_stack.predict_classes(testInput, verbose=1)
print(stack_classPred)
```

As discussed in section 4.7, 3 metrics were used to analyse the performance. This was achieved by implementing a function that compared the predictions to the actual labels and then categorising the results as shown in figure 27. The

number of occurrences for each was then counted and the metrics calculated by the appropriate formulas accordingly, also provided in section 4.7.

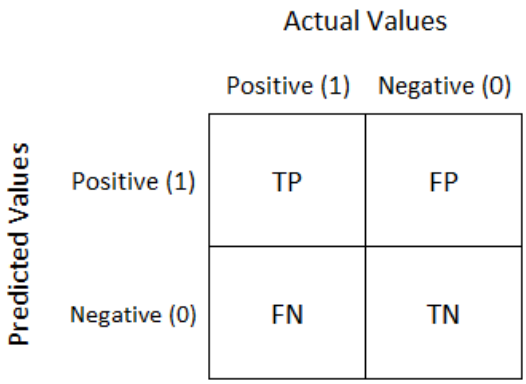



Figure 27: Confusion matrix for labelling prediction results.

Listing 14: Implemented function which evaluates the model prediction performance based on accuracy precision and recall.

```
def evaluation_metrics(prediction, actual):
  TP = 0
  TN = 0
  FP = 0
  FN = 0
  for i in range (0, len(actual)):
    if actual[i][0] == '1':
      if prediction[i][0] == 1:
        TP += 1
      else:
        FN += 1
    if actual[i][0] == '0':
      if prediction[i][0] == 0:
        TN += 1
      else:
        FP += 1

  print(TP, TN, FP, FN)
  return (TP+TN)/(TP+TN+FP+FN), TP/(TP+FP), TP/(TP+FN)

accuracy, precision, recall = evaluation_metrics(stack_classPred, testTarget)

print(accuracy, precision, recall)
```

Finally, the labels that the model predicted to be 1 were selected and filtered into another list. These values were then equated to their original raw returns value to analyse the datapoints that the model had selected. Further evaluation and analysis can be found in section 7.

Listing 15: Filtering out positively labelled predictions to extract related actual return.

```
selected = []

for i in range(len(rawtestTarget)):
```

```
    if stack_classPred[i][0] == 1:
      selected.append(rawtestTarget[i])
```

# 6. Testing

## 6.1. Test Plan

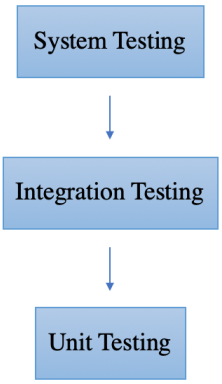



Figure 28: Levels of software testing from high (Top) to low (Bottom)

The procedure of evaluating the achievements of the project will vary according to the stage of the software development cycle. On the atomic level is the unit test, which will be used to validate that each of the separate modules and functions of the software are functioning as expected. An example[32] of how unit testing will be implemented is shown below. By using the assert statement in Python, it is possible to determine whether the function, sum(), is outputting the expected result of 6. An incorrect result will raise an AssertionError and output the message $not6$.

Listing 16: Example Test.

