MASTER'S THESIS 2020

Forecasting Financial Indices from Financial News

Gustaf Backman

ISSN 1650-2884
LU-CS-EX: 2020-33

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-33

Forecasting Financial Indices from Financial News

Gustaf Backman

Forecasting Financial Indices from Financial News

Gustaf Backman
ine14gba@student.lu.se

June 29, 2020

Master's thesis work carried out at Kidbrooke Advisory AB.

Supervisors: Pierre Nugues, pierre.nugues@cs.lth.se
Edvard Sjögren, edvard.sjogren@kidbrooke.com

Examiner: Marcus Klang, marcus.klang@cs.lth.se

Abstract

In the last few years the capability of natural language processing has increased greatly. The implications this methodical leap has for different areas is yet to be explored. In this thesis, I have evaluated the predictive power of financial news headlines on the movement of financial indices. More specifically, I tried to determine if I could improve a traditional time series model by adding news data. In my experiments, I used text representations of different complexities, from TF-IDF to recent transformer models, such as BERT and Sentence-BERT and, as targets, the S&P 500 and US treasury rates with one and three years maturity.

My findings suggest that no model can predict the movement of treasury rates based on news data, but the classification performance for the S&P 500 is well over a random baseline. The best model using the TF-IDF word representation and a random forest classifier could reach an accuracy of 59.1%.

Furthermore, adding news data can predict better than a random baseline whether a time series forecast is too high or too low. This applies to all indices. I could reach an accuracy of 64.1 % with SBERT feature extraction and logistic regression.

Keywords: NLP, fintech, deep learning, time series analysis, transformers.

Acknowledgements

The writing of this thesis has been a challenging and entertaining process, where I enjoyed the many aspects of the task.

I would like to extend my gratitude to my academic supervisor Pierre Nugues for his continuous support during the project and his advice on academic writing.

The same goes for my external supervisor Edvard Sjögren at Kidbrooke Advisory who has been a valuable asset for discussing problem formulations and methodological questions throughout this spring.

Finally, thanks to Johan Rosén for clear insights and motivational company for the duration of this thesis.

Contents

Chapter 1
Problem Description

Predicting the movement of a financial time series is generally done using traditional statistical methods based on the available historical data. There might, however, be information available not included in the historical data which has some predictive quality of the future development of an asset value.

Consider significant political or environmental events which have an effect on an asset in the long run. For instance news regarding trade relations between USA and China, the progress of Brexit, or an emerging war. This affect the value of an asset.

In order to deal with such uncertainties, analysts today have to alter the prediction from a traditional time series model with a more manual analysis of the state of the world through news data. This is a time consuming and subjective task that perhaps could be ameliorated by natural language processing of news.

1.1 Background

The last decade has seen a significant growth in the amount of published papers in the field of natural language processing related to finance (Xing et al., 2018). The use of text processing has repeatedly proven successful in tasks related to financial forecasting, for instance in Li et al. (2014), Heston and Sinha (2017), and Othan et al. (2019). The previous work in the field uses a diverse set of approaches, from crude methods of counting positive and negative words in articles to training deep neural networks on large corpora to produce meaningful vector representations of words (Arora et al., 2019).

1.2 Motivation

The field of natural language processing seems to be in an interesting phase with rather recent developments such as word2vec (Mikolov et al., 2013), Bidirectional Encoder Representations

from Transformers (BERT) (Devlin et al., 2018) and A lite BERT – ALBERT (Lan et al., 2019). As NLP is applicable to a wide variety of tasks, the performance and impact of these new models have yet to be explored in several areas.

Another positive aspect is that the source code with examples of usage of these novel methods are more often than not publicly available through GitHub. Such the models might be complex and require large computational power for training. Nonetheless, there are often pre-trained parameters available which can then be used either as-is or with further tuning for the task it will be used for.

An additional financial motivation for the thesis is the exploration of how news affects the value of an asset and how this complies with the *efficient market hypothesis* (EMH) as proposed by Malkiel and Fama (1970). The idea that the market adapts to new publicly available information instantly does not comply with the suggestion that publicly available news has any predictive power on the future value of an asset. Previous works by Xing et al. (2018) and Arora et al. (2019) have suggested that the case is indeed that public text data has some predictive power, so a motivation for this thesis is to explore the reproducibility of this effect.

1.3 Objective

The thesis is mainly concerned with evaluating whether a traditional model for time series prediction can be improved by adding additional input in the form of financial news titles. I framed this problem as two classification tasks. The first task is to predict whether an index has increased or decreased given a set of news titles. The second task is to predict if a traditional time series model forecasts higher or lower than the real value.

First, I will explore a few different approaches on word representations as well as the basic elements of traditional time series modelling. I also describe the theory concerning deep learning and machine learning in general, as well as some basic performance metrics. Finally, I will implement suitable models and evaluate the results.

The main research question is if financial news can be used to predict the movement of financial indices. More specifically, if adding financial news to a traditional time series model improves the performance.

1.4 Scope

The scope of the project is limited in a few ways. I analyzed price data from three financial indices, two US treasury rates and one US stock market index.

Text data is gathered from Reuters as financial headlines between October 2006 and November 2013. The content of the articles are available as well, but as suggested by Ding et al. (2014), using only news titles gives a higher performance.

Finally, limited access regarding powerful computing resources and time implies the models are unlikely to compete with similar state-of-the-art models trained on GPU/TPU's for several days.

Chapter 2
Theoretical Background

In this section, I will cover the underlying theory of the models and concepts I will use to construct and evaluate models. Fields with more recent advances are covered more thoroughly – such as transformers – whereas for instance traditional time series analysis is explained with less depth.

2.1 Time Series Analysis

Traditional time series analysis is concerned with finding statistical information of observations distributed in time. The purpose can be to get a better understanding of the underlying process, or to make predictions on future realisations of the process. Such processes are typically present in the fields of finance, signal processing, weather forecasting, control engineering etc. Several methods for modelling processes have been developed, where some of the oldest and most useful methods are the *autoregressive* model (AR) and the *moving average* model (MA). These two concepts can be combined to benefit from both models into an ARMA-model.

2.1.1 Autoregressive (AR) Model

An auto-regressive model of order p is a linear combination of the p previous terms plus some noise. An AR(p)-process is commonly characterized by its generating polynomial $A(z) = a_0 + a_1 z + \cdots + a_p z^p$, where $a_i \in \mathbb{R} \; \forall \; i$ and $a_0 = 1$. Furthermore, let e_t be uncorrelated white noise with variance σ^2 in discrete time defined as,

$$\mathrm{E}[e_t] = 0 \tag{2.1}$$

$$C[e_s, e_t] = \begin{cases} \sigma^2 \text{ if } s = t \\ 0 \text{ else} \end{cases} \tag{2.2}$$

Supposing $A(z)$ is a stable polynomial of degree p and e_t is as defined above, the stationary sequence X_t is called an AR(p)-process with generating polynomial $A(z)$.

$$X_t + a_1 X_{t-1} + \cdots + a_p X_{t-p} = e_t \tag{2.3}$$

The process is stable if the roots of the characteristic equation $z^p A(z^{-1}) = 0$ are all inside the unit circle. The values e_t are the called the innovations to the process and the coefficients of the $A(z)$-polynomial are tuneable parameters (Lindgren, 2014).

There are several techniques for estimating the coefficients of the $A(z)$-polynomial. One technique is to transform the problem onto regression form and view the $A(z)$-coefficients as regression coefficients. The most recent value X_t is set as the response variable y_t and the previous samples $(-X_{t-1}, \ldots, -X_{t-p})$ are the explanatory variables $\mathbf{x}_t$. Letting the coefficients $(a_0, a_1, ..., a_p)^T = \mathbf{A}$, Eq. 2.3 can be rewritten on a vector form, recognized from regression.

$$y_t = \mathbf{x}_t \mathbf{A} + e_t \tag{2.4}$$

The elements of $\mathbf{A}$ can then be estimated as the least squares estimate over n samples, i.e. the values of $\mathbf{A}$ that minimize 2.5.

$$L(\mathbf{A}) = \sum_{t=p+1}^{n} (y_t - \mathbf{x_t A})^2 \tag{2.5}$$

Or more conveniently, in matrix format.

$$\mathbf{Y} = \begin{pmatrix} y_{p+1} \\ y_{p+2} \\ \vdots \\ y_n \end{pmatrix}, \quad \mathbf{E} = \begin{pmatrix} e_{p+1} \\ e_{p+2} \\ \vdots \\ e_t \end{pmatrix}, \quad \mathbf{X} = \begin{pmatrix} \mathbf{x}_{p+1} \\ \mathbf{x}_{p+2} \\ \vdots \\ \mathbf{x}_t \end{pmatrix} \tag{2.6}$$

$$L(\mathbf{A}) = (\mathbf{Y} - \mathbf{XA})^T (\mathbf{Y} - \mathbf{XA}) \tag{2.7}$$

Differentiating 2.7 with respect to $\mathbf{A}$ and setting equal to zero yields the least squares estimate.

$$\frac{\partial L}{\partial \mathbf{A}} = -2\mathbf{X}^T\mathbf{Y} + 2\mathbf{X}^T\mathbf{XA} \tag{2.8}$$

$$\frac{\partial L}{\partial \mathbf{A}} = 0 \implies \hat{\mathbf{A}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \tag{2.9}$$

The estimated innovation variance $\hat{\sigma^2}$ is simply the sample variance of the innovations.

$$\hat{\sigma^2} = \frac{1}{n-p} \sum_{t=p+1}^{n} e_t^2 = \frac{L(\hat{\mathbf{A}})}{n-p} \tag{2.10}$$

The method above yields an estimate for the parameters of an AR(p)-process. The order p of the process still has to be determined using appropriate goodness-of-fit criteria such as Akaike information criterion (Lindgren, 2014).

