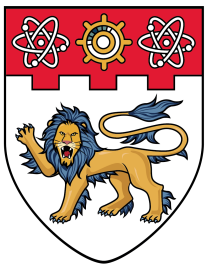


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Stock Market Portfolio Allocation Using Twitter Sentiment Analysis

Kulik Jan
Pooja Grewal Kaur
Rachel Loo Jia Yin
Wisely Neo Boon Hao

School of Physical and Mathematical Sciences

2022

Abstract

The market is driven by people and people by emotions. Emotions of individuals are easily swayed by external factors, such as social media and financial news. Recently, there have been numerous events where sentiments have been more influential in driving stock prices than any other factors pertaining to fundamental or technical aspects of the stock. Two notable examples are cryptocurrency mania and Gamestop. We believe that if the sentiment of an asset is quantified it will be possible to trade that asset profitably and minimise risk. In order to do that, unstructured tweets are first cleaned and Word2Vec is used to extract qualitative representation of tweets' features. Data are partitioned using training-test rate of 80:20. Furthermore, three models – Random Forest, Long Short-Term Memory (LSTM) and Transformers are trained and tested using the pre-processed data. Subsequently, we developed two trading strategies to utilise the acquired sentiment scores. Those strategies are then back-tested using walk-forward optimisation method and evaluated using several performance metrics. Finally, we estimate the cost of running the model and minimal investment.

Table of Contents

Introduction

Stock market prediction is of interest to many people and entities as it provides significant investment opportunities for individuals and companies to grow their assets. Stock traders predict trends in the stock market to determine when to buy or sell a stock in an attempt to maximise profits. If these trends are adequately predicted, they are able to earn a considerable profit margin. The financial markets, however, are influenced by a plethora of factors ranging from economic to national policies, disasters and seasons. The stock market has long been regarded as one of the most volatile markets and consequently, is difficult to predict. Traditionally, it was predicted using quantitative variables, mainly on the stock and the market conditions. In this report we adopt a different strategy and attempt to analyse the Twitter sentiment associated with particular stocks in order to minimise the risk of investment.

Methods

2.1 Datasets

A total of 2 datasets related to financial tweets were sourced from Kaggle, the first contains over 23,000 unlabeled tweets that will be used to train the Word2Vec model for feature generation. The second is a labeled dataset of over 5,000 tweets that will be used to train, validate and test models. Stock symbols were scraped over various sources and used to transform datasets for consistency. We transformed stock tickers to the value of "@stock" and numbers from integers to the word "@number" since numbers are arbitrary in this case and might lead to increased inconsistencies.

2.1.1 Unsupervised Dataset

This dataset consists of financial tweets that have a mix of stocks, currencies, and commodities. To generate features for the model consistent with the target outcome, we filtered the dataset with the scraped stock tickers that match up with those in the Nasdaq stock exchange. Subsequently, we did a simple exploratory data analysis (EDA) to check that there were a total of 23,598 tweets available to be fed into the feature-generating algorithm.

2.1.2 Supervised Dataset

A similar pre-processing algorithm was applied to this supervised dataset and a value of 5,791 tweets were returned for training, validation, and testing. Before splitting the data, we performed EDA on the dataset to obtain more information.

2.1.2.1 Labeled Sentiment Comparison

From the following graph it can be observed that there is a slight imbalance between negative sentiments and positive sentiments.

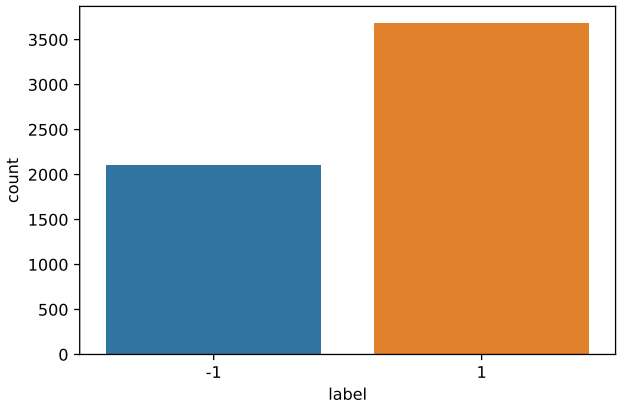
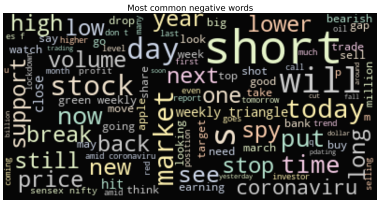


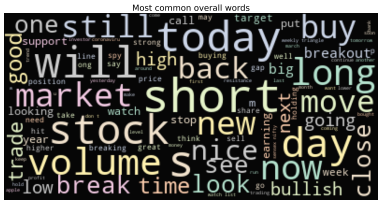
Figure 2-1: Countplot for Supervised Data

We then generated a wordcloud to visualize words associated with their respective sentiments and overall occurrence. Additionally, "@stock" and "@number" from the above mentioned transformation are removed as they appear too excessively. Figure 2-2 displays the outputs of the wordclouds, where the sizes of each word is determined to have a correlation with their respective classes.



(a) Negative Sentiment Cloud

(b) Positive Sentiment Cloud



(c) General Cloud

Figure 2-2: WordClouds

2.1.3 Testing Dataset

A final dataset is manually obtained. To apply and test the effectiveness of the best model on real world data, we scraped tweets from Twitter together with stock prices of the same day.

2.2 Portfolio Allocation

The problem of portfolio allocation has been a subject of ongoing research and analysis for years. It stems from the fact that the stock market is a highly complex system which renders it difficult to model and predict [1]. Moreover, the stock market is constantly evolving, driven by a variety of factors such as politics, pandemics and technology.

The traditional approach to portfolio allocation treats investors as if they were "unemotional" and only made rational decisions [2]. However, modern research suggests that this is in fact false and investors, to a certain extent, are driven by the sentiment towards companies [3]. Therefore, it is possible to quantify that sentiment and attempt to use that information in order

to predict the market movement and allocate portfolio accordingly.

Evaluation of collective sentiment towards a particular stock can be efficiently performed on social media platforms such as Twitter. It has enabled regular investors as well as bigger entities to share their thoughts, predictions and analyses about the stock market. Furthermore, many investors seek financial advice on social media thus potentially taking those opinions into consideration when creating their financial strategy. This results in a complex social structure, where opinions and beliefs can eventually become influential enough to have a real impact on the stock market thus impacting the price [4].

2.3 Investment Strategy

First, we need to establish a strategy to evaluate sentiment. For this purpose, we construct several models using Natural Language Processing (NLP) techniques. Those models are subsequently compared and the most accurate one is used to compute tweets' sentiment. The model processes each tweet and assigns a sentiment score, which essentially is the probability of the sentiment conveyed in a particular tweet being positive.

Furthermore, Twitter's key-word search functionality is used in order to retrieve tweets corresponding to a particular stock that is being analysed. Sentiment score of each stock is evaluated by searching for all tweets from a particular period that contained the respective ticker symbol. The models can only evaluate general sentiment of a tweet, so if multiple stocks are mentioned it is impossible for them to distinguish whether the sentiment towards a particular one is positive or negative. Therefore, those tweets have to be filtered out. After analysing all available tweets over the desired period, we averaged individual sentiment scores coming from single tweets in order to obtain the final sentiment score of a stock, which is represented by Equation 2.1

$$\mathrm{SS} = \frac{1}{N} \cdot \sum_{i=1}^{N} \mathrm{IVS}_i \tag{2.1}$$

where SS is the sentiment score, N is the total number of tweets corresponding to a particular stock and IVS is the individual sentiment score. Once the method of assigning sentiment scores is developed, it is possible to apply several strategies in order to use that information. These strategies are discussed in the following subsections.

