Ian Tai

Learning the Stock Market: Deep Learning and Sentiment Analysis-Based Stock Price Prediction

---

Computer Science Tripos – Part II

Trinity College

May 7, 2018

Proforma

Name:	**Ian Tai**
College:	**Trinity College**
Project Title:	**Learning the Stock Market: Deep Learning and Sentiment Analysis-Based Stock Price Prediction**
Examination:	**Computer Science Tripos Part II, May 2018**
Word Count:	[1]
Project Originator:	**Ian Tai**
Supervisors:	**Dr Sean Holden**
	Prof Stephen Satchell

Original Aims of the Project

Deep Learning has increasingly been applied to many fields of industry. Among Finance, the applications of Deep Learning in predicting stock market prices is an increasingly popular research field. This dissertation proposes a method of stock price prediction using a combination of Long Short-Term Memory Recurrent Neural Networks and a variety of Sentiment Analysis techniques. The project aims to apply this method to collected news headlines from selected news agencies on Twitter and stock price data from the latter two quarters of 2017.

Work Completed

This project has been successful; all success criteria have been met. I collected, parsed, and converted financial data from a Bloomberg Terminal. I built a data collection and processing system for the Twitter dataset. I implemented Gaussian and Multinomial Naive Bayes Classifiers, and used the Scikit-Learn library for implementing Multi-Class Support Vector Machines and Semi-Supervised Support Vector Machines for sentiment analysis. Finally, I built a Long Short-Term Memory Recursive Neural Network for stock

[1] This word count was computed by TEXcount

price prediction. The techniques proposed for predicting stock market prices have resulted in statistically significant results, and may benefit financial and machine learning research in this field.

Special Difficulties

Finding appropriate Twitter datasets proved to be more difficult than was foreseen, which led to trying various sources of data, different models of classification, and ultimately, manual classification of the data for train/test purposes.

Declaration

I, Ian Tai of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: Ian Tai

Date: May 7, 2018

Contents

1 Introduction **1**
- 1.1 Motivation & Aims . . . 1
- 1.2 Challenges . . . 2
- 1.3 Related Work . . . 2

2 Preparation **3**
- 2.1 Neural Networks . . . 3
  - 2.1.1 Introduction . . . 3
  - 2.1.2 Artificial Neurons . . . 4
  - 2.1.3 Training . . . 4
  - 2.1.4 Recurrent Neural Networks . . . 7
  - 2.1.5 Long Short-Term Memory Networks . . . 8
  - 2.1.6 Deep LSTM Network . . . 10
- 2.2 Sentiment Classifiers . . . 11
  - 2.2.1 Support Vector Machines . . . 11
  - 2.2.2 Naive Bayes . . . 13
  - 2.2.3 Rejected Approach . . . 15
- 2.3 Requirements Analysis . . . 16
- 2.4 Choice of Tools . . . 17
  - 2.4.1 Programming Language . . . 17
  - 2.4.2 Libraries . . . 17
  - 2.4.3 Development and Testing . . . 18
- 2.5 Starting Point . . . 18
- 2.6 Software Engineering and Approach . . . 19
- 2.7 Summary . . . 20

3 Implementation **21**
- 3.1 Sentiment Analysis . . . 21
  - 3.1.1 Rejected Approach . . . 21
  - 3.1.2 Data . . . 22
  - 3.1.3 Features . . . 23
  - 3.1.4 Multinomial Naive Bayes . . . 23
  - 3.1.5 Gaussian Naive Bayes . . . 23
  - 3.1.6 Support Vector Machine . . . 23

3.2 Price Prediction . . . 23
3.2.1 Data . . . 23
3.2.2 Features . . . 23
3.2.3 Long Short-Term Memory . . . 23

4 Evaluation **25**
4.1 Overall Results . . . 25
4.2 Hyperparameters . . . 25
4.3 Testing . . . 25
4.4 Sentiment Evaluation . . . 25
4.5 Prediction Evaluation . . . 25
4.6 Summary . . . 25

5 Conclusion **27**
5.1 Results . . . 27
5.2 Lessons Learned . . . 27
5.3 Further Work . . . 27

Bibliography **27**

A Latex source **31**

B Project Proposal **33**

List of Figures

2.1 Plots of sigmoid, RELU, and hyperbolic tangent functions 5
2.2 Layout of an MLP 6
2.3 States of a dynamic system 7
2.4 A simple RNN, in both its cyclic and unraveled form 7
2.5 Diagram of an LSTM cell 9
2.6 Example of the kernel trick 11
2.7 Overview of project components 19

Acknowledgements

I would like to thank the following people for the help they have given me:

TODO: COMPLETE AFTERWARDS

Chapter 1

Introduction

TODO: Beginning Starting Intro

This dissertation describes the implementation, fusion, and testing of several Machine Learning (ML) techniques for stock price prediction. Multinomial Naive Bayes, Gaussian Naive Bayes, and Multi Class Support Vector Machines, using different methods of feature extraction, were used to classify the sentiment of relevant Twitter posts of news headlines. The Long Short-Term Memory (LSTM) Recurrent Neural Network (RNN) was used for predicting stock prices, utilizing both financial technical indicators and the output of the sentiment classifiers.

1.1 Motivation & Aims

Under the Efficient Market Hypothesis, an efficient capital market fully reflects all relevant information in determining security prices [1]. Since all past prices and information are available to the market, it follows that correct predictions, if possible, will be immediately taken advantage of, and the resulting market adjustment quickly nullifies any possible advantage. Thus, it follows that an efficient market is unpredictable, and behaves in the manner of a Random Walk.

However, several factors contribute to potential inefficiency in practical markets – there usually exists some latency in market information availability and absorption, varying liquidity levels of different commodities and equities, different levels of information availability, and countless other real-life factors. Thus, it is argued that there exists a window for prediction. [Cite several of these studies]

The wealth of available data regarding financial markets makes it a prime target for Machine Learning.

1.2 Challenges

1.3 Related Work

Chapter 2

Preparation

This chapter includes all of the work that was completed before implementation began. This includes the theory behind each Machine Learning model (§2.1, §2.2.1, and §2.2.2), an outline of the project's requirements (§2.3), tools and libraries considered and used (§2.4), the starting point for my project (§2.5), and the development model and early outline for implementation (§2.6).

2.1 Neural Networks

The theory regarding Recursive Neural Networks and Long Short-Term Memory Networks in this section is partially adapted from [2, 3, 4, 5].

2.1.1 Introduction

Before delving into Recurrent Neural Networks (RNNs), we must first discuss the basics of Machine Learning (ML). An ML algorithm is one that learns from data. Mitchell (1997) provides a definition for learning: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."[6]

The use of a recurrent neural network in my project is in the manner of supervised learning. Supervised learning is a subset of ML, where all given data is labelled, such that each input, a vector of **features** (distinguishing numerical characteristics of the input) $\mathbf{x} \in \mathbb{R}^m$, is associated with a label y. This label can either be a discrete variable where $y \in C = [C_1, C_2, C_3, ..., C_n]$, in the case of classification, or a continuous variable where $y \in \mathbb{R}$, in the case of regression. The use of RNNs in this project is limited to regression.