```
def test_sum():
    assert sum([1, 2, 3]) == 6, "not6"
```

The next level up involves integration testing, which focuses on the correctness of functionality in combined components of the software. An example of this would be examining the outputs of a function which takes as an input a processed dataframe from another function. It is important to test the combined behaviour of these two components after individual unit testing to ensure that it is in line with expectations. Given the data orientated nature of the project, this will mainly be achieved through the development of small, well-defined test datasets to compare and contrast with the outputs of the components. This is known as example testing and is contrasted with batch testing, which focuses on passing through significant amounts of data to evaluate correctness. The latter approach is less appropriate in the context of this project since it is very time consuming both to run tests and generate vast amounts of data.

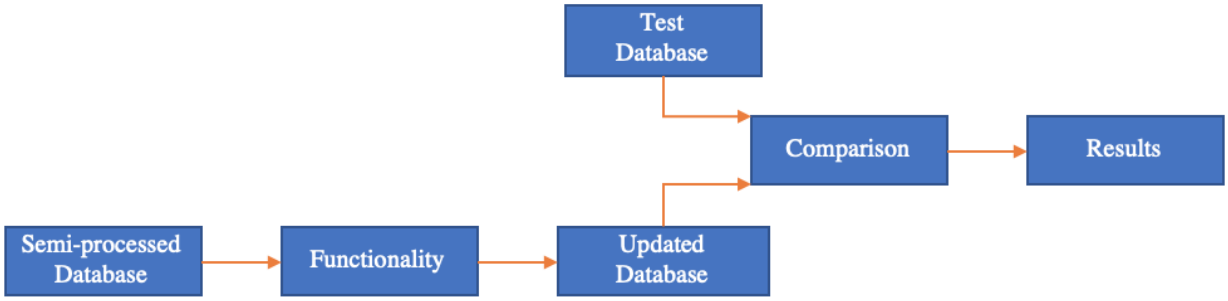



Figure 29: Flow diagram of how an integrated test may be executed.

System testing will involve evaluating the realised performance of the whole system. This will then allow for the model to be comparable to other methodologies developed in the field and industry. This will be covered in section 8.2.

### 6.2. Unit Test Implementation

The implementation for some unit tests are depicted below, with one for a DataFrame assertion and another for a *numpy* assertion. Sample tests were first initialised in lists before being converted into the appropriate format for the function that was being tested. Both correct and incorrect results for the tests were then created in lists and converted appropriately before a function was implemented to carry out the test. Listing 18 demonstrates an example of an AssertionError which also returns an error message identifying the problem. These unit tests are not exhaustive in terms of establishing edge cases but provide reasonable effectiveness at building correctly functioning code for the purpose of the project.

Listing 17: Example of a DataFrame comparison test.

```
# Test Set Generator Test

initialframe = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
outputframe1 = [7, 8, 9, 10] # success case
outputframe2 = [4, 5, 6, 7] # fail case
outputframe3 = [6, 7, 8, 9, 10] # fail case

testdf1 = pd.DataFrame(initialframe)
testdf2 = pd.DataFrame(outputframe1)
testdf2.index += 6
testdf3 = pd.DataFrame(outputframe2)
testdf3.index += 6
testdf4 = pd.DataFrame(outputframe3)
testdf4.index += 6

def test_SET_GENERATOR(test, benchmark):
  results = test_set(0.5, 0.1, test)
  pd.testing.assert_frame_equal(results, benchmark)
  print('test successful')

test_SET_GENERATOR(testdf1, testdf2)
test_SET_GENERATOR(testdf1, testdf3)
test_SET_GENERATOR(testdf1, testdf4)
```

Listing 18: Assertion Error raised by the second fail case of the Test Set Generator Test.

```
AssertionError: DataFrame.iloc[:, 0] are different

DataFrame.iloc[:, 0] values are different (100.0 %)
[left]:  [7, 8, 9, 10]
[right]: [4, 5, 6, 7]
```

Listing 19: Example of a *numpy* comparison test.

```
# Input Scaling Test
# for minmax case

inputvals = [1, 2, 3, 4, 5]

minmax1 = np.array([0, 0.25, 0.5, 0.75, 1]) # success case
minmax1 = np.reshape(minmax1,(5, 1))
minmax2 = np.array([1, 0.25, 0.75, 0.75, 0]) # fail case
minmax2 = np.reshape(minmax2,(5, 1))
```

```
testdf1 = pd.DataFrame(inputvals)

def test_INPUT_SCALER(test, benchmark):
  results = input_scaling(test)
  np.testing.assert_array_equal(results, benchmark)
  print('test successful')