2.1.2 Moving Average (MA) Model

Another popular time series model is the moving average (MA) model. This model is defined similarly to the AR-model by its generating polynomial, here called $C(z)$.

$$C(z) = c_0 + c_1 z + \cdots + c_q z^q \tag{2.11}$$

A MA(q)-process is a linear combination of the q previous white noise terms as defined in Eqs. 2.1 and 2.2, plus one new innovation term:

$$X_t = e_t + c_1 e_{t-1} + \cdots + c_q e_{t-q} \tag{2.12}$$

A common adjustment to the model is to set $c_0 = 1$ and adjust the other coefficients and the innovation variance accordingly.

An important distinction between the AR(p)-process and the MA(q)-process is that the covariance function for the MA(q)-process is zero for time lags τ larger than the order q of the process. In other words, the value X_{t+q+1} is independent of the value X_t in an MA(q)-process (Lindgren, 2014).

2.1.3 Autoregressive Moving Average (ARMA) Model

The AR-process and MA-process are commonly combined into an ARMA-process. Letting the noise on the right hand side of Eq. 2.3 be an MA(q)-process, the expression of an ARMA(p,q)-process is achieved.

$$X_t + a_1 X_{t-1} + \cdots + a_p X_{t-p} = e_t + c_1 e_{t-1} + \cdots + c_q e_{t-q} \tag{2.13}$$

There are various methods for estimating the coefficients of the polynomials $A(z)$ and $C(z)$. One is a regression approach similar to the method described in Sect. 2.1.1, where regression samples are constructed from the time series and the coefficients are estimated in a least squares sense. Another common estimation procedure is to use maximum likelihood estimation of the parameters. Assuming the noise follows some distribution – most commonly Gaussian – a distribution can be calculated for the model coefficients. The likelihood of obtaining a given set of a parameters can then be calculated and optimized (Hamilton, 1994).

2.2 Representation of Language

Language has enabled people to exchange information in an efficient manner for thousands of years, both through speech and text. The complexity and nuances that makes a language so fitting for transferring information between humans is also what makes it so difficult to represent in numbers. A few of the difficulties when interpreting text language are listed below.

Homonyms. Words that have the same spelling but different meaning. Consider for instance the word *bull*, which might refer to the animal or an investor who believes in a rising market. The same letters, but with vastly different interpretations depending on the context.

Negations. The sentences *God will help you* and *No God will help you* contain almost the same words, but have completely opposite meanings.

Sarcasm/irony. This can be hard enough to detect for humans. The phrase *That's just what I needed today!* might actually mean what it literally says, or just the opposite.

Methods used for representing words deal with these difficulties in different ways or not at all. For some shallow, more simple NLP tasks, a model might perform well without understanding homonyms or that two words are closely related. For more complex tasks such as sequence-to-sequence translation of language, a deeper understanding is naturally needed.

2.2.1 One-hot Encoding

The most intuitive way to turn words into numerical vectors is probably one-hot encoding. Thi is simply done by giving all unique words in a text an index and then letting the index represent the word. Consider a training example x_i as the sentence below:

A gorilla visited Manilla.

The first processing needed is to divide the sentence into smaller units – *tokens*. A tokenized version of the sentence above would be:

```
['A', 'gorilla', 'visited', 'Manilla', '.']
```

Some of these tokens carry information about the beginning or end of the sentence, which is obviously important. However, not all capital letters imply the start of a sequence and not all punctuation indicates the end of a sequence, e.g. *Hello Mr. Gorilla!*. There are quite a few special cases of this sort, and there are convenient functions in Python that deal with the problems of tokenization, such as *Tokenizer* from Keras (Chollet et al., 2020). The tokenizer from Keras splits the text into sequences, removes the punctuation and transforms all characters to lower case by default.

When the sentence is tokenized, it is also common to remove the most frequent words, since these probably don't give a lot of information about the difference between sentences. These are called *stop words* and typically include common words such as *a, the, but*, etc.

After the tokenization, each unique word is given an index.

$$\begin{aligned} \textit{gorilla}: &\quad 1 \\ \textit{visited}: &\quad 2 \\ \textit{manilla}: &\quad 3 \end{aligned}$$

Each word of the sentence is then transformed to a one-hot vector where all elements are zero except for element i. A sequence of words can then be represented as a sequence of vectors.

$$
\begin{aligned}
\textit{gorilla} &: \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \\
\textit{visited} &: \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \\
\textit{manilla} &: \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

2.2.2 Bag-of-words & TF-IDF

An initial *bag-of-words* (BOW) approach to represent a sentence as a vector is to simply keep track of whether a word is included in the sentence or not. If the word is included, the element on the corresponding index has value 1, otherwise 0. Using the same indices as previously and removing stop words, *A gorilla visited Manilla* would then be represented by x_i as,

$$
x_i = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}
$$

Note that even though the order of words is the same in the vector as in the original sentence, this is not necessarily the case. The less interpretable sentence *Manilla visited a gorilla* would have the same vector representation as x_i. Hence, BOW does not take order of words in a sentence into account.

There are variations of BOW that have larger representing power, such as including the count of words in the sentence rather than if it exists or not. A problem with this approach is that words that are more frequent in sentences get a higher value than words that are not as frequent, even if less frequent words might be more interesting for the context. A variation that deals with this limitation is the TF-IDF representation.

Term Frequency-Inverse Document Frequency (TF-IDF) is a widely used technique for normalizing text data. It uses the same underlying principles as BOW but with a weight normalization. As the name suggests, the value for a certain word is increased for its frequency in a sequence but decreased for the frequency in the full corpus. So, a word which has a low frequency in a full corpus is considered more important than a word with high frequency. A sequence can be a sentence, a document or some other subset of the full corpus. The entry for a word with index j in a sample vector x_i is then calculated as the product of the term frequency weight and the inverse document frequency weight.

$$
x_{ij} = f_s(t, f) \cdot f_d(t, F)
$$

For a term t with frequency f in sequence i and frequency F in the whole corpus. The function f_s is some function increasing with the number of words j in the sequence and f_d is decreasing with the number of words in the full corpus. Examples of these functions can be as below.

$$
\begin{aligned}
f_s &= \frac{|\{j \in (1, \ldots, L_i) : s_{ij} = t\}|}{L_i} \\
f_d &= \log \frac{N}{n_t}
\end{aligned}
$$

Where L_i is the length of the sequence i, N is the number of sequences in the corpus and n_t is the number of sequences in which the term t occurs at least once (Manning et al., 2008).

While this remedies some of the shortcomings of BOW, there are still some aspects where it falls short:

Firstly, the size of the vectors grows with the number of unique words in the corpus, which becomes computationally infeasible for larger texts.

Secondly, the order of the words is not accounted for in BOW. When used in an application for interpreting financial news, the representation must be able to distinguish between *Google placed a bid on Amazon* and *Amazon placed a bid on Google*. BOW and TF-IDF however simply register *if* a word has occured.

Finally, neither BOW nor TF-IDF does really capture any essence of the language. There is no way for the mode to capture the similarity between words such as *awesome* and *amazing*. This is related to the large dimensionality of the vectors representing the words, since each word has a unique dimension in the vector. This implies all dimensions are orthogonal, therefore there is no usable algebraic measure of similarity.

Most of the problems above are addressed by the concept of word embeddings in the next section.

2.2.3 Word Embeddings

As opposed to the sparse representation of one-hot encoding, *word embeddings* are dense, continuous vector representations of words. The dimension of a one-hot encoded vector corresponds to the size of the vocabulary (generally 20,000 or greater), whereas word embedding vectors typically have 100 to 1000 dimensions (Chollet, 2017).