2.3.1 Rebalance Strategy

The first important issue is defining the period over which the sentiment score is aggregated and portfolio rebalanced. Evaluation has to be performed at regular intervals after which a decision whether a particular stock should be bought, held or sold is made. Previously conducted studies have shown that the impact of sentiment on the overall stock market is rather short, around a span of one day [5]. Furthermore, Katayama et al. [6] evaluated the performance of a daily, weekly and monthly rebalance strategy in the context of sentiment analysis and concluded that the former significantly outperforms the latter two. Therefore, we decided to use the daily rebalance strategy.

Due to a limited number of tweets available each day it is crucial to select a subset of stocks to be considered such that there is enough data available to make an informed decision. Therefore,

we decided to only consider stocks from the S&P100 index, which contains 101 leading US stocks. Those are the most popular stocks that are widely discussed on English-speaking social media platforms and thus, there should be enough data available to consistently evaluate the sentiment of each of those stocks.

2.3.2 Utilising Sentiment Score

There are various strategies that can be applied in terms of utilising the obtained sentiment scores. The first one involves using the absolute value of sentiment. After evaluating all sentiment scores for the S&P100 index we select some number of stocks with highest scores to be included in the portfolio. If one of the selected stocks is being held in the portfolio from the previous day, no further action towards that stock is taken. If one of the stocks that is being held in the previous portfolio is not among the selected stocks, it is sold. Finally, if the stock is selected but is not contained in the previous portfolio, it is bought.

There are two hyperparameters that have to be tuned for this strategy: the number of stocks that are bought every day and the minimal number of tweets for the sentiment score to be considered as valid. The second threshold stems from the fact that some stocks are much more popular than others and thus, the number of available tweets can differ significantly. If the number of tweets is very low, the aggregated sentiment score might fail to represent the real sentiment towards a particular stock. Therefore, it is necessary to filter out those instances. Every time the number of available tweets about a certain stock is below the pre-defined threshold, we set its sentiment score to 0 in order to disregard it from the subsequent analysis. In case there are not enough non-zero sentiment scores available to meet the number of stocks defined by the first hyperparameter, only the stocks with non-zero scores are included.

The second strategy involves considering a daily change in the value of sentiment rather than the absolute value. This approach stems from the fact that some stocks consistently have lower sentiment scores than others, but they still might be worth investing in. Therefore, we apply a very similar approach to the previously described one, but the percentage change of sentiment with respect to the previous day is used instead of the absolute value of sentiment.

2.3.3 Portfolio Weighting

After the set of stocks to be included in the portfolio is selected, it is necessary to establish the weighting between them. First, it is possible to assign an equal weight to each of the stocks, as in Equation 2.2 [6]

$$w_{eq,i,T} = \frac{1}{N} \tag{2.2}$$

where N is the number of selected stocks, i is the index of the stock and T is the point in time. Moreover, it is possible to perform portfolio weighting based on the market capitalisation, as in Equation 2.3 [6]

$$w_{cap,i,T} = \frac{NAV_i, T}{\sum_{i=1}^{N} NAV_{i,T}} \tag{2.3}$$

where $NAV_{i,T}$ is the market capitalisation of stock i at time T, N is the number of selected stocks, i is the index of the stock and T is the point in time. Katayama et al. [6] investigated

both weighting methods and concluded that the first one performs better both in terms of the expected revenue and risk. Therefore, we choose it as the final portfolio weighting method.

2.4 Backtesting

After the model is ready, a backtest has to be performed in order to tune the hyperparameters and evaluate the actual performance. Due to the relatively small amount of data, we decided to perform a walk-forward optimisation, which is shown in Figure 2-3.

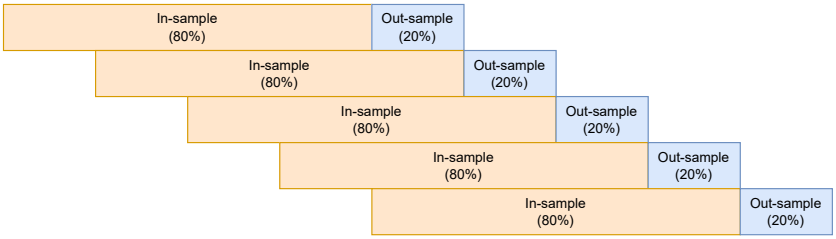


Figure 2-3: Walk-forward optimisation

The algorithm involves splitting the data into some number of folds and subsequently finding the optimal values of hyperparameters on the in-sample data of each fold. Then, the optimised model is tested on the out-sample data in order to obtain an estimation of the actual performance. Then, these steps are repeated for each fold. This procedure allows to use a much bigger portion of the available data while still providing meaningful results. The final model is backtested using a walk-forward optimisation with 10 folds.

Models

3.1 Word2Vec

This section introduces Word2Vec, an unsupervised method used to produce word embeddings. Word embedding is a set of language modeling and feature generation techniques under NLP. Word embedding captures the context of a word in a document, semantic and syntactic similarity, and relative association with other words in the document. In NLP, texts from the documents are mapped to vectors.

3.1.1 Background

Developed in 2013, it was discovered that the language model can be trained in two steps: using a simple model to train continuous word vectors followed by an N-gram feedforward neural network model [7]. Two methods were proposed then, the first being Continuous Bag of Words (CBOW) and the other being Continuous Skip-Gram Model. This project will be focusing on the use of the latter. The skip-gram model uses the current word to predict the context of surrounding words within a set window. An increased range of the window creates better word vectors but more complex computation is required [7]. In other words, the current word is the input layer with contextual words at the output. Numerous studies have used a number between 50 to 300 as the number of dimensions for their word embeddings, where it is also noted that performance for intrinsic tasks maximises at approximately 300 dimensions [8]. Since more dimensions require more computation, this project will use 100 dimensions for each word embedding due to time and computational constraints.

3.1.2 Implementing the model

The model used is from the notable "gensim" package. As mentioned in the previous chapter, a large amount of financial tweets was used to train the Word2Vec model to generate word embeddings. The generated word vector matrix then output dimensions of 6062 by 100, where each word is represented by a vector of 100 dimensions. In addition to the dimensions specified, a window size of 5 is used.

3.2 Random Forest

3.2.1 Implementation and Results

The Random Forest model was trained using the aforementioned Word2Vec model. The dataset was first split into training and testing sets, before mapping each tweet to a feature vector by averaging the word embeddings of all words in the tweet. We took the average of the word embeddings of the word vectors in a given tweet as the tweets dataset was of variable-lengths. By taking the individual word vectors and transforming them into a feature set that is the same length for every tweet, the variable-length tweet issue was resolved.

Since each word is a vector in a 100 dimensional space, vector operations can be used to combine the words in each tweet. We calculated the average feature vectors for training and

testing sets and used them to train a Random Forest.

To tune the hyperparameters and prevent overfitting we used GridSearch cross-validation, as GridSearch alone could produce a model that performs poorly on test data and highly on training data [9]. It was found that the optimal values of the hyperparameters are as follows:

- n_estimators: 100
- max_depth: 20

3.3 LSTM

An innovation of Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) network is capable of doing more than a typical RNN. Where RNNs face the problem of the inability to retain information over long sequences, the design of LSTM overcomes that issue as it uses a memory cell to store information from previous states [10]. As the nature of this project requires context from prior sequences when using neural networks, LSTM is a preferred choice over RNN.

3.3.1 Background

A vanilla LSTM model has 3 layers, the input layer, hidden layer and output layer. The LSTM model consists of 3 gates: forget gate, input gate and output gate. As information moves from one cell to another, the current cell (C_t) will receive information from the previous cell (C_{t-1}). The gates come together to form a chain of sigmoid and hyperbolic functions as opposed to just a sigmoid function in Vanilla-RNN models.