test_INPUT_SCALER(testdf1, minmax1)
test_INPUT_SCALER(testdf1, minmax2)
```

### 6.3. Integration Test Implementation

One integration test was implemented but it was decided that they were fairly convoluted to implement and redundant. The structure of the codebase is very linear, to the point where all the integration tests are simply a direct combination of unit tests. AssertionErrors in integration tests are also much less effective, as shown in listing 20. The fail case in this example has multiple errors, such as mislabelled column names and incorrectly computed values in the dataframe. However, the AssertionError simply indicates the first one encountered, which was the dataframe columns mislabelling.

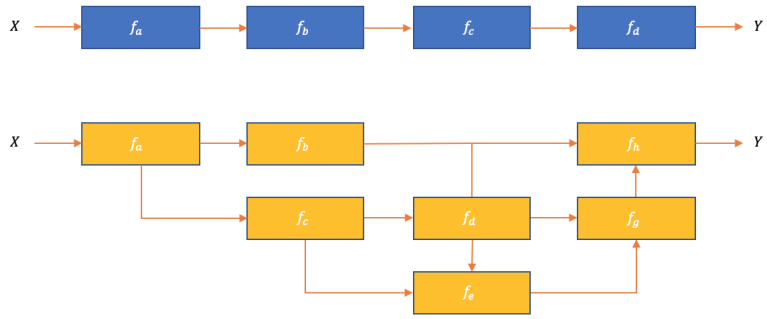



Figure 30: Integration testing may be more useful in the contrived bottom program flow example where functionality is not very linear. This project's program flow is more similar to that of the top program flow, depicted in blue.

Listing 20: Example of an integration test for 3 linked functionalities.

```
# Data Wrangling Test

start = dt.datetime.strptime("26-12-1999", "%d-%m-%Y")
end = dt.datetime.strptime("5-01-2000", "%d-%m-%Y")
dates = [start + dt.timedelta(days=x) for x in range(0, (end-start).days)]
values1 = [10000, 10000, 9000, 9000, 9900, 9900, 9999, 9999, 19998, 19998]
values2 = [float('nan'),float('nan'),float('nan'),float('nan'),float('nan'),
    float('nan'),float('nan'),float('nan'),float('nan'),float('nan')]

testdf1 = pd.DataFrame(dates)
testdf1.rename( columns={0:'Date'}, inplace=True )
testdf1['CompanyA'] = values1
testdf1['CompanyB'] = values2

# success case
test1values1 = [float('nan'), 0.0, -0.1, 0.0, 0.1, 0.0, 0.01, 0.0, 1.0, 0.0]
```

```
test1values2 = [float('nan'),float('nan'),float('nan'),float('nan'),float('nan'
    ),float('nan'),float('nan'),float('nan'),float('nan'),float('nan')]

benchmarkdf1 = pd.DataFrame(dates)
benchmarkdf1.rename(columns={0:'Date'}, inplace=True)
benchmarkdf1 = benchmarkdf1.set_index('Date')
benchmarkdf1['Ticker 1'] = test1values1
benchmarkdf1['Ticker 2'] = test1values2

# fail case
test2values1 = [10000, 10000, 9000, 9000, 9900, 9900, 9999, 9999, 19998, 19998]
test2values2 = [float('nan'),float('nan'),float('nan'),float('nan'),float('nan'
    ),float('nan'),float('nan'),float('nan'),float('nan'),float('nan')]

benchmarkdf2 = pd.DataFrame(dates)
benchmarkdf2.rename(columns={0:'Date'}, inplace=True)
benchmarkdf2 = benchmarkdf2.set_index('Date')
benchmarkdf2['Ticker 1'] = test2values1
benchmarkdf2['Ticker 3'] = test2values2

def test_WRANGLE_INTEGRATION(test, benchmark):
  # First Test
  time_index_generator(test)

  # Second Test
  anon_tickers = ticker_list_generator(len(test.columns))
  test.columns = anon_tickers

  # Third Test
  results = price_to_returns('daily', test)

  pd.testing.assert_frame_equal(results, benchmark)
  print('test successful')

test_WRANGLE_INTEGRATION(testdf1, benchmarkdf1)