A supervised learning algorithm for regression, given a training sequence

$$\mathbf{s} = ((\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), (\mathbf{x_3}, y_3), ..., (\mathbf{x_n}, y_n)) \tag{2.1}$$

where $(\mathbf{x_1}, y_1)$ is a training input, learns a function $h : \mathbb{R}^m \to \mathbb{R}$, a hypothesis, that approximates a match between an input feature vector $\mathbf{x}$ to a result y.

After learning this hypothesis function, the algorithm can then be used to predict test samples $\mathbf{x}'$. We can then run accuracy measurements and goodness-of-fit tests on the resulting outputs, given we know the original labels y'.

2.1.2 Artificial Neurons

A neural network is a network of artificial neurons that each takes an input $\mathbf{x}$, and runs a linear combination of the input feature vector $\mathbf{x}$, weights $\mathbf{w}$, and a bias b, fed through an activation function σ. The formulation is detailed below:

$$y = \sigma(\mathbf{w}\mathbf{x} + b) \tag{2.2}$$

The activation function is used to ensure the output stays within a preset bound, usually either $[0, 1]$ or $[-1, 1]$. Typical activation functions are as follows:

Logistic Function (Sigmoid):

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

Rectified Linear Unit (RELU):

$$\sigma(x) = \max(0, x) \tag{2.4}$$

Hyperbolic Tangent (tanh):

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

The output is then fed as either an input to other artificial neurons, or is the overall output of the neural network.

2.1.3 Training

The training of the network using the training set first includes initialization of the weights and biases. Initialization will be covered in depth in the Implementation section for the Long Short-Term Memory RNN (§3.2.3).

For a basic NN, such as the Multi-Layer Perceptron (MLP), the output of the NN is observed by simple **feed-forward propagation**. This means in the directed graph of

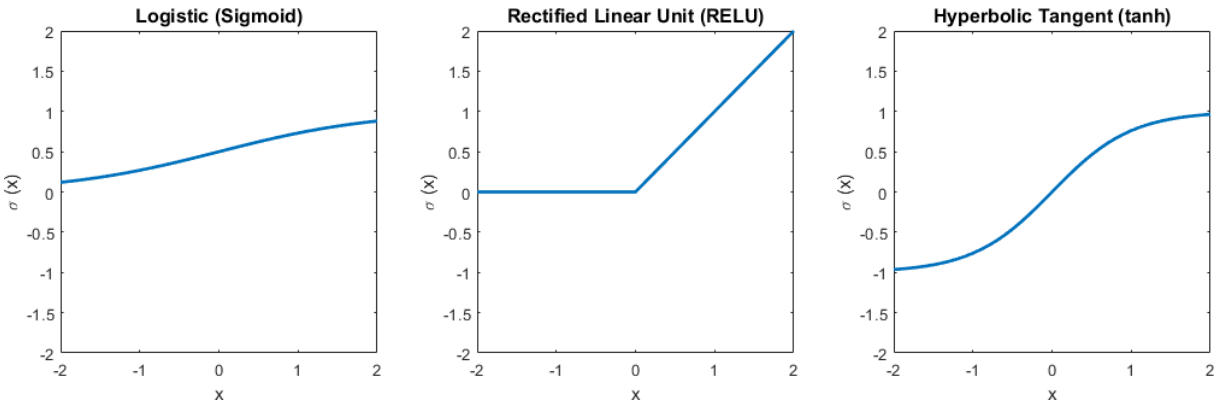

Figure 2.1: Plots of sigmoid, RELU, and hyperbolic tangent functions

the NN architecture, calculated values for each training example ($\mathbf{x_i}$) flow from neuron to neuron across layers, and the eventual output is run through a final activation function to obtain the final result.

This output (y_i') can then be fed into the **loss function** to calculate the error for this training example. A typical loss function for this purpose is the Sum of Squared Errors function[1]:

$$\mathrm{E}(\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{b}) = \sum_{i=1}^{n} (y_i - y_i')^2 \tag{2.6}$$

where $\mathbf{w}$ and $\mathbf{b}$ specify the weights and biases of the NN respectively, $\mathbf{x}$ is the set of training examples, $\mathbf{y}$ is the set of training labels, n is the number of training examples, and y_i and y_i' are the training label and output value for training example i, respectively.

Based on the output of the loss function, the NN can then adjust its weights and biases to minimize loss. This adjustment is performed using **gradient descent**. This optimization algorithm iteratively updates these parameters using the gradients calculated from the loss function until convergence is reached. Different variations of gradient descent used for the Long Short-Term Memory network (LSTM) will be discussed further in the Implementation section for LSTMs (§3.2.3).

The gradients are calculated as follows:

$$\mathbf{w_{t+1}} = \mathbf{w_t} - \lambda \frac{\delta E(\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{b})}{\delta \mathbf{w}}\bigg|_{\mathbf{w_t}} \tag{2.7}$$

$$\mathbf{b_{t+1}} = \mathbf{b_t} - \lambda \frac{\delta E(\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{b})}{\delta \mathbf{b}}\bigg|_{\mathbf{b_t}} \tag{2.8}$$

where $\mathbf{w_{t+1}}$ and $\mathbf{b_{t+1}}$ are the weight and bias vectors at iteration t, respectively. The parameter λ is known as the learning rate, a hyperparameter[2] for NNs. The learning rate

[1] Adapted from Dr Sean Holden's Machine Learning and Bayesian Inference Part II course [7]

[2] Hyperparameters are configurable settings for a Neural Network that are often manually tuned for optimal performance and desired run-time

must be chosen carefully; if it is too small, the NN may take an exceedingly long time to reach convergence during the training period, and if too large, the NN may never find an appropriate minimum.

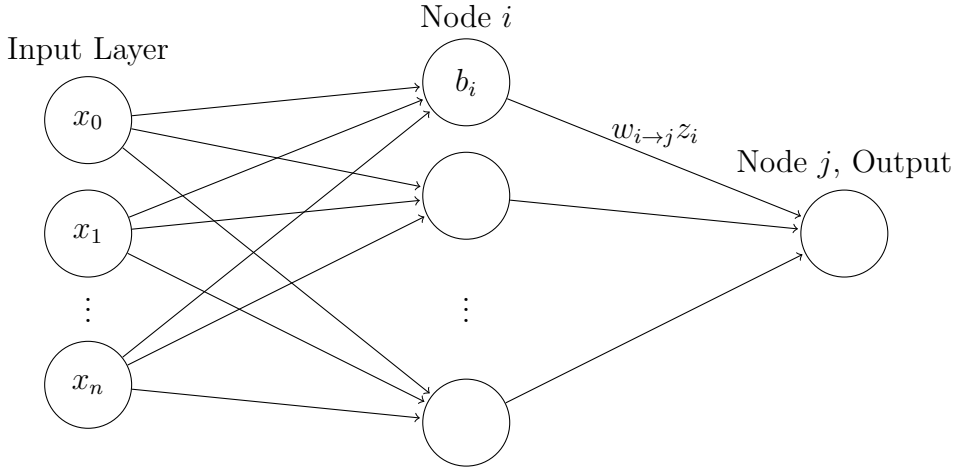

Figure 2.2: Layout of a Multi-Layer Perceptron[3]