Forget Gate: The deciding factor if the information from C_{t-1} should be kept or forgotten. Controlled by a sigmoid (σ) function, it outputs a value $[0,1]$ where if the value is 0 it tells the function to forget (ignore) everything, while 1 to retain everything.

$$f_t = \sigma(W_f \cdot [H_{t-1}, x_t] + b_f)$$

Input Gate: Takes in the current input vector (x_t) and processes the information to decide which vector should be updated.

$$i_t = \sigma(W_i \cdot [H_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [H_{t-1}, x_t] + b_C)$$

Similar to the forget gate, the sigmoid function present in i_t serves to restrict values to an interval of $[0,1]$. The hyperbolic function outputs a value between $[-1,1]$ to determine what values should be subtracted or added towards the cell state respectively. The values of the forget gate and input gate come together to form the current cell state as seen in the following equation:

$$C_t = i_t \cdot \tilde{C}_t + f_t \cdot C_{t-1}$$

Output Gate: Output using filtered values of the current cell state C_t. These output values are the values that the following cell C_{t+1} uses.

$$o_t = \sigma(W_o \cdot [H_{t-1}, x_t] + b_o)$$
$$H_t = o_t \cdot \tanh(C_t)$$

3.3.2 Running the model

Transformed vectors obtained from the trained Word2Vec model are loaded into an embedding matrix layer of the LSTM model. The embedding layer as the name suggests provides the model with details of the input, namely the dimensions and vectorized values for each word trained, where these vectors will represent the words. Dropout and recurrent dropout layers have also been specified at 0.5 to combat overfitting as well as the lack of sufficient labelled data.

Since tweets have a character count limitation, an approximation of padding to length 50 was deemed sufficient as the max number of words in the dataset is observed to be less than 40. After the LSTM layer, the network will be activated using a sigmoid function as the model is attempting to do a binary classification. The model is then evaluated while training using a validation set, and after training, the model is evaluated over a test set by observing the accuracy through a classification report.

3.4 Transformers

Transformers are one of the most recent developments in the area of NLP [11]. Similarly to RNNs, they are designed to process sequential data, which renders them widely applicable in the area of text processing. However, contrary to RNNs, Transformers process the entire input at once which allows to identify global dependencies between data more efficiently. Furthermore, Transformers allow for significantly more parallelisation thus greatly reducing training time [11].

3.4.1 Background

Architecture of Transformers consists of two stacks of layers for the encoder and decoder, which are shown in the left and right sections of Figure 3-1, respectively. They both consist of 6 fully connected, identical layers, where each layer uses the mechanism of self-attention in its sub-layer. Encoder is used to process the input and generate encodings which represent the relevancy of different segments of the input to each other. On the other hand, decoder is used to decompose the encodings and produce the final output. Self-attention mechanism is used to scan each part of the input and evaluate the corresponding relevance of other parts. When producing the output sequence, the model is able to use the information from any of the previous states and weights them based on the previously evaluated relevance. Moreover, self-attention ensures that all positions in a sequence are connected with a constant number of operations, which allows for a better parallelisation compared to commonly used recurrent layers. Self-attention layers also have lower time complexity than recurrent layers if the sequence length is smaller than the representation dimensionality, which is usually the case. Finally, self-attention leads to models that can be interpreted more easily [11].

Apart from the self-attention sub-layers, encoder and decoder layers also use fully connected feed-forward networks, which apply linear transformations with a ReLU function, as in Equation 3.1.

$$\mathrm{FFN}(x) = \max(0,\, x \cdot W_1 + b_1) \cdot W_2 + b_2 \tag{3.1}$$

Furthermore, the model uses a mechanism called multi-headed attention in order to improve the performance of the self-attention layers. It allows the model to comprehend information from different representation subspaces at different positions in a sequence [11].

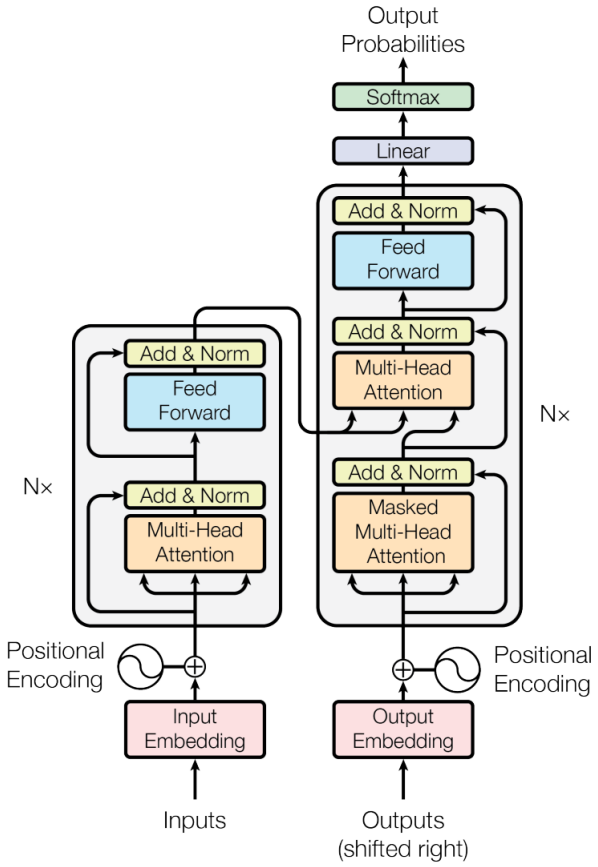


Figure 3-1: Architecture of Transformers [11]

The most efficient method of Transformers implementation is using a pre-trained model and fine-tuning it to perform a specific task. One of the most notable pre-trained models is BERT [12]. It was developed by Google in 2018 and commonly used in NLP applications since then. Since being widely adopted in the scientific community, BERT has been extensively analysed and modified, which resulted in many improved models being derived from it. One of such models is RoBERTa [13], which introduced several improvements to BERT. They primarily included training advancements, such as increased training time and batch size, longer training sequences and dynamically changing the masking pattern applied to the training data. As a result, RoBERTa is capable of performing better than other post-BERT methods [13]. For that reason, we chose to train a RoBERTa based model in order to achieve a state-of-the-art performance in sentiment analysis.

3.4.2 Model selection

The selected model was trained on a dataset of 124 million tweets in order to perform sentiment analysis. The training dataset has been based on tweets from January 2018 to December 2021 and thus the model is also capable of correctly understanding and classifying texts that relate to COVID-19. This ability may be crucial in order to correctly evaluate the market sentiment during potential future pandemic outbreaks.

3.4.3 Fine-tuning the model

The model was trained on various types of tweets. Therefore, it is necessary to fine-tune it on a dataset containing tweets about stock market in order to develop an understanding of specific financial jargon and the implications of particular words. For example, the baseline model may classify the word "bear" as neutral, but the fine-tuned model should classify it as negative since it might be associated with a bear market.

Furthermore, the baseline model classifies each piece of text as either positive, negative, or neutral. The training and test datasets contain tweets that are classified as either positive or negative, without the neutral class. Since the object of interest is binary classification, the output of the final model is converted by disregarding the probability of the neutral class and normalising the probabilities of positive and negative classes such that the entries add up to 1.

3.5 Models comparison

The following table presents test accuracy achieved by each model. Accuracy was measured on the same test set in order to obtain a meaningful comparison.

Table 3-1

Model	Test accuracy
Random forest	0.686
LSTM	0.745
roBERTa	0.827

Results

In order to gain more insight into the characteristics and dynamics of the sentiment scores, we computed and plotted them over a period of two months with the daily reevaluation strategy. This analysis was performed for some of the major stocks due to a better data availability. A resulting graph for Meta and Tesla can be seen in Figure 4-1.