# reset testdf1
testdf1 = pd.DataFrame(dates)
testdf1.rename( columns={0:'Date'}, inplace=True )
testdf1['CompanyA'] = values1
testdf1['CompanyB'] = values2

test_WRANGLE_INTEGRATION(testdf1, benchmarkdf2)
```

## 7. Results

The model was trained through iterations of epochs, with each epoch using the entire training and validation sets. The results for loss and accuracy were calculated after each epoch and plotted, as shown in figure 31 and 32. A detailed breakdown can be found in the GitHub repository link located in section 10.1.


Figure 31: LHS: Plot of multi output model's losses for next input predictor. RHS: Plot of multi output model's losses for target variable predictor.


Figure 32: LHS: Plot of multi output model's accuracy for next input predictor. RHS: Plot of multi output model's accuracy for target variable predictor.

A few interesting observations can be made in both of the models. Surprisingly, for the next input prediction model, the validation loss is actually lower than the training loss. This could possibly be explained by how Keras calculates the training and validation losses. Training loss is recorded after each epoch and is calculated as a moving average with respect to the number of epochs that have been run. Each individual training loss for an epoch is calculated by averaging the losses of each batch across the epoch. However it should be noted that as the model trains on a batch, it learns and its weights change. This means that the model running on the last batch of an epoch is substantially better than the model training on the initial batch of an epoch. For the multi output model, there are approximately 160 batches in each epoch. When the model then trains on the validation set, it is the best performing model derived from the last batch of the training set and is fixed i.e. it does not learn. This means that it does not suffer from the penalty performance of dropouts which are active during training. Similar to training, validation loss for an epoch is calculated as the average of the batch losses across the epoch. This observation can be mathematically explained as following:

$$\text{Training loss of an epoch} = \frac{1}{X}\sum_{i}^{X} e(f_i(x_i))$$

where $X$ is the number of batches in the training set and $e(f_i(x_i))$ is the loss of model $f_i$ on batch $x_i$

$$\text{Validation loss of an epoch} = \frac{1}{Y}\sum_{j}^{Y} e(f_X(x_j)).$$

where $Y$ is the number of batches in the training set and $e(f_X(x_i))$ is the loss of the best and last batch training model $f_X$ on batch $x_j$.

For the output predictor, the same observation of a higher training than validation loss can be observed up until around epoch 14. At this point, the new lines intersect and validation loss thereafter increases in a typical fashion of an overfitting model. When the lines intersect, $\frac{1}{X}\sum_{i}^{X} e(f_i(x_i) = \frac{1}{Y}\sum_{j}^{Y} e(f_X(x_j))$, meaning the average performance of the training model is equal to that of the validation model. From here on afterwards, $\frac{1}{X}\sum_{i}^{X} e(f_i(x_i) - \frac{1}{Y}\sum_{j}^{Y} e(f_X(x_j)) < 0$, which suggests weaker generalisation and thus overfitting. This observation is further backed up by the accuracy plot in figure 32 as it can be seen that validation accuracy plateaus around epoch 14.

The output prediction loss curves are fairly unintuitive, with very low initial learning. Good learning then occurs from around epoch 7 to 14. Given more time, it would have been experiment with modifying the learning rate hyperparameter of the optimiser. From the behaviour of the loss curves, it could be hypothesised that a decaying learning rate with a higher initial learning rate may achieve better performance. Additionally, experimenting with the kernel initialisation could produce better initial learning.

Listing 21: New proposed implementation of the optimiser.