Mikolov et al. created a major breakthrough on the topic of word embeddings in 2013. In their paper, they introduced efficient methods for training embedding vectors, popularized as the *word2vec* model. *Continuous bag-of-words* and *continuous skip-gram* are two model architectures for training the d-dimensional vector representations of words. Both methods share the notion that the meaning of a word is determined by the words it is commonly used together with. The representations are learned by constructing a language modeling task which is solved by a neural network. The task is generally to predict neighboring words of a given word. The weights into the hidden layer of the trained neural network are the embeddings for a word.

Continuous bag-of-words are trained by predicting the missing word in a sequence of words of a given window size. The order of the words is not taken into consideration other than for deciding which words to include in one sequence. For instance, the phrase *A gorilla visited Manilla* with window size one gives the following training samples.

$$A\ gorilla\ visited \implies x = (a, visited)\ , y = gorilla$$

$$gorilla\ visited\ Manilla \implies x = (gorilla, Manilla)\ , y = visited$$

Continuous skip-grams also use the fact that words that often occur together have some sense of similarity, but is in a way the inverse of continuous bag-of-words. Rather than predicting the missing word, the objective of the model is to predict the surrounding words. To construct training examples from the same phrase as above with a 1-skip-gram, the following samples are generated.

$$\textit{A gorilla visited} \implies x, y_1, y_2 = \textit{gorilla, a, visited}$$

$$\textit{gorilla visited Manilla} \implies x, y_1, y_2 = \textit{visited, gorilla, Manilla}$$

The known words above are *gorilla* and *visited* respectively.

According to the authors[1], the continuous bag-of-words model is faster for training but the continuous skip-gram model is better for infrequent words.

The word embeddings generated by these methods do carry some information about the semantic relationship of words. As mentioned in the previous section, a desirable function of word representation is to determine whether a word is close to another word in a semantic meaning. This is elegantly represented in word embeddings as the cosine similarity between word vectors. Consider for instance the word *Sweden*. The closest word vectors in the word2vec-vocabulary with respect to cosine similarity are displayed in Table 2.1.

Word	Cosine similarity
Finland	0.8085
Norway	0.7706
Denmark	0.7674
Swedish	0.7404
Swedes	0.7133

Table 2.1: Word vectors with the highest cosine similarity to *Sweden*. Pre-trained embeddings from the word2vec module of the python gensim library were used.

There is also a straight forward interpretation of elementwise addition and subtraction of these word embeddings. In some sense, the d-dimensions of the embeddings can be interpreted to be metrics of different properties. For instance, the sum of the embeddings for the words *doctor* and *animal* is most similar to the embedding for the word *veterinarian*. There is also reasonable syntactic results when performing simple mathematical operations. It would for instance be expected that the difference between *running* and *run* is similar to the difference between *swimming* and *swim*. This can roughly be expressed as below.

$$\textit{running - run} \approx \textit{swimming - swim} \tag{2.14}$$

Indeed, taking the embeddings for the words and calculating *running - run + swim* results in a vector which is most similar to the embedding for the word *swimming* using the 44,000 most common words in the python gensim implementation of word2vec.

2.3 Classification Models

The text vectorization methods presented in the previous section extracts features from raw text to a vector. This is then used as input to more general classification models which are not exclusive for natural language processing. These models are briefly covered in Section 2.3.1.

[1] See for instance: `https://code.google.com/archive/p/word2vec/`

In other models, the text is entered as a sequence. One way to handle this is using recurrent models, which take the order of words into account. These models are covered in Section 2.3.2.

A recent addition to the family of NLP-models are transformer based models, where the concept of attention between words are central. Transformer models are the subject of Section 2.3.3.

2.3.1 Tree-based & Feed Forward Models

Random Forests. A random forest classifier is an ensembling technique, consisting of several decision trees. Predictions are made by taking the mode (for classification) or the average (for regression) of the predictions from all of the decision trees. Each decision tree in a random forest classifier is fitted to a bootstrap sample of the training set. Every tree is likely to overfit to the data it is presented with, but the full forest of trees has better generalizability as a consequence. The random forest model is a widely popular model which tends to work well without much tuning out of the box for shallow machine learning tasks (Chollet, 2017).

Logistic Regression. Logistic regression is a commonly used baseline for machine learning classification tasks. Even though it is an old model, it can still provide good results for a wide range of classification tasks (Chollet, 2017).

Support Vector Machines. Support vector machines are a group of models which can be used for both classification and regression. The core idea of a classification support vector machine is to find a suitable decision boundary between data points of different classes. This decision boundary is a hyper plane in some dimension depending on the number of features in the input data. The input data is mapped to a higher dimensional representation where the samples are easier to separate. This is however only efficient if a *kernel function* can be constructed, i.e. a mapping k which satisfies $k(x, y) = \varphi(x) \cdot \varphi(y)$ for some function φ. The best decision boundary is then found by maximizing the distance between the closest samples and the hyper plane in the high dimensional representation. Support vector machines can quickly make predictions on unseen data since the new sample only has to be evaluated against the decision boundary consisting of the hyper plane in the high dimensional representation (Chollet, 2017).

Multi Layer Perceptron. A multi layer perceptron, or feed forward network, is a neural network that seeks to approximate some function f. For a classifier, this function would map features x to some prediction $\hat{y}$ of the true value y. The prediction depends on the weights and biases w, b of the network.

The core part of the multilayer perceptron is the single perceptron unit displayed in Figure 2.1. It is easy to see that the prediction $\hat{y}$ is a function of the inputs, weights and biases as $\hat{y} = \sigma(x_1 w_1 + x_2 w_2 + b)$, where σ is the activation function denoted by a sigmoid in the figure.

The training of the model will select the weight and biases such that the predictions are as accurate as possible. The bias term b is usually included in the term *weights* as w_0. This is made possible by adding an extra feature $x_0 = 1$, implying $b = w_0 x_0 = w_0$. This simplifies notation, especially for matrices.

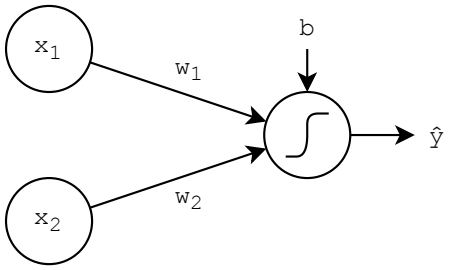


Figure 2.1: A single perceptron unit with two inputs x_1, x_2, two weights w_1, w_2 and one bias b. The sigmoid in the prediction node is an activation function usually denoted $\sigma(x)$.

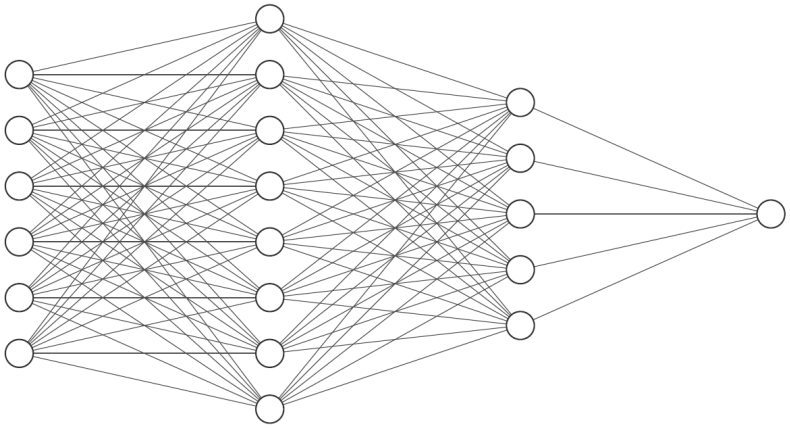


Figure 2.2: Overview of a densely connected feed forward network with six inputs, two hidden layers and one output node.

The performance of classification is measured by some loss function J, which depends on the training samples (x, y) and the weights w. A common loss function for binary classification is binary crossentropy, defined in Eq. 2.15. The notation is such that y is the true label, $\hat{y}$ is the predicted label and $p(\hat{y})$ is the predicted probability of the sample being of label 1. Conversely, $1 - p(\hat{y})$ is the probability of the sample being of label 0.

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} y_i \log\left(p(\hat{y}_i)\right) + (1 - y_i) \log\left(1 - p(\hat{y}_i)\right) \tag{2.15}$$

A single perceptron unit is not too exciting, but when combined into a network, it is more powerful. Leshno et al. (1993) proved that a neural network with a locally bounded piecewise continuous activation function can approximate any continuous function to any degree of accuracy if and only if the activation function is not a polynomial. This is known as the universal approximation theorem. While this implies that a sufficiently large multi layer perceptron is able to represent any continuous function, it does not guarantee that a network is able to learn this representation. Furthermore, the size of the network is unrestricted and can in practice be infeasible computationally (Goodfellow et al., 2016).