The weights and biases at various levels of the NN are updated using **backpropagation**. Backpropagation is the method used for calculating $\frac{\delta E(\mathbf{x},\mathbf{y},\mathbf{w},\mathbf{b})}{\delta \mathbf{w}}$ and $\frac{\delta E(\mathbf{x},\mathbf{y},\mathbf{w},\mathbf{b})}{\delta \mathbf{b}}$ for every weight $w_{i\to j}$ and bias b_i, as shown in Figure 2.2. This can be calculated directly, since the gradient at each node in a layer is independent when the chain rule is applied, using the calculated gradients of any nodes that take the input from node j. Thus, we apply backpropagation to find the gradient of each weight, starting from the output node and working our way backwards until we reach the input feature vectors. This can be summarized into the following:

$$\frac{\delta E(\mathbf{x},\mathbf{y},\mathbf{w},\mathbf{b})}{\delta w_{i\to j}} = z_i \sigma_j(a_j) \sum_k \delta_k w_{j\to k} \tag{2.9}$$

where

$$a_j = \sum_k w_{k\to j} z_k \tag{2.10}$$

where k is any node that takes the output of node j, z_i is the value output of node i, σ_j is the activation function for the node j, and δ_k is the gradient for $w_{k\to j}$ that has already been calculated. This formulation for calculating the weights' gradients can include the gradients for the bias, if we simply label the bias as an extra weight, multiplied by an arbitrary $z = 1$.

[3] $w_{i\to j}z_i$ is the weight associated with the output of node i as the input to node j multiplied by the output of node i, z_i. b_i is the bias associated with node i. Adapted from Dr Sean Holden's Artificial Intelligence I Part IB course [8]

2.1.4 Recurrent Neural Networks

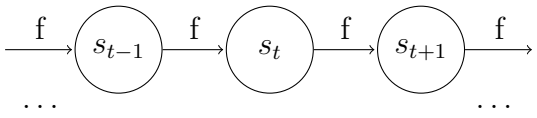

Figure 2.3: The evolution of state in a system[4]

Feed-forward networks require that each training example be input into the network in full, and that each example be independent of each other. If we have the case of sequentially dependent inputs, such as the state of a dynamic system that evolves with regards to a time t (Figure 2.3), then we can use a Recurrent Neural Network.

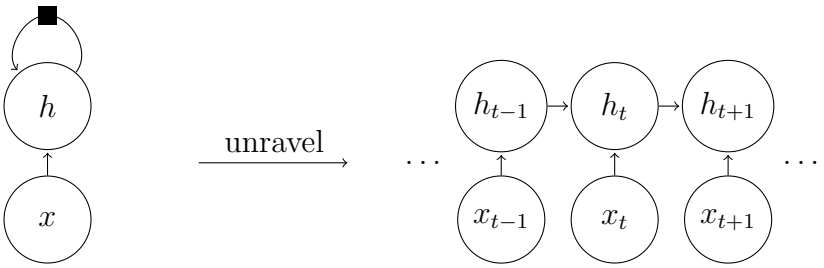

Figure 2.4: A simple RNN, in both its cyclic and unraveled form[5]

In a feed-forward NN, there are no cycles in the network and all values simply propagate towards the output node. A Recurrent Neural Network (RNN) introduces the concept of cycles, such that the output of a node at time t may be used as an input to a node at time $t+1$. Figure 2.4 shows an example of a simple RNN. From the cyclic graph on the left, we can simply unravel the graph to have an acyclic computational graph.

There are several different design pattern for RNNs, each for a different use case:

- RNNs that produce an output at each time step and have recurrent connections between hidden units at different time steps[6]
- RNNs that produce an output at each time step and have recurrent connections only from an output at a previous time step to hidden units at the next time step
- RNNs that take a whole sequence of inputs before producing a single output

For the purposes of stock price prediction, the first design pattern is most useful. This is because we expect our RNN to extract useful information about the state of the stock

[4]Each node represents the state at some time t, which is mapped according to some underlying function f, to the state at time $t+1$

[5](Left) The cyclic representation of an RNN. The rectangle in the cyclic arrow represents a delay of 1 time step. (Right) The unraveled acyclic representation of the same RNN. This RNN processes information from a system at some time t along with the input passed from the network at time $t-1$, and uses this as an input for the network at time $t+1$

[6]Interestingly, such an RNN of a finite size can compute any function computable by a Turing Machine[2]

market and hold this information within its hidden units, and this information may not be fully conveyed if we simply take the relevant output.

The relevant equations regarding the hidden state and output of an RNN are as follows:

$$\mathbf{h}^{(t)} = \tanh(\mathbf{b}_h + \mathbf{W}_h h \mathbf{h}^{(t-1)} + \mathbf{W}_x h \mathbf{x}^{(t)}) \tag{2.11}$$

$$\mathbf{o}^{(t)} = \mathbf{b}_o + \mathbf{W}_o h \mathbf{h}^{(t)} \tag{2.12}$$

where $\mathbf{h}^{(t)}$ represents the hidden state of the RNN unit at time t, and $\mathbf{o}^{(t)}$ represents the output of the RNN unit at time t. The various $\mathbf{W}$s represent represent the weights associated with the previous time step's hidden value, the current input, and the current hidden state value, respectively. The variables $\mathbf{b}$ represent the biases of the hidden state and the output, respectively[7].

A similar backpropagation algorithm to what was described in §2.1.3 can be used to train such an RNN – however, we must also be backpropagating through the time steps. This algorithm is called **backpropagation through time** (BPTT). This is simply the application of the previous backpropagation algorithm to the unraveled computation graph of the RNN.

The key problem with the RNN that prevents it from being used in its purest form is the **vanishing/exploding gradient problem**. The vanishing/exploding gradient problem is the tendency for gradients that are propagated through many stages to either vanish (become negligible) or explode (become disproportionately large). While the former is only a problem with trying to model long-term dependencies, the latter can completely disrupt the parameter optimization algorithm. The problem of the vanishing gradient follows naturally, even without any unstably small gradients – the weights given to long-term dependencies are exponentially smaller than those given to short-term dependencies due to the repeated application of the weight of the hidden unit over many time steps.

2.1.5 Long Short-Term Memory Networks

The LSTM is perhaps the most popular solution to the vanishing/exploding gradient problem of vanilla RNNs[8]. The LSTM, a gated RNN, is based on the idea of creating paths for dependencies through time where the derivatives will neither vanish nor explode, by introducing variability to the connection weights between time steps and a learned method of forgetting the old state.

The original proposal by Hochreiter and Schmidhuber for the LSTM states that it solves the vanishing/exploding gradient problem by "an efficient, gradient-based algorithm

[7]For reference, the weights are matrices, thus presented in upper case

[8]Other solutions, such as the Gated Recurrent Unit (GRU), are not within the scope of this project and will not be discussed

for an architecture enforcing constant (thus, neither exploding nor vanishing) error flow through internal states of special units (provided the gradient computation is truncated at certain architecture-specific points)"[3]. This algorithm was the inclusion of an internal recurrence in addition to the outer occurrence of an RNN. This internal recurrence, the 'state' in Figure 2.5 is preserved in the cell across time.