```
# default hyperparameters of Adam optimiser in Keras
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay
    =0.0, amsgrad=False)
# proposed new optimiser
initial_lr = 1e-2
epochs = 30
opt = Adam(lr=initial_lr, beta_1=0.9, beta_2=0.999, epsilon=None, decay=
    initial_lr/epochs, amsgrad=False)
```

The output prediction model was then used to predict labels using input vectors from the unseen test dataset. The three evaluation metrics were then calculated, as seen in table 2. Recall is the least targeted metric as false negative predictions do not need to be heavily penalised since they are still positive returns in reality. However, precision is important since this accounts for the false positives, which are negative returns in reality that have been incorrectly predicted. As mentioned on Google's developer course[10], precision and recall are often traded off against each other i.e. increasing precision results in decreasing recall. Theoretically, it could then be possible to improve the empirical performance of the model by allocating a higher classification threshold, as demonstrated in figure 33.

| Accuracy | Precision | Recall |
|---|---|---|
| 0.705 | 0.700 | 0.699 |

Table 2: Evaluation metrics of the model performed on hold out test data.

After the prediction model was applied to the test data, statistical and visual results of the selected points were compared and contrasted to that of the entire test set. This somewhat represented the initial goal of benchmarking the project against the naive diversification heuristic since that benchmark results in exposure to every single return datapoint in the test set. As shown in table 3, the mean is comparable to the absolute return of the benchmark and the variance is similar to that of the volatility or risk of a portfolio of stocks. Using these measures, it could be interpreted that the model is capable of

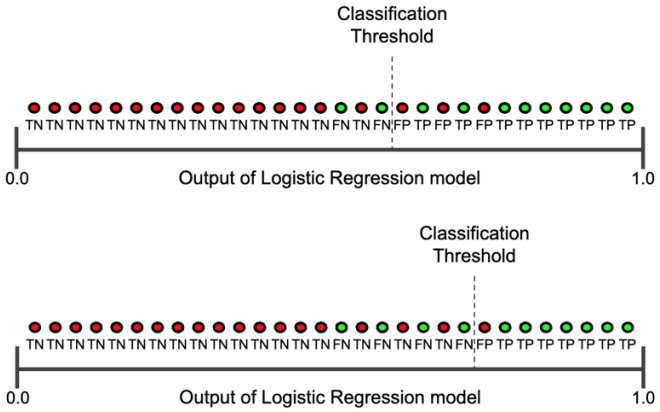



Figure 33: Top: Initial classification threshold resulting in Precision = 0.8, Recall = 0.73. Bottom: Higher classification threshold with results in Precision = 0.88, Recall = 0.64.

outperforming the benchmark by approximately 0.78% on average in terms of daily absolute return. Over the course of 252 trading days per year, the model would, on average, return $1.0076^{252} = 581\%$ return on investment (ROI) whereas the benchmark would be down approximately 3.6% ($(1 - 0.000147)^{252} = 0.96363$.) Please note that this is a highly trivialised estimation since it does not take into account either the variance or time sequential order of the returns.

More interesting is the difference in shape of the actual distributions. The selected points exhibit as positive skew whereas the general test set is negative. This suggests that a larger concentration of returns were higher than the mean return in the model selected points and conversely in the general test set. This reflects well on the performance of the model's prediction capabilities as it has correctly identified many more positive returns than negative. Additionally, variance is lower in the model selected data, indicating a potentially less volatile and higher Sharpe ratio portfolio. Kurtosis was also lower, suggesting lighter tails than in the general test set.

**Returns of Model Selected Datapoints**

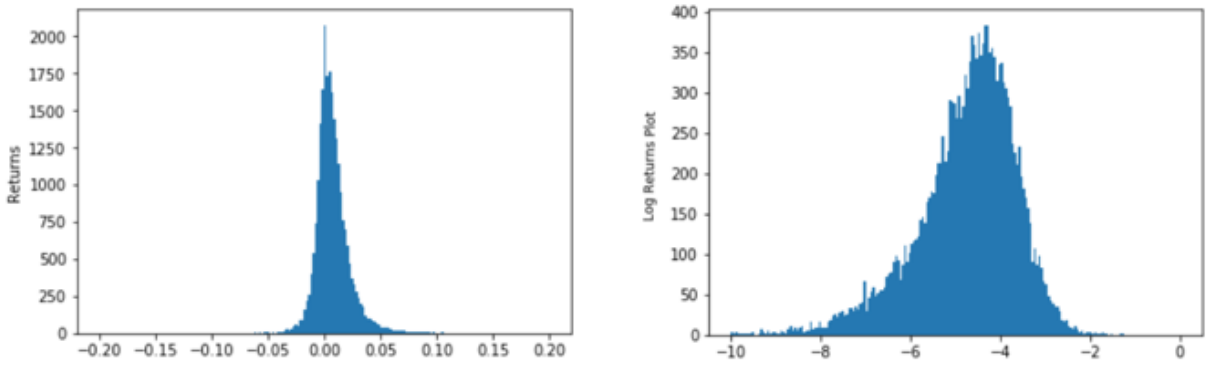



Figure 34: Histogram of the returns and log returns of the selected labels by the model.

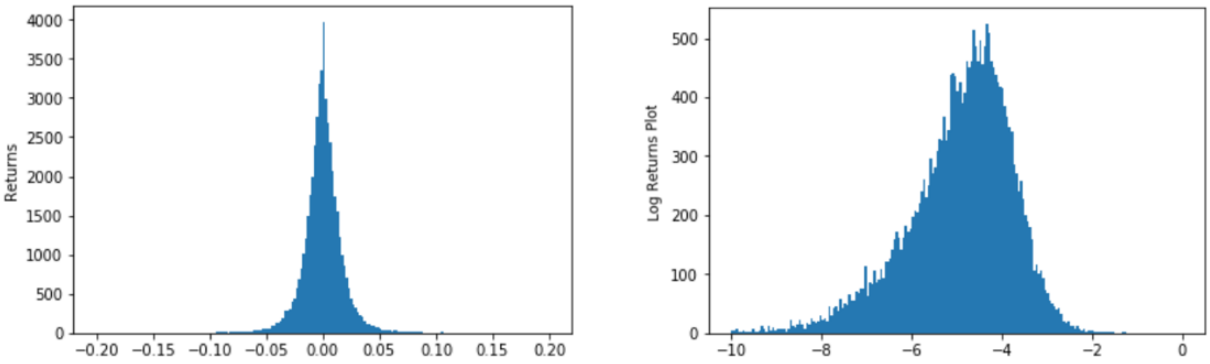