Combining the basic functionality in Figure 2.1 with the network in Figure 2.2, a neural network is achieved. This can function as both a regression and classification model. For

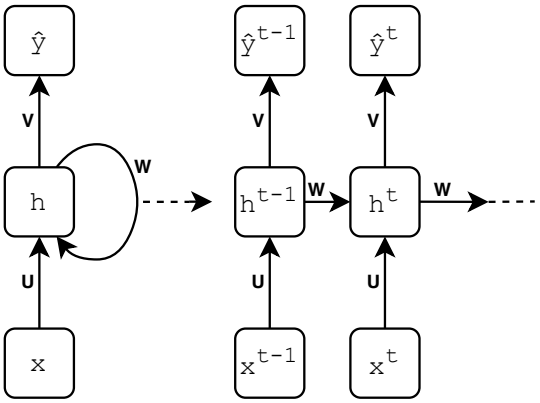


Figure 2.3: General graph of a sequence-to-sequence RNN unfolded through time, where a sequence of inputs x predicts a sequence of outputs y. $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ are weights that are shared throughout the sequence, independent of the position.

regression, the activation function in the final node is typically a linear function as $\varphi(x) = cx$, for some real constant c. For classification, it is usually some sigmoid function bounded between 0 and 1, such as $\varphi(x) = \dfrac{1}{1+e^{-x}}$. The most popular activation function in the nodes which are inside the network is the rectified linear unit function (ReLU), defined as $\varphi(x) = \max(0, x)$ (Goodfellow et al., 2016). These functions are all conveniently implemented in Keras, as well as a structure using different layers to build a model (Chollet et al., 2020).

2.3.2 Recurrent Models

A drawback which affects all of the models presented in the previous section is that the order of words is not taken into account. Of course, the order can be considered in the pre-processing if some more complex transformation is used (e.g. Sentence-BERT), but these transformations usually utilize some recurrent or attention-based method. A recurrent model takes the sequence order into account.

Recurrent Neural Networks. A recurrent neural network (RNN) is rather similar to a multilayer perceptron, but it takes previous inputs into account. The input to an RNN is a sequence of samples. In natural language processing, a sequence is typically a sentence. An RNN can however be used for other tasks where the order of samples are important, e.g. time series forecasting.

Obviously, a sentence can not be defined just by the word it contains, one must also take the order of the words in the sentence into account. We keep a general idea of the meaning of what we have previously read and adjust the idea depending on the new words in the sentence. Recurrent neural networks work in a rather similar way, storing hidden representations of inputs and conveying that information to the next input in the sequence (Goodfellow et al., 2016).

As seen in Figure 2.3, an RNN can be seen as a feed-forward neural network where previous time steps are used as input. Assuming a hidden layer activation function σ_h and an

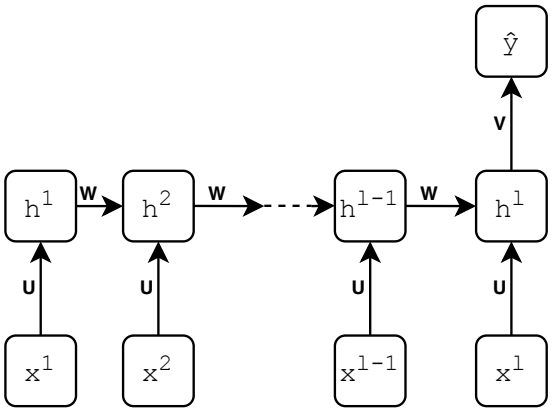


Figure 2.4: Schematic graph of a sequence-to-one RNN unfolded through time, where a sequence of inputs $(x^{(1)}, ..., x^{(l)})$ predicts one output y.

output activation function σ_o, the recurrent nature of the model can be further explained. Let the other notation be as in Figure 2.3.

$$\hat{y}_t = \sigma_o(V \cdot h^{(t)}) = \sigma_o(V \cdot \sigma_h(Ux^{(t)} + Wh^{(t-1)})) \tag{2.16}$$

$$h^{(t-1)} = \sigma_h(Ux^{(t-1)} + Wh^{(t-2)}) \tag{2.17}$$

Clearly, the prediction $\hat{y}^{(t)}$ is dependent on the previous inputs $x^{(i)}$ and hidden states $h^{(i)}$.

The model structure is similar for sequence-to-one classifications, i.e. a sequence

$$(x_i^{(1)}, x_i^{(2)}, ..., x_i^{(l)})$$

predicts one label $\hat{y}_i$. Given a sequence length l, only the final step $x_i^{(l)}$ provides an output y_i. This output is however influenced by the previous inputs $x_i^{(t)}$ through the hidden layers $h^{(t)}$ and the weights W, see Figure 2.4.

Long Short-Term Memory (LSTM). Even though the general idea of the previously described method is widely used, the model is not used precisely as described. The simple RNN-structure suffers from the *vanishing gradient problem*. To understand this problem, one must understand how the network is trained, which is outside the scope of this report. In short, long term dependencies are too many time-steps away to have an impact when the network is trained.

In practice, either Long-Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) are used instead. The main difference is that information that is deemed important is allowed to pass on to later time-steps without too much interference from hidden several dot products and activation functions. This alleviates the vanishing gradient problem, even though the core idea is the same as the previously described simple RNN (Chollet, 2017).

Bidirectional LSTM. As a final note for the recurrent neural network approach, it should be mentioned that when RNNs are used in natural language processing, they are often

wrapped with a bidirectional layer. This reverts the input sequence and enters the sequence in both the original and the reverse direction to two separate RNNs, usually LSTM or GRU.

The usefulness of this is particularly intuitive when looking at the network in Figure 2.3. When processing the entry $x^{(j)}$, only the entries $t < j$ are known. However, tokens later in the sequence might have an impact on the previous outputs of the model. Bidirectional LSTMs can catch patterns that are overlooked by regular LSTMs.

Bidirectional LSTMs have been widely popular in the field of natural language processing. In 2017, the model behind Google Translate was powered by seven stacked LSTM layers (Chollet, 2017).

2.3.3 Transformers

In 2017, Vaswani et al. wrote a paper, *Attention is all you need*, which had a considerable influence on language modelling and machine translation. The authors presented a new model – the Transformer – which does away with the recurrence and instead only uses the concept of attention. One of the problems with recurrent networks is the lack of parallelizability when training the network. The Transformer allows for more parallelization than previous models and consequently trains quicker than models based on recurrence. This is proved by a new state of the art score of 28.4 BLEU on the WMT 2014 English-to-German translation task which was trained at a fraction of the training cost compared to the previous state of the art models.

In the rest of the section, I will follow Vaswani et al. (2017), as well as a more detailed description by Alammar (2018) to describe the principles behind the Transformer. The explained model is used for sequence-to-sequence modelling such as translation, but is also the foundation for BERT which can be used for classification, and is presented later. This is the motivation for this rather detailed section on transformers, even though the model itself is not used in this project.

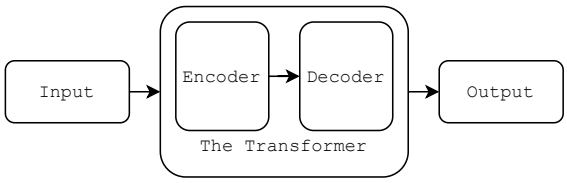


Figure 2.5: Schematic graph of a transformer.

The core idea is to encode the input into a manageable, general representation and then decode this representation into the sought output. Both the encoder and decoder parts consist of 6 identical layers of encoders and decoders.

Transformer Tokenization

Firstly, the input to the model is tokenized in a different way than what has been described for previous models. It is similar in the sense that each token is transformed into an embedding. Vaswani et al. (2017) uses a WordPiece embeddings of dimension $d_{model} = 512$. The way the model handles positional information of words is however quite different.

