Ian Tai

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

Computer Science Tripos – Part II

Trinity College

May 1, 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 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.

[1]This word count was computed by TEXcount

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 1, 2018

Contents

List of Figures

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, and §2.3), financial theory (§2.4), feature extraction (§2.5), an outline of the project's requirements (§2.6), tools and libraries considered and used (§2.7), development model (§2.10), an early outline for implementation (§2.9), and the starting point for my project (§2.8)

2.1 Neural Networks

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

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." [3]

The use of a recurrent neural network in my project is in the manner of supervised learning. Supervised learning is a subset of Machine Learning, where all given data is labelled, such that each input feature vector $\mathbf{x} \in \mathbb{R}^m$ 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:

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

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):

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

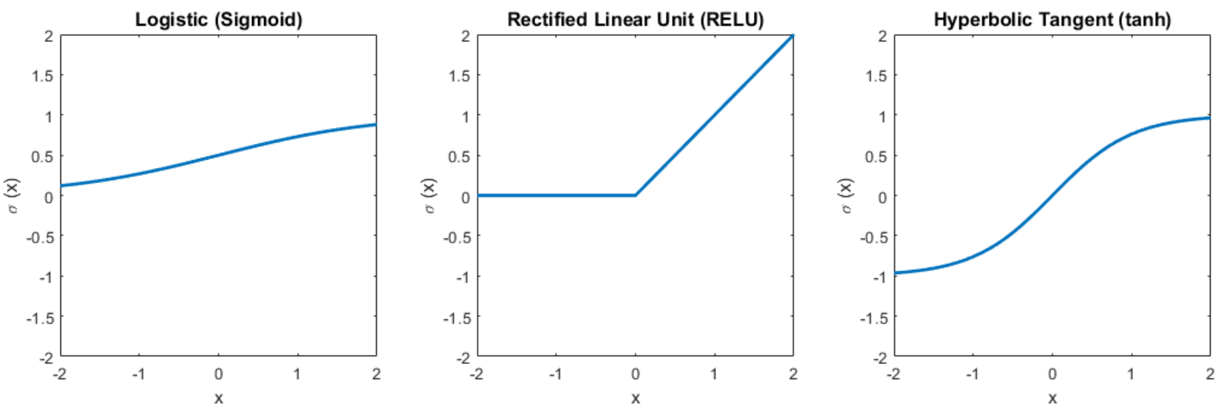

Figure 2.1: Plots of Sigmoid, RELU, and Hyperbolic Tangent functions

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 (§3.2.2).

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 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 (§3.2.2).

The gradients are calculated as follows:

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

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

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 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.

[1]Adapted from Dr Sean Holden's Machine Learning and Bayesian Inference Part II course [4]

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

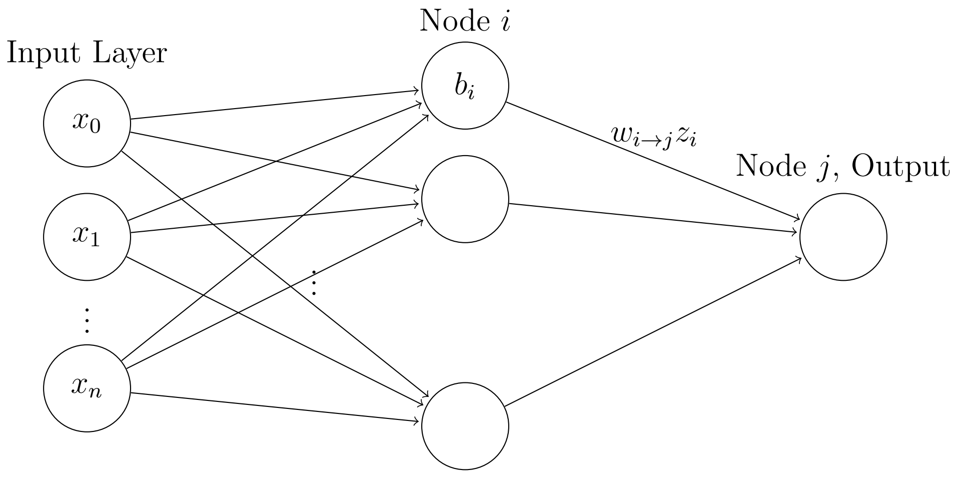

Figure 2.2: Layout of a Multi-Layer Perceptron[3]

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 [5]

2.1.4 Recurrent Neural Networks

$$\begin{tikzcd} \cdots \arrow[r, "f"] & s_{t-1} \arrow[r, "f"] & s_t \arrow[r, "f"] & s_{t+1} \arrow[r, "f"] & \cdots \end{tikzcd}$$

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.