This design was later improved upon by Gers, Schmidhuber and Cummins, who observed that the "LSTM fails to learn to correctly process certain very long or continual time series that are not *a priori* segmented into appropriate training subsequences with clearly defined beginnings and ends"[4]. This is because the values of the LSTM cell state can grow without bound if the input stream is continuous. Thus, their paper introduced the forget gate, whose role is to learn to reset the LSTM cell's memory contents when the information is not needed anymore. Figure 2.5 shows the resulting computational graph for an LSTM cell with the addition of forget gates.

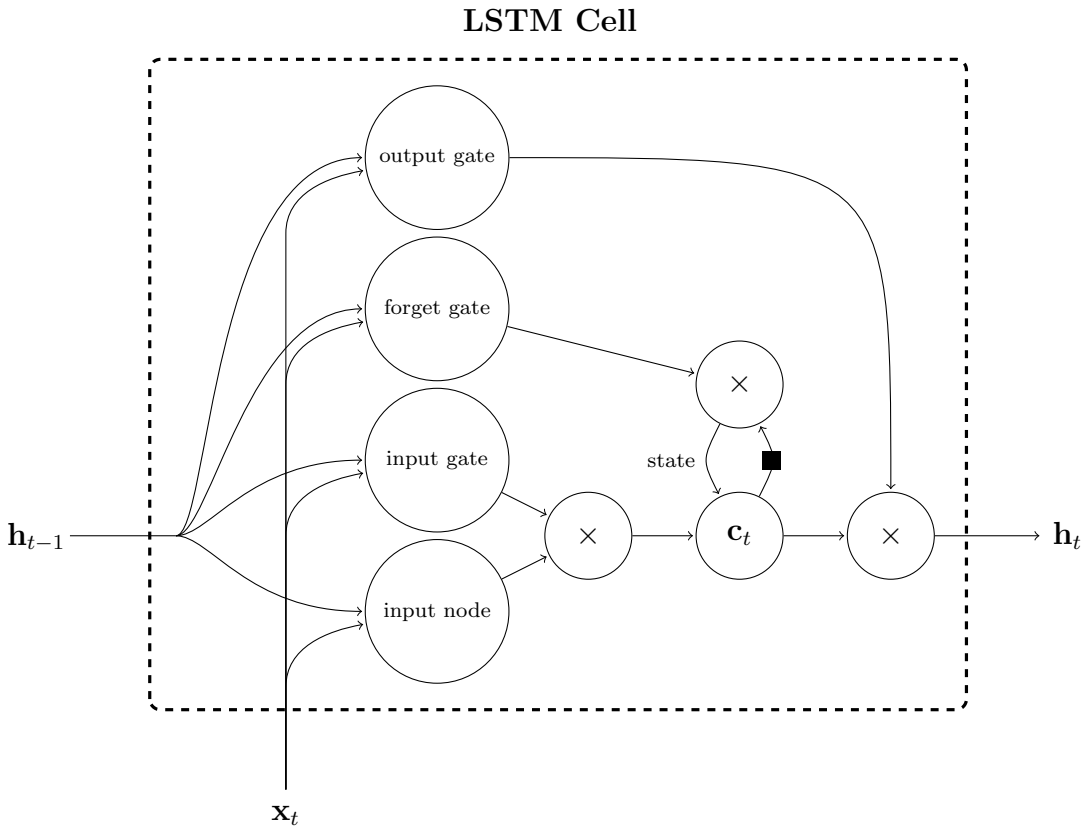

Figure 2.5: The computational graph of an LSTM cell[9]

Figure 2.5 shows the computational graph of an LSTM cell at time t, where $\mathbf{x}_t$ is the input vector at time t, $\mathbf{c}_t$ is cell activation vector at time t, and $\mathbf{h}_t$ and $\mathbf{h}_{t-1}$ are the outer occurrences (and outputs) of the RNN at times t and $t-1$, respectively.

[9]The black rectangle on the arrow between the looping nodes represents a delay of 1 time step.

The formulation for the various gates, nodes, and outputs shown are as follows:

- Input gate
$$\mathbf{i}_t = \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{W}_{ci}\mathbf{c}_{t-1} + \mathbf{b}_i) \tag{2.13}$$

- Forget gate
$$\mathbf{f}_t = \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{W}_{cf}\mathbf{c}_{t-1} + \mathbf{b}_f) \tag{2.14}$$

- Cell activation
$$\mathbf{c}_t = \mathbf{f}_t\mathbf{c}_{t-1} + \mathbf{i}_t \tanh(\mathbf{W}_{xc}\mathbf{x}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \tag{2.15}$$

- Output gate
$$\mathbf{o}_t = \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{W}_{co}\mathbf{c}_{t-1} + \mathbf{b}_o) \tag{2.16}$$

- Output
$$\mathbf{h}_t = \mathbf{o}_t \tanh(\mathbf{c}_t) \tag{2.17}$$

where σ is the logistic sigmoid function, and all of the gates and the outputs are vectors of the same size. The weight matrices ($\mathbf{W}$) in each gate are diagonal, so each element m in the gate vector only receives input from the respective element m of the input vectors.

2.1.6 Deep LSTM Network

In 2013, Graves et. al showed that LSTMs provide much stronger prediction capabilities if 'stacked', in the form of hidden layers between the input and output, similar to the manner of typical deep neural networks[5]. This is what we will call a **deep LSTM network**. The structural implication of stacking layers is such that the outputs of a layer are fed as the inputs to the next layer, and that each layer is simply a collection of independent LSTM cells. The advantage of such a model is that it is able to approximate functions that are non-linear – the more layers in the network, the more complex the possible approximation.

In this schema, the number of LSTM cells in each subsequent hidden layer is kept constant, so as to incorporate a direct single output-input relationship between cells of consecutive layers. The final output $\mathbf{y}_t$ of the LSTM network is then a final output calculation defined by:

$$\mathbf{y}_t = \mathbf{W}_{\mathbf{h}^N y}\mathbf{H}_t^N + \mathbf{b}_y \tag{2.18}$$

where N is the total number of layers in the LSTM network, and $\mathbf{H}_t^N$ is the matrix of the h_t vectors for all LSTM cells in layer N.

This deep LSTM network structure is what I chose as my primary learning model framework for stock price prediction. This framework's capacity to learn both long-term

and short-term dependencies in data makes it suitable for a large variety of learning tasks whose inputs are sequential, such as time series prediction, speech recognition, semantic parsing, and even composition of music[10]. Thus, it is a good candidate for tackling the learning problem of stock price prediction.

2.2 Sentiment Classifiers

2.2.1 Support Vector Machines

The theory behind Support Vector Machines in this section is adapted from [7].