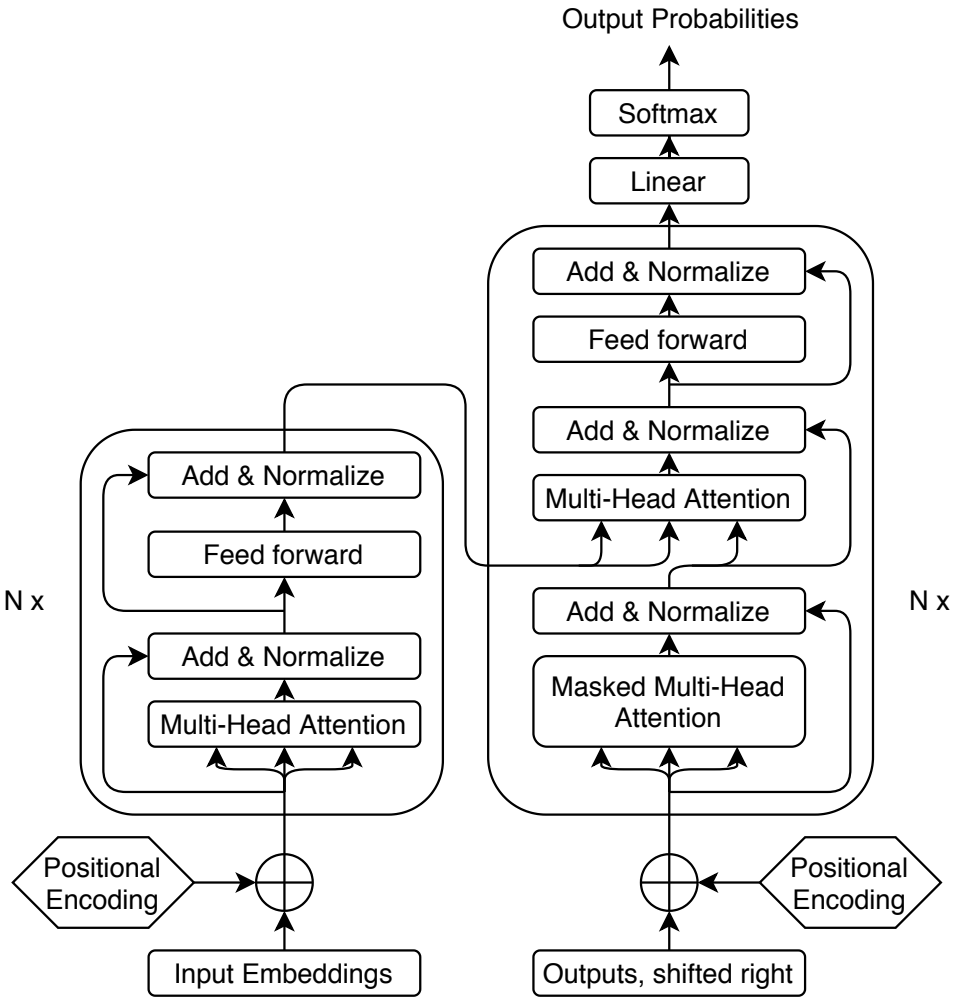


Figure 2.6: Model architecture of the transformer with the encoder to the left and the decoder to the right. The paper uses $N = 6$ layers for both the encoder and the decoder. After Vaswani et al. (2017).

Rather than letting a sequential input to the model represent the position of tokens similar to an RNN, the positional information is included in sine and cosine functions with different frequencies.

$$PE(pos, i) = \begin{cases} \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), & i = 2k \\ \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right), & i = 2k+1 \end{cases} \qquad k \in \mathbb{N} \tag{2.18}$$

In Eq. 2.18, pos is the position of the token in the sentence and i the dimension of the embedding. One word has one position pos and $d_{model} = 512$ dimensions.

This way of representing the positions of a word might seem cumbersome at first, but it has some nice features:

Dimensional consistency. These positional embeddings can be chosen to have the same di-

mension as the vectors used for the word embedding. In the original papers, the positional encodings are added to each element i of the $d_{model} = 512$ dimensions.

Generalizable. This method can extrapolate to sentence lengths not seen in the training data.

Simple relative positioning. The fact that $PE(pos, i)$ can be written as a linear function of $PE(pos, i + k)$ for any k, should according to Vaswani et al. (2017) make it easier for the model to acknowledge relative positions.

No recurrence. This is not unique for this sinusoidal version of positional encoding, but is however one of the main features of the transformer, allowing for more parallel computations.

Encoder & Self-Attention

After the input embedding and positional encoding, a sample is passed through to the stack of encoder layers, shown to the left in Figure 2.6. This is where the concept of self-attention comes in.

At a conceptual level, self-attention for words could be described as how much other words in the sentence represent the current word, i.e. what other words in the sentence the word pays attention to. As an example, consider the following two sentences.

> The gorilla didn't like Manilla, it was too crowded.
> The gorilla didn't like Manilla, it was too tired.

In order to interpret the sentence correctly, a model must understand that *it* refers to *Manilla* in the first sentence and to *the gorilla* in the second. This is the purpose of the self-attention, to detect which words in the neighborhood that are important for the meaning of the current word. I explain the mathematics behind it below. As seen in Figure 2.6, there are 6 layers of multi-head attention with following feed forward layers. One multi-head attention is made up of 8 identical parallel self-attention layers. Each attention layer has unique weights. In the first layer, the inputs are simply word embeddings and positional encodings. Consider the sentence *The gorilla visited Manilla* with some word embeddings x and positional encodings PE:

A	gorilla	visited	Manilla
x_1	x_2	x_3	x_4
PE_1	PE_2	PE_3	PE_4

The vector which is given to the self-attention layer is then h^o as in Eq. 2.21. Note that the dimension for x, PE and h^o is $l \times d_{model}$, where $l = 4$ in this example and $d_{model} = 512$ in the original paper.

$$x = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}^\mathsf{T} \tag{2.19}$$

$$PE = \begin{bmatrix} PE_1 & PE_2 & PE_3 & PE_4 \end{bmatrix}^\mathsf{T} \tag{2.20}$$

$$h^o = \begin{bmatrix} x_1 + PE_1 & x_2 + PE_2 & x_3 + PE_3 & x_4 + PE_4 \end{bmatrix}^\mathsf{T} \tag{2.21}$$

The first calculations in the attention layer are three linear transformations to construct three vectors: a query vector, a key vector and a value vector. The vectors W^Q, W^K and W^V are weights which are tuned during the training and determine the query, key and value vectors.

$$q_i = h_i^o W^Q \tag{2.22}$$
$$k_i = h_i^o W^K \tag{2.23}$$
$$v_i = h_i^o W^V \tag{2.24}$$

These vectors are used to calculate an attention score for a given word i versus all other words j in the sentence. The score is calculated as the dot product of q_i and k_j and scaled by $\frac{1}{\sqrt{d_k}}$ where the original paper uses a dimension of $d_k = 64$ for the query, key and value vectors.

$$\text{score}_{i,j} = \frac{q_i \cdot k_j}{\sqrt{d_k}} \tag{2.25}$$

The scaling by $\sqrt{d_k}$ is made to keep the gradients of the attention layer at a more manageable level. The scores for the word i against all other words j are then normalized with a softmax function, ensuring the sum over all words j sums up to one and that larger scores are boosted. The softmaxed score for word i against word j is then calculated as in Eq. 2.26.

$$\text{softscore}_{i,j} = \frac{e^{\text{score}_{i,j}}}{\sum_{\forall j} e^{\text{score}_{i,j}}} \tag{2.26}$$

Finally, the contribution to the attention for each word is calculated by adding up all the value vectors v_j weighted by the softmaxed score of i and j. This concludes the calculation of the new hidden state h_i^{o+1} for word i.

$$h_i^{o+1} = \sum_{\forall j} v_j \cdot \text{softscore}_{i,j} \tag{2.27}$$

Figure 2.7 shows a flow graph over how this is calculated.

The details of the calculation until now have concerned one word at the time in the sentence. It can however be neatly compressed into matrix form.

Letting h^o be as defined in Eq. 2.21, we can write the query, key, and value vectors as matrices where every row corresponds to the vector for a word in the sentence.

$$Q = h^o W^Q \tag{2.28}$$
$$K = h^o W^K \tag{2.29}$$
$$V = h^o W^V \tag{2.30}$$

The attention of the layer can then be calculated as in Eq. 2.31, by the steps shown for one word in the previous segments.

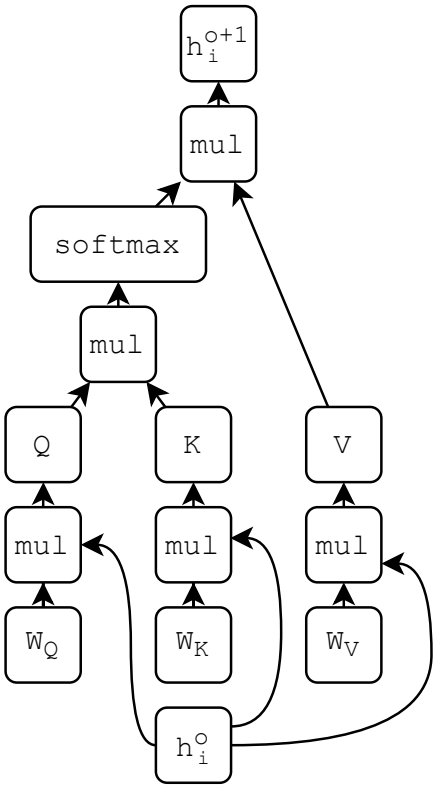


Figure 2.7: Calculation of self-attention from hidden layer o to $o+1$ for word i. `mul` denotes a matrix multiplication or a dot product.

$$\text{Attention}(Q, K, V) = \text{softmax}\Big(\frac{QK^\intercal}{\sqrt{d_k}}\Big)V \tag{2.31}$$

What has been demonstrated until now is the flow for a sample through one attention layer. However, the transformer model uses a multi-head attention layer. This is simply 8 attention layers that run in parallel. The weights across the layers are not shared, but the idea is that having eight attention layers with random initialization enables focusing on different aspects of language. The output of the eight layers are concatenated and multiplied by another matrix of weights W^O in order to reduce the dimensions to $l \times d_{model}$, which are the same dimensions as was fed to the model in Eq. 2.21.