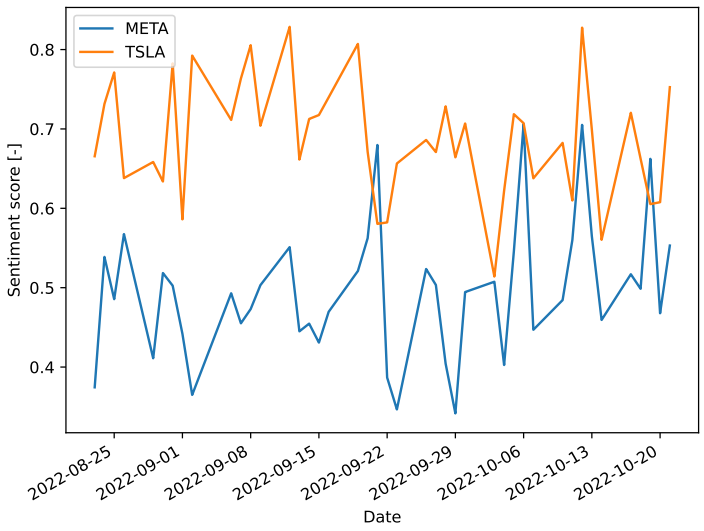


Figure 4-1: Sentiment scores of Meta and Tesla stocks over a span of two months

As it can be seen, the average value for Tesla is significantly higher than for Meta. Furthermore, similar dynamics between the two curves can be observed around the sudden spikes and falls, where they follow the same trajectory. This indicates that those changes are probably a market response to the general price trends, not necessarily to some event related to a particular company. Nevertheless, there are regions where the curves behave differently, which implies that the sentiment score can also serve as an indicator of the relative sentiment of different stocks. Furthermore, we plotted the sentiment score against the actual stock price. A resulting graph for Apple can be seen in Figure 4-2.

Again, it can be seen that both curves follow similar dynamics, with overlapping spikes and falls. This indicated that the sentiment score can be used as a price indicator, and potentially also to minimise the risk associated with price decline. Finally, we applied the two different approaches for portfolio allocation as discussed in subsection 2.3.2 and evaluated them through the method of back-testing. The result of this analysis is presented in Figure 4-3.

The third strategy called “Benchmark” serves as a baseline model for comparison with the strategies involving sentiment analysis. It involves creating an equally-weighted portfolio consisting of all S&P100 stocks and just waiting for a given amount of time. These three strategies will be evaluated and compared based on several metrics:

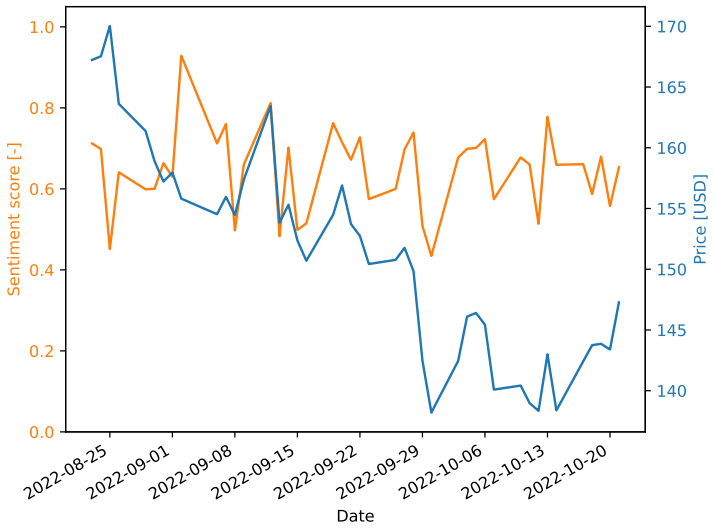


Figure 4-2: Sentiment score and price of Apple stock over a span of two months

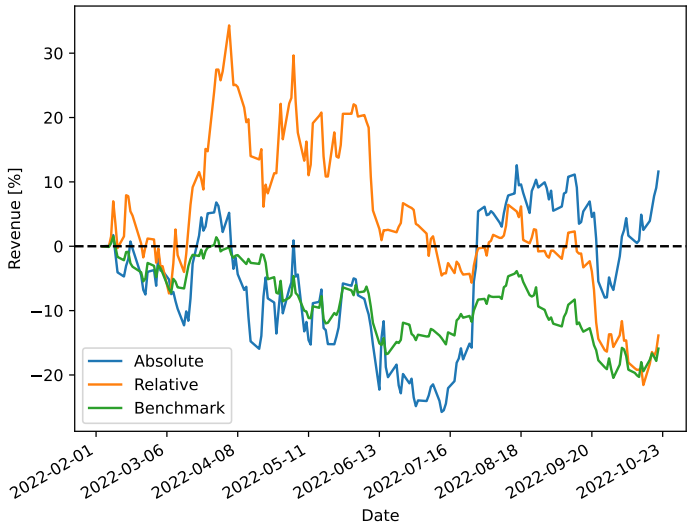


Figure 4-3: Cumulative returns of different trading strategies

- Cumulative return: total value that the strategy has earned over a given period of time. It is calculated using Equation 4.1.

$$\text{Cumulative return} = \frac{\text{Final value of investment} - \text{Initial value of investment}}{\text{Initial value of investment}} \tag{4.1}$$

- Annualised return: total value that the strategy would earn over a period of one year. It is calculated using Equation 4.2.

$$\text{Annualised return} = \left(1 + \text{Cumulative return}^{\frac{365}{\text{no. of days}}}\right) - 1 \tag{4.2}$$

- Annualised volatility: a measure of risk that is obtained by calculating standard deviation of the portfolio returns.

- Sharpe ratio: an indicator that compares the return of an investment to its risk. It is based on "risk-free" returns, which come from assets that are not subject to risk, such

as government bonds. For the sake of this comparison the U.S. 10 Year Treasury Note will be used with the value of 4.21% as of the 21st of October, 2022. Sharpe ratio is calculated using Equation 4.3.

$$\text{Sharpe ratio} = \frac{\text{Portfolio returns} - \text{Risk free returns}}{\text{Standard deviation of the portfolio returns}} \tag{4.3}$$

- Maximum drawdown: a measure of the maximal loss from peak to trough of a portfolio. It is calculated using Equation 4.4.

$$\text{Maximum drawdown} = \frac{\text{Price at trough} - \text{Price at peak}}{\text{Price at peak}} \tag{4.4}$$

Performance metrics of the three strategies are presented in Table 4-1. As can be seen, the absolute sentiment strategy achieves significantly better returns than the other two strategies. However, it also suffers from higher volatility and maximum drawdown, especially compared to the benchmark strategy. Nevertheless, the Sharpe ratio, which takes into account both returns and volatility, indicates that the absolute sentiment strategy actually performs better than the other two strategies.

Table 4-1: Performance metrics of different trading strategies

	Absolute	Relative	Benchmark
Cumulative return	11.61%	-13.87%	-15.90%
Annualised return	16.62%	-18.86%	-21.53%
Annualised volatility	49.87%	43.12%	22.96%
Sharpe ratio	0.55	-0.28	-0.96
Maximum drawdown	-34.06%	-41.61%	-21.66%

Discussion

Next, the cost of implementing the developed Transformers model to perform portfolio allocation is compared against the cost of just hiring a team of data scientists and analysts. As seen in diagram Table 5-1 below, on average, the total investment for a five-year total cost of ownership (TCO) ranges from approximately S$60,750 to S$94,500 [14]. This range is largely due to the two structures - Bare-Bones and MLOps Framework. The bare-bones structure lacks essential elements, resulting in performance degradation over time. The MLOps on the other hand, consist of not just developing a machine learning model, but also implementing it as a production system. This would increase automation and improve the quality of production models. The cost breakdown for both approaches is seen in Table 5-1.

On the other hand, the cost of hiring a full team of data scientists is significantly greater, with a rough estimation in the neighbourhood of S$1.05 million to S$2.31 million for the same five-year period. The three main roles in a data science team consist of a data scientist, data/ business analyst, and subject matter expert [15]. The pay range for those roles can be seen in Table 5-2.

Cumulatively, the total cost of hiring a team of data scientists far exceeds the total investment of implementing the developed Transformers model. Ergo, it would be optimal and more economical to develop and implement the developed model with the MLOps framework approach. Furthermore, given that the average annualised return of the developed model is 16.62% (see Table 4-1), it would be necessary to invest at least S$43,809 for a period of 5 years in order to start making profit. It has to be noted that the potential profits are not guaranteed and the model exhibits relatively high volatility.