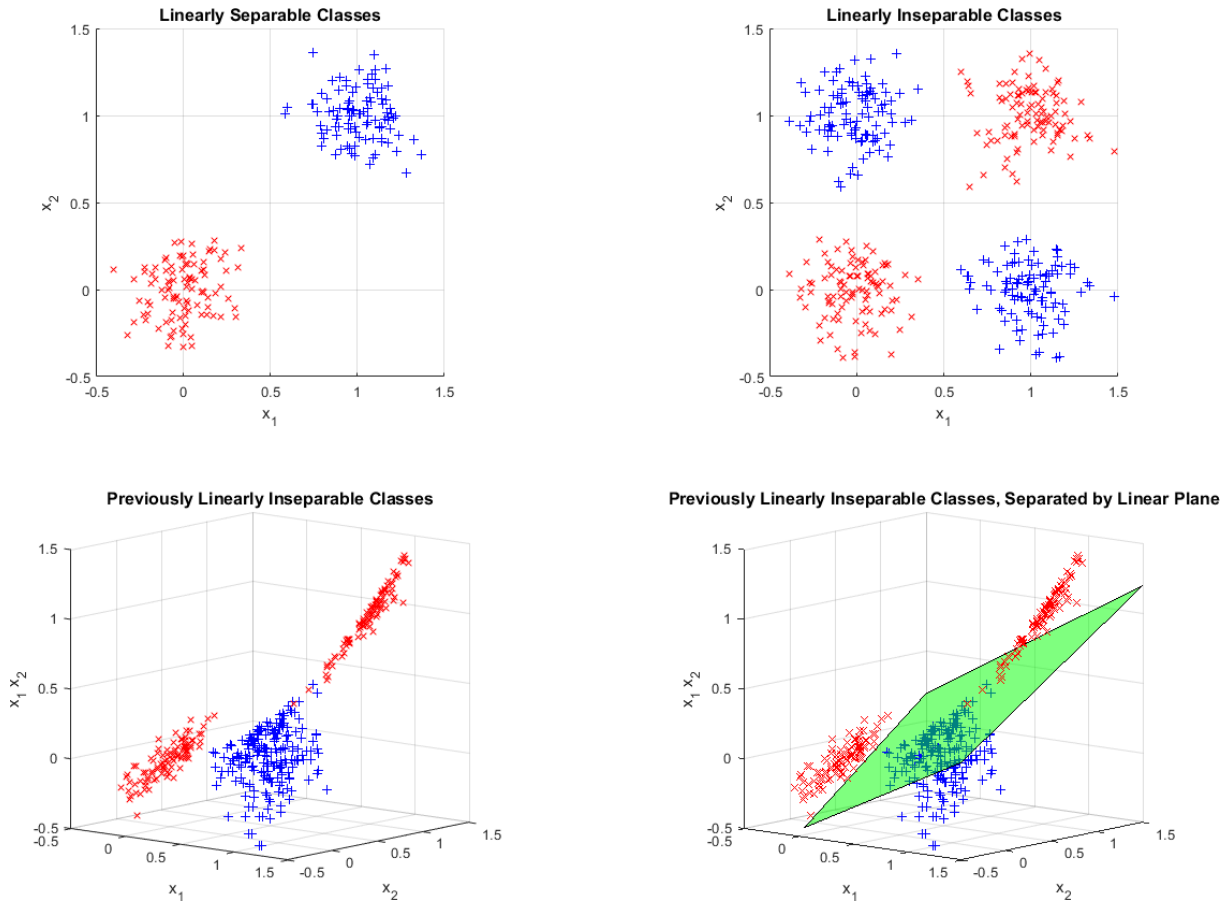

Figure 2.6: Separating linearly inseparable data by introducing an additional dimension[12]

Support Vector Machines (SVMs) are currently the gold standard for learning problems where the availability of data is not sufficient for a well-trained NN[9]. Because sentiment labelling of the Twitter dataset was performed manually, the SVM is a prime candidate for our problem of sentiment classification.

[10] `http://people.idsia.ch/~juergen/blues/`

[12] The additional dimension is $x_1 x_2$. The green plane indicates the plane of separation between classes

SVMs tackle the problem of efficiently computing a model for linearly inseparable data. Figure 2.6 displays the usage of the **kernel trick**, which is the introduction of new dimensions dependent on the original dimensions in order to increase linear separability. This is the main idea behind SVMs.

In order to understand SVMs, we must first delve into pattern recognition theory. A pattern recognition classifier estimates a function

$$\begin{aligned} f &: \mathbb{R}^N \to \{\pm 1\} \\ f(\mathbf{x}_n) &= y_n \end{aligned} \tag{2.19}$$

where N is the dimensionality of the input vector, and $\mathbf{x}_n$ and y_n are the input vector and correct label of the example n, respectively[10]. This function aims to correctly classify *new* examples – examples that were not included in the training set but are sampled from the same distribution. If we put no restrictions on the estimation of f, the created function may perform exceedingly well on the training data, but fail miserably on the unseen data. Such a classifier would be effectively useless in the real world. Statistical learning theory shows that we must restrict the class of functions that the machine can learn to one with a suitable *capacity* for our training data[11]. This capacity can be seen as the complexity of the function, in the sense that a highly complex function (e.g. high-degree polynomial) should not be used to model linearly correlated data.

The discriminator for an SVM is based on hyperplanes

$$h(\mathbf{x}) = \mathbf{w}\mathbf{x} + b \tag{2.20}$$

corresponding to the decision function:

$$f(\mathbf{x}) = \text{sign}(\mathbf{w}\mathbf{x} + b) \tag{2.21}$$

Note that the class labels for a typical SVM are either -1 or 1. The optimal hyperplane is defined as the hyperplane with the maximum separation between two classes, such that the Euclidean distances (or margins) between the closest examples of each class to the hyperplane are maximised. This optimal solution also has the lowest capacity.

In order to make this linear classifier non-linear, we can introduce **basis functions** ϕ_i[7]

$$\begin{aligned} \mathbf{\Phi}^T(\mathbf{x}) &= \left[\phi_1(\mathbf{x})\ \phi_2(\mathbf{x}) \dots \phi_k(\mathbf{x})\right] \\ h(\mathbf{x}) &= \sigma(\mathbf{w}^T\mathbf{\Phi}(\mathbf{x}) + b) \end{aligned} \tag{2.22}$$

These basis functions can then be transformed into kernels $K(\mathbf{x}_i, \mathbf{x})$:

$$\begin{aligned} h(\mathbf{x}) &= \text{sign}(b + \mathbf{w}^T \mathbf{\Phi}(\mathbf{x})) \\ &= \text{sign}(b + \sum_{i=1}^{m} \alpha_i y_i \mathbf{\Phi}^T(\mathbf{x}_i)\mathbf{\Phi}(\mathbf{x})) \\ &= \text{sign}(b + \sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})) \end{aligned} \tag{2.23}$$

where m is the total number of training examples, and $\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{\Phi}(\mathbf{x}_i)$ arises from solving the constrained optimization problem:

Minimise

$$\frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{m} \xi_i \tag{2.24}$$

such that

$$y_i f(\mathbf{x}_i) >= 1 - \xi_i \text{ and } \xi_i >= 0 \text{ for } i = 1, ..., m \tag{2.25}$$

by solving the Lagrangian

$$\begin{aligned} L(\mathbf{w}, b, \xi, \alpha, \lambda) = \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{m} \xi_i \\ - \sum_i \alpha_i (y_i f(\mathbf{x}_i) + \xi_i - 1) - \sum_i \lambda_i \xi_i \end{aligned} \tag{2.26}$$

where minimizing $\frac{1}{2}||\mathbf{w}||^2$ is equivalent to maximizing margins, ξ is the measurement of error(misclassification) and $C\sum_{i=1}^{m} \xi_i$ is minimised for minimal error for some hyperparameter C[13].