Figure 35: Histogram of the returns and log returns of the entire unseen test data set.

| | Entire Dataset | Test Dataset | Model Selected Data |
|---|---|---|---|
| Mean | 0.000389 | -0.000147 | 0.00764 |
| Variance | 0.000546 | 0.000364 | 0.000269 |
| Max | 7.098 | 0.278 | 0.277 |
| Min | -0.714 | -0.482 | -0.368 |
| Skewness | 64.114 | -1.489 | 1.224 |
| Kurtosis | 18474.170 | 43.882 | 31.110 |

Table 3: Statistical summary of percentage changes i.e. returns of sets of data. Entire Dataset equates to all returns from 2000 onwards, Test Data is the returns allocated to the test split and Model Selected Data is the collection of returns that the model predicted would have positive returns from the Test Dataset.

# 8. Conclusions

## 8.1. Objectives Completion

Below is an analytical breakdown into the extend of which the requirements capture, as specified in section 3.1, has been achieved or changed.

1. Data Preprocessing: One of the more difficult aspects of the project was evaluating how to handle the data sparsity for a model which relied on consistent, time series data. The objective of preprocessing the data so that it had some detectable pattern that the model could learn was achieved, although it was very generic and blanket. For example, min-max scaling the inputs achieved the objective of scaling it correctly for the model, but begs the question of whether the scaling was appropriate for each individual attribute.
2. Building and Testing Benchmark: The .
3. Implementing and Testing Model Architecture: The implementation of a few models was successful in terms of functionality but disappointing in terms of performance. It quickly became obvious that Keras, although easy to understand and build with, offered minimal low level controls that were necessary for efficient and optimal architecture design. An example of this was the lack of skip connection implementation for models, which was an area that would very much have been explored if there was more time. It would only have been possible to implement this with the lower level TensorFlow library which is known to be fairly slow in terms of development speed.
4. Integration Testing and Debugging: This part of the project was successful achieved and extremely useful in debugging erroneous code. Testing helps signpost and indicate where faults occur although actual debugging often requires deeper analysis into data structures and function behaviour to truly solve the problem. One of the most useful aspects of testing was keeping track of object type conversions when calling functions from external packages. Although Python's dynamic datatypes allows for fast development, it was difficult to keep track of the types as the codebase grew. The linear flow of the program made it fairly easy to combine unit tests into integration tests.
5. Hyperparameter Optimisation: In depth research was done to arrive at the initialisation values for all of the hyperparameters but it was disappointing to see that adjustments failed to produce any marked improvement in the performance of the model. The project has taught that the fundamental structuring of data and the network is far more important for generating results and it is fairly useless to attempt to fine tune a poorly functioning model.

## 8.2. Evaluation

The main highlight achieved in this project was actually implementing a model which showed some signs of signs of outperformance in terms of prediction against the designated benchmark. Although, the input data did not solely rely on fundamental data as initially desired, the addition of technical analysis data was deemed vital for the success of the model. It could be argued that this reflects more accurately on the real world factors that influence the price of a stock - they are not just a function of the fundamental values of a company.