Note that the first inputs to the model are word embeddings and positional encodings, but the rest of the encoder layers has the output from the previous layer as input, which is of the same dimension as the input embeddings.

The final step in the encoder layer is a position-wise feed forward network, roughly as in Figure 2.2. The network has one hidden layer of size $d_{ff} = 2048$. The input and output are both of size $d_{model} = 512$. The difference with a regular multi layer perceptron is that the weights are applied identically to each position, greatly reducing the number of tuneable weights. The hidden layer has ReLU activation functions, so the output for any input can be conveniently written as in Eq. 2.32

$$F(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2.32}$$

Another important property of the Transformer is the residual connections seen in Figure 2.6. These allow information to be transferred to a new layer without being distorted in the attention calculations. Vaswani et al. (2017) claim this is particularly important for the positional encodings to stay intact deeper into the network.

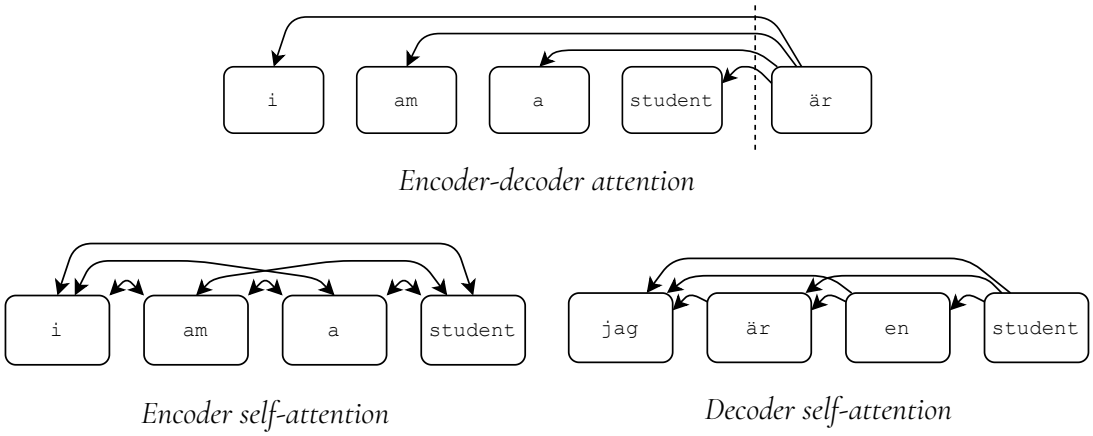


Figure 2.8: Different types of attention in a transformer. Imaged by a translation task from the English sentence "i am a student" to the Swedish sentence "jag är en student".

Decoder & Output

There are many similarities between the encoder and decoder, but some things differ. The output from the encoder is used to calculate the attention vectors $\boldsymbol{K}$ and $\boldsymbol{V}$ which are used to predict a sequential output from the decoder. The first input to the decoder are these attention vectors and a start-of-sequence token. These then progress through the decoder layers and produce an output. The output is made by a linear transformation to a logits vector, where each entry is an index in a vocabulary. A softmax function is applied, and the entry in the vector with the highest values is chosen as the prediction. The predicted word is then transformed with word embedding and positional encoding as in the input, and the next word in the sequence is predicted using the previous word as input. The calculated attention vectors $\boldsymbol{K}$ and $\boldsymbol{V}$ are the same for all of the positions in the decoder.

One large difference to the encoding part is that the self-attention layers are only allowed to attend to previous inputs, i.e. words to the left. This is done by setting the entries for words to the right of the currently processed word to $-\infty$ in the dot product of $\boldsymbol{Q}$ and $\boldsymbol{K}$ before the softmax as seen in Figure 2.7. This ensures no attention is given to latter words in the sentence in the decoding part. The decoder does however have access to the key and value vectors from all of the words in the input. A general flow chart of how words influence each other in a translation task is shown in Figure 2.8.

BERT

The transformer presented by Vaswani et al. was influential for several other models. Devlin et al. presented the language representation model BERT in 2018. BERT is short for

Bidirectional Encoder Representations from Transformers. This model achieved new state-of-the-art results on eleven NLP tasks. The success of BERT is largey due to the fact that it can be used for a wide variety of NLP tasks with only small changes to model architecture and further training.

A lot of the mechanics of BERT is included in the encoder part of the transformer seen to the left in Figure 2.6. The bidirectionality in the model is represented in the same way as the self-attention in the encoder of the transformer. As mentioned in the explanation of the transformer, any given word in a sentence is allowed to give a high attention score to all other words in the sentence. Even to words to the right of the word. This is not the case for the decoder. As seen in Figure 2.8, the decoder self-attention only has access to the previously decoded values. The attention structure from the input sequence generated in the decoder is however the same throughout the decoding. The bidirectionality in BERT is thus not quite the same as the bidirectionality in an LSTM or GRU, since the latter reverses the input sequence to achieve bidirectionality. In BERT, the sequences are handled in a more parallel sense, as the positions are considered only as the positional embeddings – not by the position in the sequence.

Devlin et al. implemented two versions of BERT with the same structure but different sizes – $\text{BERT}_{\text{LARGE}}$ and $\text{BERT}_{\text{BASE}}$. The base model has 110M parameters, including 12 stacked transformer encoder blocks, 768 dimensions in the input/hidden states/output vectors and 12 self-attention heads in every multi-head attention block. The large model has 340M parameters, 24 transformer blocks, 1024 dimensions and 16 self-attention heads.

The input and output representations of BERT share some properties with the Transformer, but with some extensions. Each input in a sequence is the sum of three embeddings – position, segment and token, see Figure 2.9.

The position embeddings are represented in the same sinusoidal way as described for the Transformer. The segment embedding is an indicator for which segment a sentence belongs to, used where pairs of sentences are given as input. These are separated by a special [SEP]-token, but also by the segment embeddings which specify if a word belongs to segment A or B. If the task does not contain sentence pairs, all words belong to segment A. The tokens are created by the same method as for the Transformer, i.e. WordPiece embeddings.

There is also an initial token with several purposes seen in Figure 2.9 – the [CLS]-token. This indicates that a new sequence is starting, but also serves as an aggregate sequence representation. Since this token aims to capture the summary of a sentence, it is suitable as feature extraction for classification tasks after fine tuning. Each output token is of the same dimension as the inputs and hidden layers, either 768 or 1024 depending on if it is the base or large model.

While a lot of the BERT model structure is similar to the encoder part of the transformer, the training is different. BERT is pre-trained on two unsupervised tasks: *masked language model* and *next sentence prediction*. The model can then be fine tuned to the specific task it will be used for.

Masked Language Model. Some words in the input are randomly replaced with another token with a probability of 15 %. The token is replaced with a special [MASK]-token 80 % of the time, a random token 10 % of the time and not replaced at all 10 % of the time. The reason for not using the [MASK]-token all the time is that this token is only seen in the pre-training, not in the fine-tuning or the prediction. Replacing with random words or the actual word improves the models performance on un-masked sentences.

Next Sentence Prediction. This trains the model to learn relationships between sentences. Two consecutive sentences are randomly picked from the training corpus, as well as two random sentences. 50 % are given the label IsNextSentence and 50% NotNextSentence. Consider for instance these pairs.

> The man went to the store. He bought a gallon of milk.
> The man went to the store. Penguins are flightless.

Where obviously the first pair is reasonable and the second is not.

Devlin et al. pre-trained the model on two large corpora, the BooksCorpus (800M words) and the English Wikipedia (2,500M words).

After the pre-training, the model is fine-tuned for the task at hand. This implies updating the weights in one or several of the upper transformer layers. One of the major benefits of BERT is that it can be used for most NLP tasks with great performance even compared to more task specific models. The inputs and outputs just have to be changed accordingly.