Table 5-1: Operational Cost of ML solution

	Bare-Bones	MLOps Framework
Model Infrastructure	$9,000/yr	$8,000/yr
Data Support	$6,750/yr (labour)	$10,000 (labour) + $3,200/yr
Engineering/Deployment	$9,000 (labour)	$24,500 (labour) + $516/yr
Total Investment 5 year TCO	$60,750	$94,500

Table 5-2: Data Science Team without Transformer Model

	Pay Range	
Data Scientist	$9,000/month	$30,000/month
Subject Matter Expert	$3,500/month	$3,500/month
Data/Business Analyst	$5,000/month	$5,000/month
Total Investment 5 year TCO	S$1,050,000	S$2,310,000

Conclusion

With emotions as a factor driving the ever-unpredictable stock market, the objective of this project was to gauge the sentiments of the community and determine a viable trading strategy. To gauge those strategies, we used notable methods known in the machine learning industry – RF, LSTM, and Transformers. Word embeddings were generated from unsupervised tweets before applying them to the RF and LSTM algorithms, while Transformers have a more unique independent model. The model with the highest test accuracy was chosen to conduct back-testing on tweets and stock prices scraped in real-time. As the most recently developed of the three methods, the Transformers model was used for further analysis. Back-testing included 3 different investment strategies and analysed over 5 performance metrics to determine what would be the most optimal strategy on top of the sentiment analysis by the model. The sentiment score was applied to the respective strategical equations, weighted and evaluated together with a walk-forward optimisation of 10-folds. In conclusion, the highest returns for the model was noted to be the Absolute sentiment strategy, outperforming the Relative sentiment strategy and Benchmark strategy in terms of returns and Sharpe ratio. The annualised return of the final strategy was estimated to be 16.62% on average. Furthermore, the cost of running the model for 5 years in an MLOps framework was estimated to be S$94,500, which implies that it would be necessary to invest at least S$43,809 for a period of 5 years in order to start making profit.

However, while the numbers seem to determine this as a promising strategy, further research is required due to the constraints of data and time. The data obtained is rather biased due to the short time frame which does not consider the possibility of huge changes in society such as wars, disasters, and other inexhaustible reasons. Other methods could also be further optimised such as increasing the number of tweets to train Word2Vec, as published works have mentioned 300 dimensions as a peak parameter for the method.

References

[1] B. Jacobs and K. Lev, "The complexity of the stock market," 1989.

[2] E. F. Fama, "The behavior of stock-market prices," 1965.

[3] P. Koratamaddi, K. Wadhwani, M. Gupta, and S. G. Sanjeev, "Market sentiment-aware deep reinforcement learning approach for stock portfolio allocation," 2021.

[4] J. Smailović, M. Grčar, N. Lavrač, and M. Žnidaršič, "Stream-based active learning for sentiment analysis in the financial domain," 2014.

[5] D. Katayama and K. Tsuda, "A method of measurement of the impact of japanese news on stock market," 2018.

[6] D. Katayama and K. Tsuda, "A method of using news sentiment for stock investment strategy," 2020.

[7] M. Tomas, C. Kai, C. Greg, and D. Jeffrey, "Efficient estimation of word representations in vector space," 2013.

[8] P. Kevin and B. Pushpak, "Towards lower bounds on number of dimensions for word embeddings," 2017.

[9] S. Kale, "Grid search tuning of hyper parameters in random forest classifier," 2022.

[10] Y. Wang, Y. Liu, M. Wang, and R. Liu, "LSTM model optimization on stock price forecasting," pp. 173–177, 2018. DOI: 10.1109/DCABES.2018.00052.

[11] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," 2017.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018.

[13] L. Yinhan, O. Myle, G. Naman, *et al.*, "Roberta: A robustly optimized BERT pretraining approach," 2019.

[14] R. Coop, "What is the cost to deploy and maintain a machine learning model?," 2021.

[15] J. Saltz, "8 key data science team roles," 2022.

Appendix

A.1 Data processing

```python
import pandas as pd
import re
from nltk.corpus import words


def cleanText(text):
    text = re.sub(r'https?\S+', 'http', text)
    text = text.replace('user', '@user')
    text = re.sub(r'@\S+', '@user', text)
    text = re.sub(r'\d+', '@number', text)

    upperCase = re.findall(r'\b[A-Z]+(?:\s+[A-Z]+)*\b', text)
    for i in range(len(upperCase)):
        upperCaseSplit = upperCase[i].split()
        for j in range(len(upperCaseSplit)):
            if upperCaseSplit[j] in nasdaqSymbols or upperCaseSplit[j].lower() not in
                englishWords:
                text = text.replace(upperCaseSplit[j], '@stock')

    text = re.sub('[^A-Za-z0-9@]', ' ', text)
    text = re.sub(' +', ' ', text)
    text = text.lower()
    text = text.strip()
    return text


df = pd.read_csv("data/stock_tweets.csv")
nasdaq = pd.read_csv("data/nasdaq_stocks.csv")
nasdaqSymbols = nasdaq.iloc[:, 0].values
englishWords = set(words.words())

df['Text'] = df['Text'].apply(cleanText)

df = df.rename(columns={'Text': 'text', 'Sentiment': 'label'})
df.to_csv("data/cleaned_tweets.csv", index=False)

```

Listing A.1: Data cleaning

```python
from datasets import load_dataset
import pandas as pd
import matplotlib.pylab as plt
import numpy as np

dataset = load_dataset('csv', data_files="data/cleaned_tweets.csv", split='train')
dataset = dataset.train_test_split(test_size=0.2, shuffle=True, seed=1)

train_data = pd.DataFrame(dataset['train'])
test_data = pd.DataFrame(dataset['test'])

train_data.to_csv("data/train_data.csv", index=False)
test_data.to_csv("data/test_data.csv", index=False)
```

Listing A.2: Data splitting

A.2 Word2vec

```
import pandas as pd
from gensim.models import Word2Vec, KeyedVectors

df = pd.read_csv("data/cleaned_tweets.csv")
data = [row.split(' ') for row in df['text']]

model = Word2Vec(data, min_count=3, vector_size=100, workers=3, window=5, sg=1,
    epochs=100)
word_vectors = model.wv
word_vectors.save("data/vectors.kv")

reloaded_word_vectors = KeyedVectors.load("data/vectors.kv")
print(reloaded_word_vectors)
```