The two most common kernel functions are the Polynomial Function

$$K_{cd}(\mathbf{x}, \mathbf{x}') = (c + \mathbf{x}^T \mathbf{x}')^d \tag{2.27}$$

where c and d are hyperparameters, and the Radial Basis Function (RBF)

$$K_{\sigma^2}(\mathbf{x}, \mathbf{x}') = e^{-\frac{1}{2\sigma^2}||\mathbf{x}-\mathbf{x}'||^2} \tag{2.28}$$

where σ is a hyperparameter. We will be using the RBF kernel for sentiment classification.

2.2.2 Naive Bayes

When tackling a complex learning problem, it is good practice to start by using a simple model. A Naive Bayes (NB) model is an application of Bayes' theorem for classification based on conditional probability, with the 'naive' assumption that the random variables

[13]The exact methods for solving this optimization problem are beyond the scope of this project and will not be discussed

constituting features are **independent**. While this assumption isn't completely valid for most ML problems, it allows us to use an easily built model that often produces good results regardless of dependent features.

The key idea of NB is to classify an example

$$\mathbf{x} = [x_1\ x_2\ ...\ x_n] \tag{2.29}$$

as a class

$$C_k \in \{C_1, C_2, ..., C_K\} \tag{2.30}$$

by choosing the class C_k representing by

$$\Pr(C_k|x_1\ x_2 ...\ x_n) \tag{2.31}$$

which is the probability that an example is of class C_k conditioned on its features, which are assumed to be independent [14]. By application of Bayes' theorem:

$$\Pr(C_k|\mathbf{x}) = \frac{\Pr(C_k)\Pr(\mathbf{x}|C_k)}{\Pr(\mathbf{x})} \tag{2.32}$$

In practice, we are only concerned with the numerator, since $\Pr(\mathbf{x})$ is not dependent on the class C_k. $\Pr(C_k)\Pr(\mathbf{x}|C_k)$ is equivalent to $\Pr(\mathbf{x}\ C_k)$, and thus:

$$\begin{aligned}
\Pr(\mathbf{x}\ C_k) &= \Pr(x_1\ x_2\ ...\ x_n\ C_k) \\
&= \Pr(x_1|x_2\ ...\ x_n\ C_k)\Pr(x_2\ ...\ x_n\ C_k) \\
&= \Pr(x_1|x_2\ ...\ x_n\ C_k)\Pr(x_2|x_3\ ...\ x_n\ C_k)\Pr(x_3\ ...\ x_n\ C_k) \\
&\ \ \vdots \\
&= \Pr(x_1|x_2\ ...\ x_n\ C_k)\Pr(x_2|x_3\ ...\ x_n\ C_k)\ ...\ \Pr(x_n\ C_k) \\
&= \Pr(x_1|x_2\ ...\ x_n\ C_k)\Pr(x_2|x_3\ ...\ x_n\ C_k)\ ...\ \Pr(x_n|C_k)\Pr(C_k)
\end{aligned} \tag{2.33}$$

Since we made the assumption that each feature is independent

$$\Pr(x_1|x_2\ ...\ x_n\ C_k)\Pr(x_2|x_3\ ...\ x_n\ C_k)\ ...\ \Pr(x_n|C_k)\Pr(C_k) = \Pr(C_k)\prod_{i=1}^{n}\Pr(x_i|C_k) \tag{2.34}$$

Our classifier's output is then $h = C_k$ for the class satisfying

$$\underset{k}{\text{argmax}} \quad \Pr(C_k)\prod_{i=1}^{n}\Pr(x_i|C_k) \tag{2.35}$$

The probability $\Pr(C_k)$ can estimated by counting the number of training examples in each class:

$$\Pr(C_k) \approx \frac{\text{count}(y = C_k)}{\text{count}(y)} \tag{2.36}$$

[14] Note that K denotes the total number of possible classes

The types of NB classifiers, such as Gaussian Naive Bayes and Multinomial Naive Bayes, differ in the assumed distributions of $\Pr(x_i|C_k)$.

For Gaussian NB, the likelihood of a feature is assumed to be Gaussian:

$$\Pr(x_i|C_k) \approx \frac{1}{\sqrt{2\pi\sigma_{C_k}^2}} \exp\left(- \frac{(x_i - \mu_{C_k})^2}{2\sigma_{C_k}^2} \right) \tag{2.37}$$

Similarly, for Multinomial NB, the likelihood of a feature is assumed to be Multinomial. This is used for when the features are simply counts of distinct events. The conditional probability is estimated[12]:

$$\Pr(x_i|C_k) \approx \frac{1 + \sum_j^n x_i^j}{1 + \sum_m^n \sum_j^n x_m^j} \qquad \forall j.\ y^j = C_k \tag{2.38}$$

where the addition of 1 to both the numerator and denominator is known as Laplace smoothing, or add-one smoothing. This is to prevent the overall probability for an example belonging to a class being 0 if one or more features had never been seen for examples of the class.

A further optimization to combat decimal underflow is to instead calculate the log of this conditional probability, since log is a monotonically increasing function and no probability for any class should $= 0$.

$$\underset{k}{\text{argmax}} \quad \Pr(C_k) \prod_{i=1}^{n} \Pr(x_i|C_k) \tag{2.39}$$

$$= \underset{k}{\text{argmax}} \quad \log\Big(\Pr(C_k) \prod_{i=1}^{n} \Pr(x_i|C_k) \Big) \tag{2.40}$$

$$= \underset{k}{\text{argmax}} \quad \log\big(\Pr(C_k) \big) + \sum_{i=1}^{n} \log\big(\Pr(x_i|C_k) \big) \tag{2.41}$$

2.2.3 Rejected Approach

As an initial approach to the sentiment analysis problem, **semi-supervised learning** was attempted on a large Twitter dataset, as a possible alternative to supervised learning. This meant a much larger dataset was used, with only a small subset of the data being labelled. The classifier was an iterative application of an SVM on the labelled dataset, where it would find the unlabelled examples that were clearly separated by the decision boundary and assign the predicted label to those examples. The training process was then re-run with the newly autonomously labelled examples in the training set, until all examples were eventually labelled by the SVM. This process is known as **Label Spreading**.

§3.1.1 gives an overview of the implementation of this model, and the reasons why it was ultimately scrapped.

2.3 Requirements Analysis

Having studied the basics of Recurrent Neural Networks, Support Vector Machines, and Naive Bayes, the project requirements were decided. The aim of the project is to perform sentiment analysis on Twitter news headlines, and use the results as well as financial time-series data to predict stock market prices with an LSTM. This splits the project into two main logical groups – the sentiment analysis and the stock price prediction.

Sentiment Analysis

The most important goal of the Sentiment Analysis part of the project is to have a high enough accuracy that the output is useful for the stock price prediction. In terms of functional requirements, several classifiers must be implemented, and from those I will select the most suitable one for use with the LSTM. These classifiers will be the Support Vector Machine, Multinomial Naive Bayes, and Gaussian Naive Bayes.

Because of the need for Twitter data that matches both the time period and subject matter of the financial data, I will build a data scraping and processing system to gather my dataset. I will also be experimenting with several feature extraction methods from the Twitter dataset, which are Doc2Vec[13] and Bag of Words.