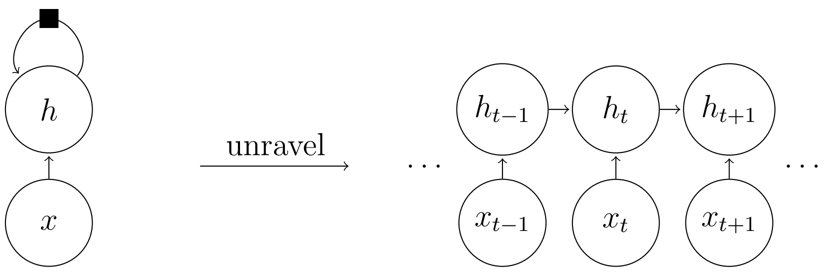

Figure 2.4: A simple RNN, in both its cyclic and unraveled form[5]

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:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} \tag{2.11}$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)}) \tag{2.12}$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)} \tag{2.13}$$

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 variables $\mathbf{W}$, $\mathbf{U}$, and $\mathbf{V}$ represent the weights associated with the previous time step's hidden value, the current input, and the current hidden state value, respectively. The variables $\mathbf{b}$ and $\mathbf{c}$ represent the biases of the hidden state and the output, respectively[7].

A similar backpropagation algorithm to what was described in **??** 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 in its simplest form 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 one of several attempted solutions to the vanishing/exploding gradient problem of vanilla RNNs[8][6, 7]. 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.

[7]For reference, the weights are presented in upper case and the biases in lower case simply for ease of reading

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

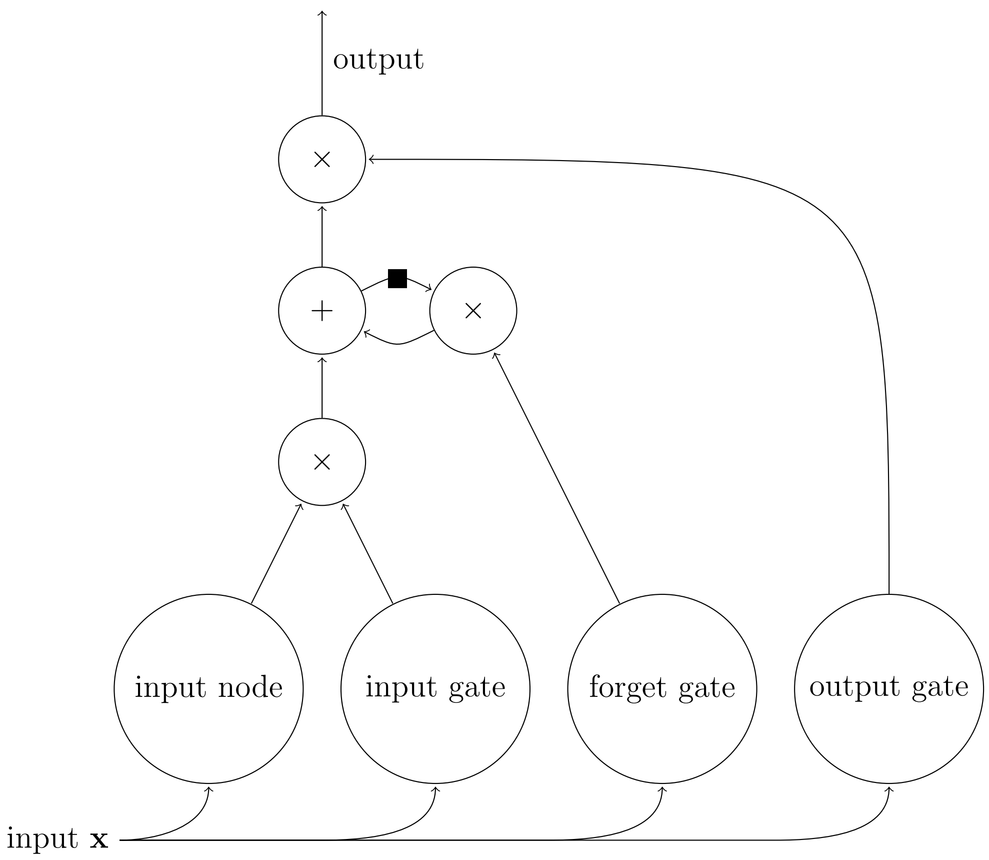

Figure 2.5: The cyclic form of an LSTM cell[9]

[9]The rectangle in the cyclic arrow represents a delay of 1 time step.

2.2 Support Vector Machines

2.3 Naive Bayes

2.4 Financial Markets

2.5 Feature Extraction

2.6 Requirements Analysis

2.7 Choice of Tools

2.8 Starting Point

2.9 Implementation Approach

2.10 Software Engineering Techniques

2.11 Summary

Chapter 3

Implementation

3.1 Sentiment Analysis

3.1.1 Data

Crawling

Pipeline

3.1.2 Features

3.1.3 Multinomial Naive Bayes

3.1.4 Gaussian Naive Bayes

3.1.5 Support Vector Machine

3.2 Price Prediction

3.2.1 Data

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

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

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

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

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

Appendix A

Latex source

Appendix B

Project Proposal