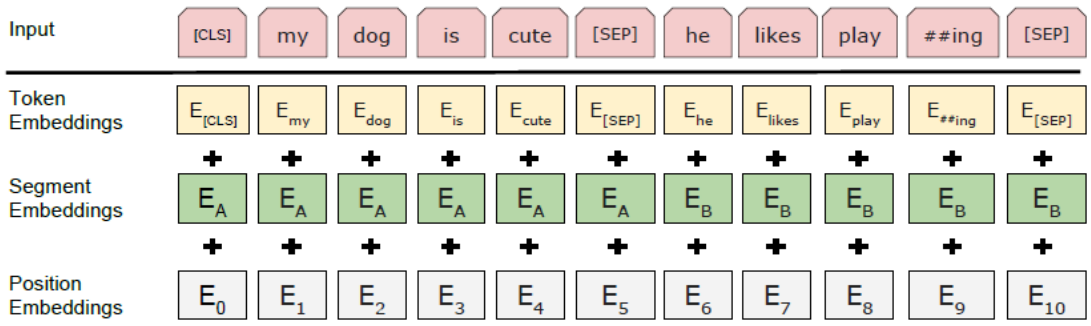


Figure 2.9: BERT input visualization for a sentence pair, A & B. After Devlin et al. (2018).

Sentence-BERT

A further development of BERT for representing sentences is Sentence-BERT. Reimers and Gurevych presented SBERT in 2019, where the main motivation was to be able to derive semantically meaningful sentence representations. The authors define semantic similarity as proximity in vector space, for instance measured by cosine similarity. SBERT uses a pre-trained BERT model and computes some pooling operation on the last layer. As seen in Figure 2.9, the length of the output sequence is equal to the input sequence. In order to get a fixed size representation of a sentence, the dimension is reduced to the same as the dimension of the words.

SBERT allows for three different pooling operations – using the [CLS]-token, mean pooling or max-over-time pooling. The model is fine tuned using siamese and triplet networks. These are BERT networks with shared weights. Two sentences are fed into one network each and the similarity is compared between them. The authors explore different objective functions, one being mean squared error loss of the cosine similarity between two sentences.

2.4 Overfitting & Hyperparameter Optimization

This section concerns the problem of overfitting in a machine learning context. The contents in Sections 2.4.1 through 2.4.4 mainly follow Goodfellow et al. but is covered thoroughly and in a similar fashion in most basic machine learning summaries.

2.4.1 Train, Validation, and Test Set

A popular definition of machine learning is quoted from Mitchell:

> *"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."*

The main goal of machine learning is to be able to have some predictive power on previously unseen data. This is a difference between machine learning optimization and optimization in general. A model is said to *generalize* well if it shows good performance on data it has not seen before. This is also how a model is typically used for some application – we want to use the model to predict things of uncertainty, not things we already know.

Suppose we have gathered a dataset $\mathcal{D}$ to train a model. In order to approximate how the model performs on unseen data, we can hide a partition of the dataset during training. Letting our initial dataset $\mathcal{D}$ be the set $\{\mathcal{D}_{train}, \mathcal{D}_{test}\}$, we optimize the model on the data $\mathcal{D}_{train}$ by minimizing some loss function $\mathcal{L}$ over the samples in $\mathcal{D}_{train}$. The generalization error of the trained model is then approximated by $\mathcal{L}(\mathcal{D}_{test})$.

The training of a machine learning model can be simply expressed as minimizing these two objectives:

1. Minimize the training error.
2. Minimize the difference between the training error and the test error.

If the training error is small while the difference between the training error and the test error is large, we say that the model has *overfitted* to the training data. This usually means that the model has too high capacity and tries to represent too complex patterns. If the training error is large, but the difference between training error and test error is small, it indicates that the model is *underfitted*. This implies that the model is not complex enough.

A fundamental rule of machine learning is that the results on the test set can not influence the model structure in any way. The purpose of the test set is to estimate how well the model performs on new data. It is not a metric that is to be used for model optimization. Still, the objective is to optimize the model on unseen data. This is addressed by introducing a *validation set*.

Let the full dataset $\mathcal{D}$ be partitioned into three sets, $\{\mathcal{D}_{train}, \mathcal{D}_{validation}, \mathcal{D}_{test}\}$. The train set is then used to optimize the parameters of the model, while the validation set is not included in the training. The generalization error is approximated by the validation error. The validation error is by contrast to the test error however allowed to influence the model structure. This is typically done by adjusting the *hyperparameters* of the model and introducing some *regularization*, covered further in the following sections.

2.4.2 Hyperparameters

A machine learning model is defined by its tuneable parameters and its hyperparameters. The tuneable parameters are normally just called parameters and are the trainable weights in a model. The hyperparameters are properties of the model that must be set before the training is started. Hyperparameters for a neural networks are typically the number of nodes in each layer, which activation function to use in the nodes, the depth of the network, what sort of optimization routine is used, the learning rate, etc.

Many of these parameters can adjust the capacity of the network, i.e. how complex structures the network can model. If the model is too complex, it might be too sensitive to the training data and adapt to patterns in the training data which are not transferable to unseen data. If the model is not complex enough, it might fail to retrieve important patterns from the training data.

Other parameters rather adjust the speed and precision of training. The learning rate is such a parameter, deciding the size of the steps in the gradient descent-like optimization algorithm. A too big learning rate can imply that the optimization fails to reach the sought minimum, a too small learning rate can imply that the optimization converges too slowly (Goodfellow et al., 2016).

2.4.3 Regularization

Regularization is a way to prioritize more likely parameters over less likely parameters. It assumes we have some prior belief of which parameters that are more likely. The basic principle is the same as for Occam's razor – if two solutions to a problem give equal results, choose the simplest one.

This concept is used to combat overfitting and to make the model focus on the most important patterns in the data. A model with high regularization is penalised for making complex decision rules, generally giving a higher training error. A simple way to introduce regularization is to add a *weight decay* to the loss function. Given the weights w in a neural network and some cost function $\mathcal{L}$, a new cost function $\mathcal{L}_R$ with weight decay can be constructed as in Eq. 2.33.

$$\mathcal{L}_R(w) = \mathcal{L}(w) + \lambda w^T w \tag{2.33}$$

The parameter λ is here a hyperparameter used to control the regularization strength. A higher λ implies more regularization.

Hinton et al. presented another regularization method used in the context of neural networks in 2012 – *dropout* regularization. This removes a node in the network with probability p during every step in the optimization. It is arguably quite similar to training several different networks, since the network is constantly changing and removing nodes, altering its structure. Dropout can be seen as a cheap way of training and evaluating an ensemble of exponentially many neural networks.

The nodes are only removed with probability p during training. When the model is used for prediction, the weights of the nodes where dropout has been used is multiplied by $1 - p$. The argument for why this is reasonable is that seems to capture the correct expected value

of the output from that node. Not a very theoretical argument, but it seems to work well (Hinton et al., 2012).

2.4.4 Cross-Validation

If the available dataset is small, it might be unrealistic to have the same statistical properties in the training, validation and test data. A small size also gives a large variance on both the validation and test set, making it difficult to select hyperparameters and estimate the performance on unseen data. A way to improve the accuracy of the estimate is to use k-fold cross validation.

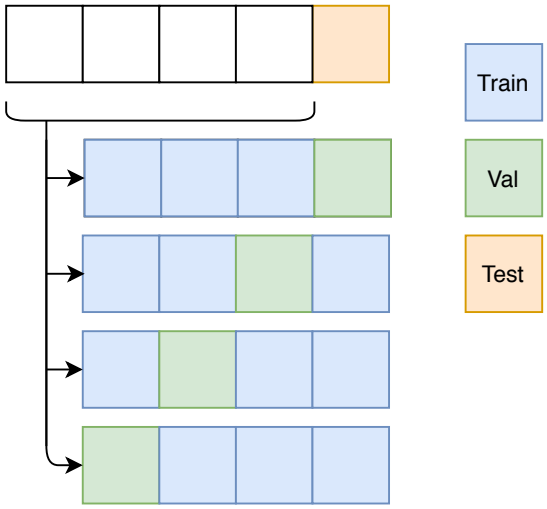


Figure 2.10: Overview of how a dataset is used for k-fold cross validation with $k = 4$.

Firstly, an independent test set is extracted from the dataset. The test set should have the same distribution of labels as the dataset. This is seen as the orange square in Figure 2.10. The test set is typically 10-20 % of the dataset, it does however depend on how large the dataset is.

The remaining samples in the dataset are then divided randomly into k parts of equal size. A validation error with lower variance can then be calculated by taking the mean over all k validation errors after fitting the model to the k different training sets. This validation error can then be used to determine which set of hyperparameters gives the best performance.

Finally, the performance of the model on unseen data is estimated on the test set with the hyperparameters found in the k-fold cross validation. If a more accurate test error is desired, an additional loop over k test sets can be performed in a similar fashion as for the validation set. This is however not used for hyperparameter selection, only to get a better estimate of the generalization performance (Goodfellow et al., 2016).