Stock Price Prediction

The ultimate goal of the project is to be able to predict stock prices more accurately than a random agent is able to. This accuracy is a performance measurement of the LSTM model used for predicting stock prices. The functional requirements for the price prediction portion of the project are that the sentiment classification output must combine with the financial data, a feature extractor for financial indicators must be built, and that the LSTM must function properly, with well-tuned hyperparameters.

A summary of the requirements for this project is shown in table 2.1.

Requirement	Priority	Risk	Difficulty
Multinomial Naive Bayes	Medium	Medium	Medium
Gaussian Naive Bayes	Medium	Medium	Medium
Support Vector Machine	High	Medium	Low
Twitter data processing	High	Medium	Medium
Compare and evaluate Twitter feature extraction methods	Low	Low	Medium
Compare and evaluate sentiment analysis models	High	Low	Medium
Combine sentiment classifier output and financial data	High	Medium	Medium
Feature extractor for financial data	Medium	Low	Medium
Long Short-Term Memory Network	High	High	High
Hyperparameter tuning	Medium	Low	Medium

Table 2.1: Goals of the project

2.4 Choice of Tools

2.4.1 Programming Language

For machine learning projects, it is quite standard to use `Python`, because of its support for useful ML libraries (such as Tensorflow, Keras, Theano), its ease of use and readability, and relaxed type system. Though the language is not as fast as other languages such as `Java` and `C++`, many of its powerful libraries use `CPython`, which is a reference implementation and interpreter of `Python` in `C`. This allows these libraries to exploit the performance of `C` while maintaining the benefits of `Python`. I have chosen to use `Python` as the main language for my project, while also using `MATLAB` for some trivial plotting and mathematical tasks.

2.4.2 Libraries

The main third party libraries used are shown in table 2.2. Tensorflow[15] and Scikit-learn[16] are of particular importance. Both libraries are well-documented and widely used in industry, with the latter often being used for prototyping. GetOldTweets[17] is not an official library, but is an open-source project for scraping Twitter searches to retrieve Tweets that are older than allowed by the official Twitter API's query limit.

[15]https://www.tensorflow.org/
[16]http://scikit-learn.org/
[17]https://github.com/Jefferson-Henrique/GetOldTweets-python

Library	Version	Use	License
Tensorflow	1.5.0	Building LSTM	Apache 2.0 License
Scikit-learn	0.19.0	Using SVM	BSD-new License
Numpy	1.14.0	Efficient array manipulation	BSD-new License
Scipy	0.19.1	Probability functions for NB	BSD-new License
Pandas	0.20.3	Data representation and manipulation	BSD-new License
GetOldTweets	Unknown	Twitter web scraping	MIT License

Table 2.2: Main libraries used in this project

2.4.3 Development and Testing

Development was carried out on my personal laptop, which runs Windows 10. Because of its relatively powerful graphics card (Nvidia GTX 1060), it was also capable of training and testing the LSTM models.

For revision control, I used `git`, the revision control system that is almost ubiquitous in industry. I created and synchronised repositories on GitHub[18], for both my project and the dissertation write-up. This was used not only for planning and tracking progress, but also as a back-up tool.

2.5 Starting Point

The Computer Science Tripos has several courses that proved useful for my project. These are outlined in table 2.3. In particular, Machine Learning and Bayesian Inference provided me with the base knowledge required for understanding SVMs, Naive Bayes, and Neural Networks – however, personal study was required to learn about RNNs and LSTMs, as well as different hyperparameter tuning techniques.

My knowledge of `Python` prior to this project was intermediate, having used it only as a minor part of a prior internship. I had no experience with Tensorflow or other Deep Learning libraries. This meant I had to learn the design concepts behind Tensorflow, which is to construct mathematical operations for NNs as computational graphs operating on tensors.

[18]https://github.com, a free and popular `git` repository hosting service

Course	Application
Machine Learning and Bayesian Inference	Neural Networks, SVM
Natural Language Processing	Twitter feature extraction
LaTeX and MATLAB	Plotting and simple SVM model prototype
Mathematical Methods for CS	Understanding probability functions
Software Engineering	Iterative and incremental development model

Table 2.3: Tripos courses applicable to this project

2.6 Software Engineering and Approach

For a project of this scale, it is imperative to plan the different stages of development, and to compartmentalise the project. As was already outlined in §2.3, the project was split into two main sections: sentiment analysis and stock price prediction. An overview schematic of the project's components are show in Figure 2.7.

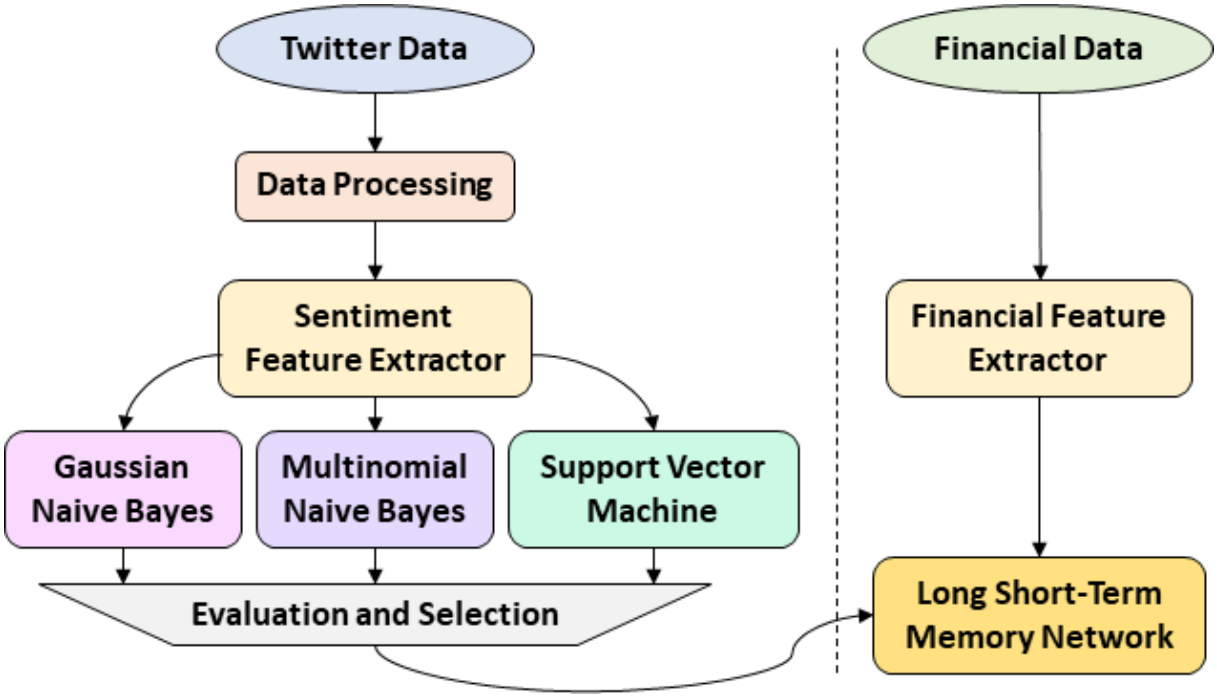

Figure 2.7: Overview of project components

The **Iterative and Incremental Development Model** was adopted for this project. The idea was to first build a 'walking skeleton', an runnable end-to-end system that has all of the essential components of the final end-product, though the components may simply be placeholders[14]. These components can then be properly implemented at a later date,

with the project's architecture already in place. These components were then iteratively tuned and improved.