Listing A.3: Word2vec

A.3 Random Forest

```
from datasets import load_dataset, Dataset
#Relabel the sentiment value to [0,1] where 0 = neg and 1 = pos
train_data['label'] = train_data['label'].replace(-1,0)
test_data['label'] = test_data['label'].replace(-1,0)
#Transform dataset from pandas to dataset for consistency
train_df = Dataset.from_pandas(train_data)
test_df = Dataset.from_pandas(test_data)
trainval_df = train_df.train_test_split(test_size=0.2)
trainval_df, test_df
x_train = trainval_df['train']["text"]
x_val = trainval_df['test']['text']
x_test = test_df["text"]
y_train = trainval_df['train']["label"]
y_val = trainval_df['test']['text']
y_test = test_df["label"]

def make_feature_vec(words, model, num_features):
    """
    Average the word vectors for a set of words
    """
    feature_vec = np.zeros((num_features,),dtype="float32")  # pre-initialize (for
        speed)
    nwords = 0.
    index2word_set = set(model.wv.index_to_key)  # words known to the model

    for word in words:
        if word in index2word_set:
            nwords = nwords + 1.
            feature_vec = np.add(feature_vec,model.wv[word])

    feature_vec = np.divide(feature_vec, nwords)
    return feature_vec


def get_avg_feature_vecs(reviews, model, num_features):
    """
    Calculate average feature vectors for all tweets
    """
    counter = 0
```

```
    review_feature_vecs = np.zeros((len(reviews),num_features), dtype='float32')  # pre-initialize (for speed)

    for review in reviews:
        review_feature_vecs[counter] = make_feature_vec(review, model, num_features)
        counter = counter + 1
    return review_feature_vecs

# calculate average feature vectors for training and test sets
num_features = 100
clean_train_reviews = x_train
trainDataVecs = get_avg_feature_vecs(clean_train_reviews, model, num_features)

clean_test_reviews = x_test
testDataVecs = get_avg_feature_vecs(clean_test_reviews, model, num_features)
trainDataVecs.shape

# remove instances in test set that could not be represented as feature vectors
nan_indices = list({x for x,y in np.argwhere(np.isnan(testDataVecs))})
if len(nan_indices) > 0:
    print('Removing {:d} instances from test set.'.format(len(nan_indices)))
    testDataVecs = np.delete(testDataVecs, nan_indices, axis=0)
    x_test.drop(x_test.iloc[nan_indices, :].index, axis=0, inplace=True)
    assert testDataVecs.shape[0] == len(x_test)

# Fit a random forest to the training data before hyperparameter tuning
forest = RandomForestClassifier()
print("Fitting a random forest to labeled training data...")
forest = forest.fit(trainDataVecs, y_train)

#print classification report before hyperparameter tuning
print("Predicting labels for test data before hyperparameter tuning")
result = forest.predict(testDataVecs)
print(classification_report(y_test, result))
accuracy = round(accuracy_score(y_test, result),3)
print(accuracy)

#Now we'll do some hyperparameter tuning using GridSearchCV.
def print_results(results):
    print('BEST PARAMS: {}\n'.format(results.best_params_))
    means = results.cv_results_['mean_test_score']
    stds = results.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, results.cv_results_['params']):
        print('{} (+/-{}) for {}'.format(round(mean, 3), round(std * 2, 3), params))

from sklearn.model_selection import GridSearchCV
rf = RandomForestClassifier()
parameters = {
    'n_estimators': [5,50,100],
    'max_depth': [2,10,20,None]
}

cv = GridSearchCV(rf,parameters)
cv.fit(trainDataVecs,y_train)
print_results(cv)

#using the best parameter, we train a new random forest rf2
rf2 = RandomForestClassifier(n_estimators = 100, max_depth = 20)
rf2.fit(trainDataVecs, y_train)

#print classification report after hyperparameter tuning
print("Predicting labels for test data after hyperparameter tuning")
predictions = rf2.predict(testDataVecs)
```

```
print(classification_report(y_test, predictions))
accuracy1 = round(accuracy_score(y_test, predictions), 3)
print(accuracy1)
```

Listing A.4: Random Forest

A.4 LSTM

```
from gensim.models import Word2Vec, KeyedVectors
from datasets import load_dataset, Dataset
import pandas as pd
reloaded_word_vectors = KeyedVectors.load('vectors.kv')

dataset = load_dataset('csv', data_files="data/cleaned_tweets.csv", split='train')
dataset = dataset.train_test_split(test_size=0.2, shuffle=True, seed=1)
#Load training and test dataset from cleaned and split data
train_data = pd.read_csv('data/train_data.csv')
test_data = pd.read_csv('data/test_data.csv')
#Relabel the sentiment value to [0,1] where 0 = neg and 1 = pos
train_data['label'] = train_data['label'].replace(-1,0)
test_data['label'] = test_data['label'].replace(-1,0)
#Transform dataset from pandas to dataset for consistency
train_df = Dataset.from_pandas(train_data)
test_df = Dataset.from_pandas(test_data)
#Split into training and validation dataset
trainval_df = train_df.train_test_split(test_size=0.2)

#Specify hyperparameters
vocab_size = embedding_matrix[0]
mxlen = 50
nb_classes = 2

tokenizer = Tokenizer(num_words=vocab_size)
tokenizer.fit_on_texts(trainval_df['train']['text'])
sequences_train = tokenizer.texts_to_sequences(trainval_df['train']['text'])
sequences_test = tokenizer.texts_to_sequences(test_df['text'])
sequences_val = tokenizer.texts_to_sequences(trainval_df['test']['text'])

word_index = reloaded_word_vectors.key_to_index
for i in word_index: #to add 1 to the indexes given by Word2Vec as index 0 is
    reserved for padding
    if word_index['@stock'] < 2:
        word_index[i] += 1
    else:
        word_index['@stock'] = 1
        break

print('Found %s unique tokens.' % len(reloaded_word_vectors.key_to_index))
res = [(key,val) for key, val in word_index.items()]
print('Top few parts of token dict',res[:5])

#Pad vectors to same length
x_train = sequence.pad_sequences(sequences_train, maxlen=mxlen)
x_test = sequence.pad_sequences(sequences_test, maxlen=mxlen)
x_val = sequence.pad_sequences(sequences_val, maxlen=mxlen)

#Transform sentiment value into categorical classes
y_train = np_utils.to_categorical(trainval_df['train']['label'], nb_classes)
y_test = np_utils.to_categorical(test_df['label'], nb_classes)
y_val = np_utils.to_categorical(trainval_df['test']['label'], nb_classes)

```

```
batch_size = 32
nb_epoch = 20

embedding_layer = Embedding(reloaded_word_vectors.vectors.shape[0],
                            reloaded_word_vectors.vectors.shape[1],
                            weights=[reloaded_word_vectors.vectors],
                            trainable=False)

model = Sequential()
model.add(embedding_layer)
model.add(LSTM(128, recurrent_dropout=0.5, dropout=0.5))
model.add(Dense(nb_classes))
model.add(Activation('sigmoid'))
model.summary()

model.compile(optimizer='adam', loss='binary _crossentropy', metrics=['accuracy'])
model.save("LSTM Model")

rnn = model.fit(x_train, y_train, epochs = nb_epoch, batch_size=batch_size,
    shuffle=True, validation_data=(x_val, y_val))
score = model.evaluate(x_val, y_val)
print("Val Loss: %.2f%%" % (score[0]*100))
print("Val Accuracy: %.2f%%" % (score[1]*100))
score_2 = model.evaluate(x_test, y_test)
print("Test Loss: %.2f%%" % (score_2[0]*100))
print("Test Accuracy: %.2f%%" % (score_2[1]*100))

plt.figure(0)
plt.plot(rnn.history['accuracy'],'r')
plt.plot(rnn.history['val_accuracy'],'g')
plt.xticks(np.arange(0, nb_epoch+1, nb_epoch/5))
plt.rcParams['figure.figsize'] = (8, 6)
plt.xlabel("Num of Epochs")
plt.ylabel("Accuracy")
plt.title("Training vs Validation Accuracy LSTM l=50, epochs=20") # for max length =
    10 and 20 epochs
plt.legend(['train', 'validation'])

plt.figure(1)
plt.plot(rnn.history['loss'],'r')
plt.plot(rnn.history['val_loss'],'g')
plt.xticks(np.arange(0, nb_epoch+1, nb_epoch/5))
plt.rcParams['figure.figsize'] = (8, 6)
plt.xlabel("Num of Epochs")
plt.ylabel("Training vs Validation Loss LSTM l=50, epochs=20") # for max length = 10
    and 20 epochs
plt.legend(['train', 'validation'])
plt.show()

#Check data
y_pred = model.predict(x_test)
# Convert Y_Test into 1D array
yy_true = [np.argmax(i) for i in y_test]
yy_scores = [np.argmax(i) for i in y_pred]

test_df = test_data
test_df['pred'] = yy_scores
test_df

#Reporting Scores
from sklearn.metrics import classification_report, confusion_matrix
target_names = ['Neg', 'Positive']
```

```
print(classification_report(test_df['label'], test_df['pred'],
    target_names=target_names))
```