The project evaluates well a theoretical application of recurrent networks to the problem of predicting prices. However, it would be hard to translate any of the research generated into a practical investment application.

Although there is not a significant amount of structured research into price prediction models, the results align with those seen in various forecasting papers, such as Smyl's findings at Uber. It was believed that part of the poor performance of the model was due to the fact that not all the factors that affect the price were considered, especially at the noisy 'daily' time step. There is a

## 8.3. Future Work

There are countless future development paths that could be done to improve and generate more interesting results from this project. A few are listed below:

- Implementation of a real world application benchmark:

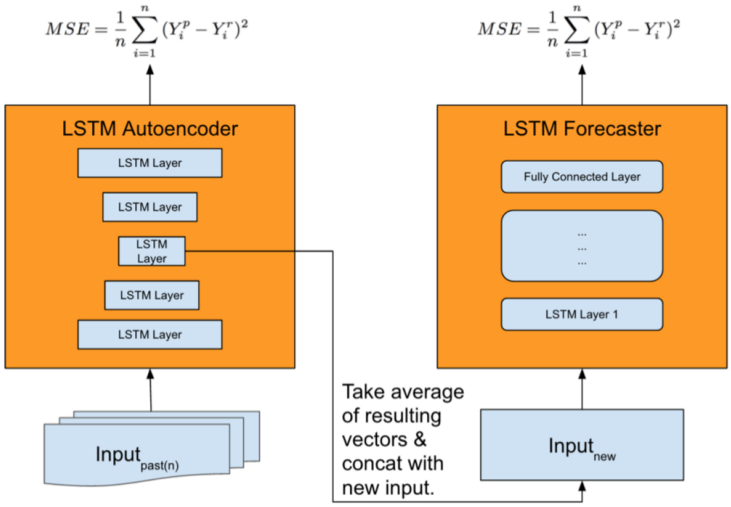



Figure 36: Feature selection and forecast models by Uber.

- Unsupervised learning algorithms:
- Further data cleaning and processing:
- Improving model design: Only LSTM networks were explored in this paper

# 9. Bibliography

# References


[1] *Jupyter Notebook Documentation*. Project Jupyter, 2018.

[2] H. Ahmadi. *Testability of the Arbitrage Pricing Theory by Neural Network*. Proceedings of the International Conference on Neural Networks, 1990.

[3] J. Alberg and Z. C. Lipton. *Improving Factor-Based Quantitative Investing by Forecasting Company Fundamentals*. arXiv, 2018.

[4] C. Bergmeir, R. J. Hyndman, and B. Koo. *A Note on the Validity of Cross-Validation for Evaluating Autoregressive Time Series Prediction*. Elsevier, 2017.

[5] J. Brownlee. *A Gentle Introduction to the Rectified Linear Unit (ReLU) for Deep Learning Neural Networks*. https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/, 2019.

[6] V. DeMiguel, L. Garlappi, and R. Uppal. *Optimal Versus Naive Diversification: How Inefficient is the 1/N Portfolio Strategy?* Oxford University Press, 2007.

[7] C. S. Eun, W. Huang, and S. Lai. *International Diversification with Large- and Small-Cap Stocks*. Journal of Financial and Quantitative Analysis, 2008.

[8] E. F. Fama and K. R. French. *Common risk factors in the returns on stocks and bonds*. Journal of Financial Economics, 1993.

[9] Y. Gal and Z. Ghahramani. *A Theoretically Grounded Application of Dropout in Recurrent Neural Networks*. arXiv, 2016.

[10] Google. *Classification: Precision and Recall*. https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall, 2019.

[11] B. Graham and D. Dodd. *Security Analysis*. Whittlesey House, 1934.

[12] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. *DRAW: A Recurrent Neural Network For Image Generation*. arXiv, 2015.

[13] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. D. Jesus. *Neural Network Design*. Martin Hagan, 2014.

[14] B. M. Henrique, V. A. Sobreiro, and H. Kimura. *Literature review: Machine learning techniques applied to financial market prediction*. Expert Systems With Applications, 2019.

[15] S. Hockreiter and J. Schmidhuber. *Long Short Term Memory*. MIT Press, 1997.

[16] K. jae Kim. *Financial time series forecasting using support vector machines*. Neurocomputing, 2003.