The list of components to complete were as follows, ranked by implementation order:

1. Collect Twitter and financial data
2. Build Twitter data processing system
3. Manually label Twitter dataset
4. Implement sentiment feature extractors
5. Build Gaussian and Multinomial Naive Bayes
6. Build SVM
7. Evaluate and pick the best-performing classifier
8. Implement financial feature extractor
9. Build LSTM

The components described in steps 4-9 were then iteratively refined after initial completion.

2.7 Summary

In this chapter, I have discussed the relevant theory that had to be understood, the planning that occurred, before implementation was started. I gave an introduction to the concepts behind the machine learning models I will be implementing, and outlined the goals and key development strategies. I have also given an overview of the tools and libraries I will be using.

The next chapter will detail the implementation of the project, showing how the goals and plans outlined in this chapter were achieved.

Chapter 3

Implementation

This chapter details the implementation of the project, and how the goals and design principles outlined in the previous chapter were accomplished. The project proved to be a hefty undertaking, with the codebase totaling just over 6000 lines. I will be presenting the sentiment analysis portion of the project first, before moving into price prediction section. This was the order in which I implemented the project, as outlined in the previous chapter, and provides a logical flow between the components.

3.1 Sentiment Analysis

This section first gives an overview of the initial rejected approaches to the learning problem.

The rest of the section is discussed in the logical order of data flowing through the system – we discuss the collection of data and the pipelining process, before moving onto feature extraction, and finally the different sentiment classifiers.

3.1.1 Rejected Approach

This section details the initial approach to the sentiment analysis problem that was ultimately scrapped.

Semi-Supervised Learning

My first experimental forays into the sentiment analysis problem was to simply acquire as much relevant data as I could, and manually label a small subset to feed into a semi-supervised learning model. The initial reasons behind this decision were that there was a large abundance of official Twitter accounts of large news agencies, and that there may be a sufficiently large number of samples for a semi-supervised Deep Learning model, such as a **Hybrid Deep Belief Network**[15].

My approach to examining potential solutions to a learning problem is to first use a simple model in order to gain insight on whether the problem can feasibly be solved. Following this ethos, I used Scikit-Learn's semi-supervised learning module's label spreading implementation to test the feasibility of using a semi-supervised learning method for this dataset. This module uses the label spreading method mentioned in §2.2.3, combined with a standard SVM. After manually labelling around 2000 tweets, out of a total of around 500,000 tweets, I separated the labelled dataset into an 80:20 split for train:test, a standard ratio for partitioning data. The labelled 80with the rest of the unlabelled dataset and used to train the semi-supervised SVM. The trained model was then evaluated on the separated labelled test dataset.

The results gave very little confidence in moving forward with this technique, as the accuracy for sentiment classification was 54to be fed into the LSTM network as an input, it was imperative that the output be mostly correct, in order to ensure that inputs to the LSTM are kept as noise-free as possible. Thus, I decided to pursue a supervised learning approach to the sentiment analysis problem.

3.1.2 Data

Before collecting the data, key decisions were made regarding sources (Twitter accounts) and time-frame. The original plan was to collect a large amount of data (in the magnitude of 500,000 tweets), and to label a small subset of these tweets and feed it into a semi-supervised learning model. After experimenting with this technique, I quickly realized that it may be more effective to be selective about the data I was collecting in order to have a dataset that both was small enough to manually label, and also only provided relevant information to the price prediction model.

I settled on using tweets from the official Reuters Twitter account[1]. From my experiment with semi-supervised learning, I quickly realized that there was a great deal of redundancy when sampling from many news agencies, as any important news story is often broadcasted by most agencies. I chose Reuters because it is a leading provider of financial news, and most news regarding specific companies that were registered on the stock exchange of study, NASDAQ, were labelled with the corresponding ticker symbol[2]. In order to account for irregularities in tweet styles, I also performed a search for mentions of the company's name in each tweet.

Initial Approaches

My first approach to acquiring the Twitter dataset was to try to find labelled datasets from other studies that have been conducted regarding Twitter sentiment analysis on financial data. After failing to find any that were sufficiently related to the aims of my project, I moved to collect the data through Twitter's official web API. However, after

[1] `https://twitter.com/Reuters`

[2] A ticker symbol is the identifier given to a company on a stock exchange

discovering that the API calls for retrieving tweets that are older than a week were locked behind various paywalls, I decided to find a way around this issue.

Crawling

As was mentioned in §2.4.2, I used the GetOldTweets library for web scraping the relevant Twitter page. The aforementioned limitation on historical tweet retrieval prevented me from acquiring data that matched the full time period of my financial data, as my financial dataset required a sufficiently large number of samples for a Deep Learning model.

This library generates a search query URL on Twitter, which specifies the username, time frame, language, search query, and maximum number of returned tweets. An HTTP GET request is then sent to the search URL, and the JSON-formatted response of the search results in returned. This response only contains a page worth of results, so the process is repeated until all results are gathered. This response is then parsed and converted to a custom Tweet object, and returned from the original function call.

Pipeline

3.1.3 Features

3.1.4 Multinomial Naive Bayes

3.1.5 Gaussian Naive Bayes

3.1.6 Support Vector Machine

3.2 Price Prediction

3.2.1 Data

3.2.2 Features

3.2.3 Long Short-Term Memory

Chapter 4

Evaluation

4.1 Overall Results

4.2 Hyperparameters

4.3 Testing

4.4 Sentiment Evaluation

4.5 Prediction Evaluation

4.6 Summary

Chapter 5

Conclusion

5.1 Results

5.2 Lessons Learned

5.3 Further Work

Bibliography

[1] Burton G. Malkiel. Efficient market hypothesis. In John Eatwell, Murray Milgate, and Peter Newman, editors, *Finance*, pages 127–134. Palgrave Macmillan UK, London, 1989.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Sepp Hochreiter and Jrgen Schmidhuber. Long short-term memory. 9:1735–80, 12 1997.

[4] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural Computation*, 12:2451–2471, 1999.

[5] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

[6] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[7] Sean B. Holden. Machine learning and bayesian inference, 2018. `https://www.cl.cam.ac.uk/teaching/1718/MLBayInfer/ml-bayes-18.pdf`.

[8] Sean B. Holden. Artificial intelligence 1, 2018. `https://www.cl.cam.ac.uk/teaching/1718/ArtInt/ai1-18.pdf`.

[9] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems. *J. Mach. Learn. Res*, 15(1):3133–3181, 2014.

[10] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28, 1998.

[11] Vladimir Naumovich Vapnik. An overview of statistical learning theory. *IEEE transactions on neural networks*, 10(5):988–999, 1999.

[12] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[13] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196, 2014.

[14] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams: A Human-Powered Methodology for Small Teams*. Pearson Education, 2004.

[15] Shusen Zhou, Qingcai Chen, and Xiaolong Wang. Fuzzy deep belief networks for semi-supervised sentiment classification. *Neurocomputing*, 131:312 – 322, 2014.

Appendix A

Latex source

Appendix B

Project Proposal