Listing A.5: LSTM

A.5 roBERTa

```
from datasets import load_dataset
from transformers import AutoTokenizer
from transformers import AutoModelForSequenceClassification
import numpy as np
import evaluate
from transformers import TrainingArguments, Trainer
import pandas as pd


def adjustLabels(label):
    return label + 1


def findMaxLen(dataset):
    maxLen = 0
    for sequence in dataset:
        if len(sequence.split()) > maxLen:
            maxLen = len(sequence.split())
    return maxLen


def tokenizeFunction(data):
    return tokenizer(data['text'], padding="max_length", truncation=True,
        max_length=maxLen)


def computeMetrics(eval_pred):
    logits, labels = eval_pred
    # setting up all logits corresponding to neutral to nan in order to filter them
        out
    logits[:, 1] = np.nan
    predictions = np.nanargmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)


# adjusting labels to 0 -> negative and 2 -> positive
df = pd.read_csv("data/cleaned_tweets.csv")
df['label'] = df['label'].apply(adjustLabels)
df.to_csv("data/tweets_roberta.csv", index=False)

dataset = load_dataset('csv', data_files="data/tweets_roberta.csv", split='train')
maxLen = findMaxLen(dataset['text'])

dataset = dataset.train_test_split(test_size=0.2, shuffle=True, seed=1)
tokenizer =
    AutoTokenizer.from_pretrained("cardiffnlp/twitter-roberta-base-sentiment-latest")
tokenized_datasets = dataset.map(tokenizeFunction, batched=True)

train_dataset = tokenized_datasets['train']
eval_dataset = tokenized_datasets['test']

model =
    AutoModelForSequenceClassification.from_pretrained("cardiffnlp/twitter-roberta-
                                                        base-sentiment-latest")
```

```
training_args = TrainingArguments(output_dir="training_checkpoints",
    evaluation_strategy="epoch")

metric = evaluate.load('accuracy')

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    compute_metrics=computeMetrics,
)

trainer.train()
trainer.save_model('tuned_model')
tokenizer.save_pretrained('tuned_model')
```

Listing A.6: roBERTa training

```
from transformers import AutoModelForSequenceClassification
from transformers import AutoTokenizer, AutoConfig
import numpy as np
from scipy.special import softmax
import re


def preprocess(text):
    text = re.sub(r'https?\S+', 'http', text)
    text = re.sub(r'@\S+', '@user', text)
    text = re.sub(r'\d+', '@number', text)
    text = re.sub(r'[$][A-Za-z]\S*', '@stock', text)
    text = re.sub('[^A-Za-z0-9@]', ' ', text)
    text = re.sub(' +', ' ', text)
    text = text.lower()
    text = text.strip()
    return text


def getScore(text):
    text = preprocess(text)
    encoded_input = tokenizer(text, return_tensors='pt')
    output = model(**encoded_input)
    scores = output[0][0].detach().numpy()
    scores = softmax(scores)
    scores[1] = 0
    scores = scores / np.sum(scores)
    return scores[2]


MODEL = f"tuned_model"
tokenizer = AutoTokenizer.from_pretrained(MODEL)
config = AutoConfig.from_pretrained(MODEL)
model = AutoModelForSequenceClassification.from_pretrained(MODEL)
```

Listing A.7: roBERTa sentiment score

A.6 Results

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

```
import re
import snscrape.modules.twitter as sntwitter
from joblib import Parallel, delayed
from roberta import getScore
import time


def checkSymbols(symbols):
    check = True
    for symbol in symbols:
        if symbols[0] != symbol:
            check = False
            break
    return check


def historicalSentiment(i):
    stopDate = dt.datetime.strptime(dates[i], '%Y-%m-%d') + dt.timedelta(days=int(1))
    stopDate = stopDate.strftime('%Y-%m-%d')
    startDate = dates[i]

    scores = np.zeros(2 * len(symbols))
    for j in range(len(symbols)):
        query = '$' + symbols[
            j] + ' lang:en since:' + startDate + ' until:' + stopDate + '
                -filter:links -filter:replies'
        symbolScores = np.array([])
        for k, tweet in enumerate(sntwitter.TwitterSearchScraper(query).get_items()):
            if len(symbolScores) >= 100:
                break
            if checkSymbols(re.findall(r'[$][A-Za-z]\S*', tweet.content)):
                symbolScores = np.append(symbolScores, getScore(tweet.content))

        scores[2 * j] = np.average(symbolScores) if len(symbolScores) > 0 else 0
        scores[2 * j + 1] = len(symbolScores)

    return scores


symbols = pd.read_csv("data/S&P100_symbols.csv")['Symbol']

columns = []
for i in range(len(symbols)):
    columns.append(symbols[i])
    columns.append(symbols[i] + ' TweetCount')

date = dt.datetime.strptime('2022-02-02', '%Y-%m-%d')
dates = []
for i in range(365):
    if date.weekday() < 5:
        dates.append(date.strftime('%Y-%m-%d'))
    date = date - dt.timedelta(days=int(1))

start = time.time()

output = pd.DataFrame(columns=columns, index=dates)
results = np.array(Parallel(n_jobs=2)(delayed(historicalSentiment)(i) for i in
    range(2)))

output = pd.DataFrame(results, columns=columns, index=dates[:results.shape[0]])
output.index.name = 'Date'
output.to_csv("data/historical_sentiment.csv")

```

```
end = time.time()
print('Time elapsed:', round((end - start) / 60, 2))
```

Listing A.8: Historical sentiment

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import datetime as dt


def benchmark(indexDateStart, indexDateStop):
    stockWallet = np.zeros(prices.shape)
    wealth = np.array([1])

    for i in range(indexDateStop - indexDateStart):
        date = dates[indexDateStart + i]
        if date not in sentiment.index:
            break

        if i > 0:
            wealth = np.append(wealth, np.sum(stockWallet[i - 1] *
                prices.loc[date].to_numpy()))

        for j in range(stockWallet.shape[1]):
            stockWallet[i, j] = wealth[i] / stockWallet.shape[1] / prices.loc[date,
                prices.columns[j]]

    return wealth


def backtestAbsolute(numStocks, minTweetCount, indexDateStart, indexDateStop,
    startWealth):
    stockWallet = np.zeros(prices.shape)
    wealth = np.array([startWealth])

    for i in range(indexDateStop - indexDateStart):
        date = dates[indexDateStart + i]
        if date not in sentiment.index:
            break

        if i > 0:
            wealth = np.append(wealth, np.sum(stockWallet[i - 1] *
                prices.loc[date].to_numpy()))

        tweetCount = sentiment.loc[date].to_numpy()[1::2]
        sentimentScores = sentiment.loc[date].to_numpy()[::2]
        sentimentScores = np.where(tweetCount > minTweetCount, sentimentScores, 0)

        if np.count_nonzero(sentimentScores) < numStocks:
            numStocks = np.count_nonzero(sentimentScores)

        sortedTickers = np.argsort(-sentimentScores)

        for j in range(stockWallet.shape[1]):
            if np.where(sortedTickers == j)[0][0] < numStocks:
                stockWallet[i, j] = wealth[i] / numStocks / prices.loc[date,
                    prices.columns[j]]

    if indexDateStop < len(dates):
        finalWealth = np.sum(stockWallet[indexDateStop - indexDateStart - 1] *
            prices.loc[dates[indexDateStop]].to_numpy())
```