[17] K. Janocha and W. M. Czarnecki. *On Loss Functions for Deep Neural Networks in Classification*. arXiv, 2017.

[18] D. P. Kingma and J. L. Ba. *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*. arXiv, 2014.

[19] S. Makridakis, E. Spiliotis, and V. Assimakopoulos. *The M4 Competition: Results, findings, conclusion and way forward*. International Journal of Forecasting, 2018.

[20] T. Merz and P. Janus. *Low-Volatility Investing: Empirical evidence of the defensive properties of low volatility enhanced portfolios*. UBS, 2016.

[21] C. Olah. *Understanding LSTM Networks*. colah.github.io., 2015.

[22] R. Pascanu, T. Mikolov, and Y. Bengio. *On the difficulty of training Recurrent Neural Networks*. arXiv, 2013.

[23] J. Racine. *Consistent cross-validatory model-selection for dependent data: hv-block cross-validation*. Journal of Econometrics, 2000.

[24] N. Reimers and I. Gurevych. *Optimal Hyperparameters for Deep LSTM-Networks for Sequence Labeling Tasks*. arXiv, 2017.

[25] G. Robertson. *Machine Learning in Investment Management*. Man AHL, 2018.

[26] S. A. Ross. *The Arbitrage Theory of Capital Asset Pricing*. Journal of Economic Theory, 1976.

[27] G. Rouwenhorst and W. N. Goetzmann. *The Origins of Value: The Financial Innovations that Created Modern Capital Markets*. Oxford University Press, 2005.

[28] Ruder. *An overview of gradient descent optimization algorithms*. arXiv, 2017.

[29] D. E. Rumelhart, P. Smolensky, J. L. McClelland, and G. E. Hinton. *Schemata and Sequential Thought Processes in PDP Models*. Morgan Kaufmann Publishers, 1986.

[30] H. Sak, A. Senior, and F. Beaufays. *Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling*. Interspeech, 2014.

[31] W. F. Sharpe. *Mutual Fund Performance*. Journal of Business, 1966.

[32] A. Shaw. *Getting Started With Testing in Python*. https://realpython.com/python-testing/.

[33] S. Smyl, J. Ranganathan, and A. Pasqua. *M4 Forecasting Competition: Introducing a New Hybrid ES-RNN Model*. https://eng.uber.com/m4-forecasting-competition/, 2018.

[34] J. L. Treynor. *Market Value, Time, and Risk*. 1961.

[35] P. R. Winters. *Forecasting Sales by Exponentially Weighted Moving Averages*. Management Science, 1960.

## 10. Appendix

Keras source code can be found at this GitHub repository submodule. Documentation for the library can be found here.

Listing 22: Depiction of the list of external packages used to develop the model. These packages are roughly split by use case; mathematical structuring, data visualisation, functionalities required, machine learning tools and miscellaneous.

```
import numpy as np
import pandas as pd
import datetime as dt

import matplotlib.dates as mdates
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import MinMaxScaler, Normalizer
from scipy import stats

from keras.models import Sequential, Model, load_model
from keras.layers import Activation, Dense, LSTM, Dropout, Masking, Input
from keras import optimizers
from keras.utils.vis_utils import model_to_dot

from IPython.display import SVG

from google.colab import files

import os
```

### 10.1. Source Code

The project repository, which contains the codebase can be found on GitHub with the link here. The code can be found in both its original Jupyter Notebook (.ipynb) and python file (.py) formats. It is suggested to use the Google Colaboratory environment, which integrates all of the required libraries. The original file can be found here. Please see the README file and notebook text cells within the GitHub repository for documentation on the functionality of the codebase.