```
    else:
        finalWealth = 0
    return wealth, finalWealth


def backtestRelative(numStocks, minTweetCount, indexDateStart, indexDateStop, startWealth):
    stockWallet = np.zeros(prices.shape)
    wealth = np.array([startWealth, startWealth])

    for i in range(1, indexDateStop - indexDateStart):
        datePrevious = dates[indexDateStart + i - 1]
        date = dates[indexDateStart + i]
        if date not in sentiment.index:
            break

        if i > 1:
            wealth = np.append(wealth, np.sum(stockWallet[i - 1] *
                prices.loc[date].to_numpy()))

        if i > 0:
            tweetCountPrevious = sentiment.loc[datePrevious].to_numpy()[1::2]
            tweetCount = sentiment.loc[date].to_numpy()[1::2]
            sentimentScoresPrevious = sentiment.loc[datePrevious].to_numpy()[::2]
            sentimentScores = sentiment.loc[date].to_numpy()[::2]

            sentimentScores = np.divide(sentimentScores - sentimentScoresPrevious,
                                        sentimentScoresPrevious,
                                        out=np.zeros_like(sentimentScores -
                                            sentimentScoresPrevious),
                                        where=sentimentScoresPrevious != 0)
            sentimentScores = np.where(np.logical_and(tweetCountPrevious >
                minTweetCount, tweetCount > minTweetCount),
                                       sentimentScores, 0)

            if np.count_nonzero(sentimentScores) < numStocks:
                numStocks = np.count_nonzero(sentimentScores)

            sortedTickers = np.argsort(-sentimentScores)

            for j in range(stockWallet.shape[1]):
                if np.where(sortedTickers == j)[0][0] < numStocks:
                    stockWallet[i, j] = wealth[i] / numStocks / prices.loc[date,
                        prices.columns[j]]

    if indexDateStop < len(dates):
        finalWealth = np.sum(stockWallet[indexDateStop - indexDateStart - 1] *
            prices.loc[dates[indexDateStop]].to_numpy())
    else:
        finalWealth = 0
    return wealth, finalWealth


def optimise(numStockRange, minTweetCountRange, indexDateStart, indexDateStop, method):
    xdata = np.zeros(numStockRange * minTweetCountRange)
    ydata = np.zeros(numStockRange * minTweetCountRange)
    zdata = np.zeros(numStockRange * minTweetCountRange)

    bestNumStock = 0
    bestMinTweetCount = 0
    bestWealth = 0

```

```
    for i in range(1, numStockRange + 1):
        for j in range(1, minTweetCountRange + 1):
            wealth = np.average(method(i, j, indexDateStart, indexDateStop, 1)[0])

            xdata[(i - 1) * numStockRange + j - 1] = j
            ydata[(i - 1) * numStockRange + j - 1] = i
            zdata[(i - 1) * numStockRange + j - 1] = wealth

            if wealth > bestWealth:
                bestNumStock = i
                bestMinTweetCount = j
                bestWealth = wealth

    # fig = plt.figure()
    # ax = fig.add_subplot(projection='3d')
    # ax.scatter(xdata, ydata, zdata - 1)
    # ax.set_xlabel('Minimal number of tweets')
    # ax.set_ylabel('Number of traded stocks')
    # ax.set_zlabel('Revenue [%]')
    # fig.tight_layout()
    # plt.show()

    return bestNumStock, bestMinTweetCount, bestWealth


sentiment = pd.read_csv("data/historical_sentiment.csv", index_col='Date')
prices = pd.read_csv("data/historical_prices.csv", index_col='Date')
sentiment = sentiment.iloc[::-1]
prices = prices.iloc[::-1]

dates = list(prices.index)

split = 0.8
folds = 10
lenInSample = round(len(dates) * split / folds / (1 - split + split / folds))
lenOutSample = round(lenInSample * (1 - split) / split)

wealthAbsolute = np.array([])
wealthRelative = np.array([])
finalWealthAbsolute = 1
finalWealthRelative = 1
for i in range(folds):
    print("Progress: " + str(round(i / folds * 100, 2)) + "%")

    numStockAbsolute, minTweetCountAbsolute = optimise(20,
                                                       60,
                                                       lenOutSample * i,
                                                       lenOutSample * i + lenInSample,
                                                       backtestAbsolute)[:2]
    numStockRelative, minTweetCountRelative = optimise(20,
                                                       60,
                                                       lenOutSample * i,
                                                       lenOutSample * i + lenInSample,
                                                       backtestRelative)[:2]

    appendedWealthAbsolute, finalWealthAbsolute = backtestAbsolute(numStockAbsolute,
                                                                   minTweetCountAbsolute,
                                                                   lenOutSample * i +
                                                                       lenInSample,
                                                                   lenOutSample * i +
                                                                       lenInSample +
                                                                       lenOutSample,
                                                                   finalWealthAbsolute)
```

```
169
170     appendedWealthRelative, finalWealthRelative = backtestRelative(numStockRelative,
171                                                                    minTweetCountRelative,
172                                                                    lenOutSample * i +
                                                                           lenInSample,
173                                                                    lenOutSample * i +
                                                                           lenInSample +
                                                                           lenOutSample,
174                                                                    finalWealthRelative)
175
176     wealthAbsolute = np.hstack((wealthAbsolute, appendedWealthAbsolute))
177     wealthRelative = np.hstack((wealthRelative, appendedWealthRelative))
178
179 plotDates = list(prices.index)[lenInSample:]
180 x = [dt.datetime.strptime(d, '%Y-%m-%d').date() for d in plotDates]
181 wealthBenchmark = benchmark(lenInSample, len(dates))
182
183 dailyReturnsAbsolute = np.append([0], np.diff(wealthAbsolute) / wealthAbsolute[:-1])
184 dailyReturnsRelative = np.append([0], np.diff(wealthRelative) / wealthRelative[:-1])
185 dailyReturnsBenchmark = np.append([0], np.diff(wealthBenchmark) /
        wealthBenchmark[:-1])
186
187 print("Cumulative return absolute [%]:", (wealthAbsolute[-1] - 1) * 100)
188 print("Cumulative return relative [%]:", (wealthRelative[-1] - 1) * 100)
189 print("Cumulative return benchmark [%]:", (wealthBenchmark[-1] - 1) * 100, '\n')
190
191 print("Annualised return absolute [%]:", (wealthAbsolute[-1] ** (252 / len(x)) - 1) *
        100)
192 print("Annualised return relative [%]:", (wealthRelative[-1] ** (252 / len(x)) - 1) *
        100)
193 print("Annualised return benchmark [%]:", (wealthBenchmark[-1] ** (252 / len(x)) - 1)
        * 100, '\n')
194
195 print("Annualised volatility absolute [%]:", np.std(dailyReturnsAbsolute) *
        np.sqrt(252) * 100)
196 print("Annualised volatility relative [%]:", np.std(dailyReturnsRelative) *
        np.sqrt(252) * 100)
197 print("Annualised volatility benchmark [%]:", np.std(dailyReturnsBenchmark) *
        np.sqrt(252) * 100, '\n')
198
199 riskFreeRate = 0.0421 / 10 / 252
200 print("Sharpe ratio absolute:",
201       np.sqrt(252) * (np.average(dailyReturnsAbsolute) - riskFreeRate) /
              np.std(dailyReturnsAbsolute))
202 print("Sharpe ratio relative:",
203       np.sqrt(252) * (np.average(dailyReturnsRelative) - riskFreeRate) /
              np.std(dailyReturnsRelative))
204 print("Sharpe ratio benchmark:",
205       np.sqrt(252) * (np.average(dailyReturnsBenchmark) - riskFreeRate) /
              np.std(dailyReturnsBenchmark), '\n')
206
207 print("Maximum drowdown absolute [%]:",
208       (np.min(wealthAbsolute) - np.max(wealthAbsolute)) / np.max(wealthAbsolute) *
              100)
209 print("Maximum drowdown relative [%]:",
210       (np.min(wealthRelative) - np.max(wealthRelative)) / np.max(wealthRelative) *
              100)
211 print("Maximum drowdown benchmark [%]:",
212       (np.min(wealthBenchmark) - np.max(wealthBenchmark)) / np.max(wealthBenchmark) *
              100)
213
214 plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
```

```
plt.gca().xaxis.set_major_locator(mdates.DayLocator(interval=round((len(dates) -
    lenOutSample) / 7)))

plt.plot(x, (wealthAbsolute - 1) * 100, label="Absolute")
plt.plot(x, (wealthRelative - 1) * 100, label="Relative")
plt.plot(x, (benchmark(lenInSample, len(dates)) - 1) * 100, label="Benchmark")
plt.axhline(y=0, color='black', linestyle='--')

# for i in range(folds):
#     plt.axvline(x=x[lenOutSample * i], color='b')

plt.legend(loc="lower left")
plt.xlabel('Date')
plt.ylabel('Revenue [%]')

plt.tight_layout()
plt.gcf().autofmt_xdate()
plt.savefig('plots/revenue.pdf', bbox_inches='tight')
plt.show()
```

Listing A.9: Backtesting