2.4.5 Grid Search & Random Search

Using the k-fold cross validation technique previously presented can be computationally expensive if a large set of hyperparameters are to be tested. Suppose we want to try the number

Hyperparameter	#
Dropout rate	3
Learning rate	5
Hidden layers	3
Nodes	10
Optimizer	5
Activation function	3
Batch size	5
Total # combinations	**6750**

Table 2.2

of hyperparameters in Table 2.2. This gives 6750 combinations of parameters. Furthermore, using k-fold cross validation with $k = 10$ would imply 67,500 fitted models in order to find the best set of hyperparameters. This might obviously be reasonable for smaller sets, but it rather quickly grows infeasible to do an exhaustive search over all possible combinations.

A common and effective alternative is to use a random search. Only the possible distributions of the hyperparameters are defined, and a fixed number of combinations of these are evaluated. Random search finds good solutions quicker than grid search, as it reduces the validation error more for a given number of tested hyperparameters (Goodfellow et al., 2016).

2.5 Performance Metrics

The metrics used to evaluate the models are accuracy and F1-score. The F1-score is defined by the precision and recall, so these metrics are explained as well. For convenience a binary confusion matrix is displayed in Table 2.3. Visualizations for accuarcy, precision and recall are shown in Figure 2.11

		Predictions	
		0	1
Actual values	0	True Negatives (TN)	False Positives (FP)
	1	False Negatives (FN)	True Positives (TP)

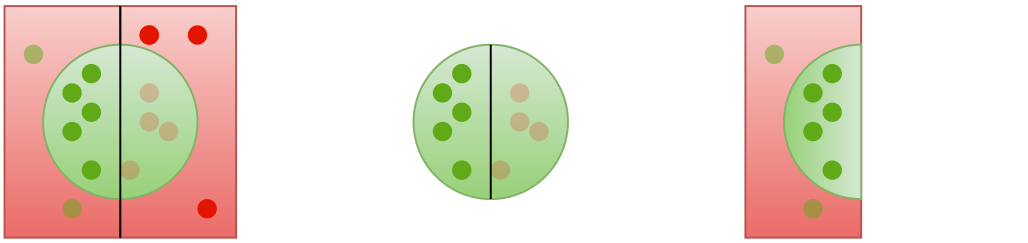
Table 2.3: Confusion matrix for binary classification.

Accuracy: The most intuitive measure of performance, the ratio of correct classifications to the total number of classifications. If the labels of a data set are symmetric, this is a good measure of performance. However, if 90 % of the samples are of label a and 10% of label b, a model which constantly predicts label a gets an accuracy of 90%. It doesn't reveal any information about which data is classified incorrectly.

Precision: The number of correctly classified samples in one class divided by the total number of predictions of that class. The precision P for class 0 in Table 2.3 is calculated as $P = \frac{TN}{TN + FN}$.

Recall: The number of correctly classified samples in one class divided by the total number of samples in that class. In other words, a measure of how many percent of a class that was found. In relation to Table 2.3 it is calculated as $R = \frac{TN}{TN + FP}$. Recall is also referred to as *sensitivity* in statistical literature.

F1-score: A measure which combines precision and recall as the harmonic mean of the two. Calculated as $F1 = 2\frac{R \cdot P}{R + P}$. An F1-score of 1 implies perfect recall and precision.



(a) Accuracy = $\frac{8}{14}$ **(b)** Precision, green = $\frac{5}{9}$ **(c)** Recall, green = $\frac{5}{7}$

Figure 2.11: Visualizations of different performance metrics for a binary classifier. The dots are samples and the classifier predicts *red* in the red area and *green* in the green area.

The precision, recall and F1-score are metrics which are calculated for every class in a classification problem. In order to get a single metric for a set of samples, these can be weighted by the number of true labels of each class. Hence for a binary classification problem, $F1_w = F1_0 \cdot w_0 + F1_1 \cdot w_1$, where w_i is the ratio of labels in class i in the training set (Ting, 2017). A more interesting metric for a skewed data set is the macro F1-score, which averages the F1-score for each label regardless of the ratio of labels. Consider a dataset with 99% of label *A* and 1% label *B*. If a classifier predicts *A* for all targets, the weighted F1-score would be $F1_w = 1 \cdot \frac{99}{100} + 0 \cdot \frac{1}{100} = \frac{99}{100}$. The macro F1-score would take the average over both F1-scores, i.e. $F1_m = 0.5$. This is arguably a more reasonable measure for a skewed set of labels.

Chapter 3
Method

This chapter presents the data used for training and evaluating models, the models I will test and how the text is processed to be of appropriate format for the models.

The financial indices I considered in this project are the S&P 500 and the US treasury rate with 1 and 3 years to maturity. These are collected from Refinitiv, but are publicly available through free channels as well.

- S&P 500 (*Standard & Poor's*) is a stock market index reflecting the performance of the 500 largest companies listed on the stock exchanges in the United States.
- The US Treasury rate with maturity n is the yield on a government zero coupon bond with the same maturity.

Two text datasets are used. The first one is a dataset of movie reviews from IMDb, commonly used for benchmarking models. The purpose of this dataset is to ensure the models perform well on text classification tasks. This makes the interpretation of the performance on the financial tasks easier.

The other text dataset is a collection of financial news from 2006 to 2013. This is combined in various ways with the time series data of financial indices, described in Section 3.2.

3.1 Data Collection and Pre-Processing

All models are trained on two datasets – a labeled movie review dataset from IMDb for benchmarking and an unlabeled financial news dataset from Reuters combined with time series data for the financial interpretation.

3.1.1 IMDd Dataset – Benchmarking

The IMDb dataset is a commonly used benchmarking dataset for binary classification of text data. The dataset was compiled by Maas et al. (2011) and contains 50,000 polarized

movie reviews with 25,000 positive and negative reviews respectively. Evaluating models on this dataset can validate that the models can capture some meaningful information from text data and thus give better insight about the performance on the financial task. It also generally confirms that the model is implemented appropriately. For comparison, Xie et al. achieved 95.8% accuracy using BERT large and unsupervised data augmentation on the IMDb dataset. This is in the state of the art-region, even though there are models that have scored slightly higher.

Two reviews are presented below, first a positive one and then a negative one.

> "This program was quite interesting. The way the program was displayed made it all the more interesting. String Theory is also very interesting to listen too. The whole three hours in my opinion were well worth it. I enjoyed listening to the ideas given by the physicists. Extra dimensions really boggle the mind. If you have the chance, watch this amazing documentary"

> "There wasn't a 0 in the voting option so i was compelled to use the next available figure. It is a sad day for bollywood when such type of movies which have star-cast actors is nothing more are than a bunch of juvenile acting, and an awful script. This movie is nowhere near to be called a clone of Hitch. Salman khan with his usual take-off-you-shirt theme and Govinda with his in-humorous laughs. If somebody had told 2 decades ago that I would be writing a comment on Salman (after his success with Maine Pyar Kiya), I would have written him/her off. "

3.1.2 Reuters Financial News Dataset

The corpus used for this project consists of 109,110 financial news articles from Reuters. It was first compiled and used by Ding et al. (2014) for predicting stock price direction. Only the titles of the news data is used to train the model, as suggested by Ding et al. (2014). The reasoning behind this is that including the contents of the articles do not significantly improve the model.

The data contains financial news regarding the US market from the 10th of October 2006 to the 11th of November 2013, with approximately 5-50 news articles per day and is publicly available[1]. A few examples of headlines follows.

> 2006-12-18 – "Shares fall on tech worries, oil stocks"
> 2008-09-12 – "S&P analyst says does not expect Lehman to fail"
> 2008-09-15 – "Lehman files for bankruptcy, plans to sell units"
> 2010-06-22 – "Market ends down on housing data and technicals "
> 2012-04-11 – "U.S. weighs higher threshold for swap dealers: report"

Pre-Processing

The first part of the pre-processing of text data is similar for all models. The intersection of dates available with news data and financial data is assessed and headlines for these days are

[1] Available at `https://github.com/duynht/financial-news-dataset`.

extracted. The text is quite clean since it is the headlines of published news articles. It does not require as much cleaning as for instance a corpus of tweets.

The next step of the pre-processing is to tokenize the data. This is done differently depending on what model to use, but a parameter that is set for all models is the *vocabulary size*. The vocabulary size is simply the amount of words to keep track of. Keras has a tokenization functionality which includes words based on frequency – the most common words are included. How the tokenization is performed for each model is explained more carefully under each section.

For some of the used models the natural language processing part can be seen as pre-processing, for instance when pre-trained word embeddings are used. This is explained further under each model.

3.2 Time Series Processing

I used three time series for this project – 1 year treasury rate, 3 year treasury rate, and S&P 500. All of these indices are middle rates, calculated as the median average of the bid and ask rate over each day. These are shown in Figure 3.1

Figure 3.1: Development of time series over time. 1 year rate (left), 3 year rate (middle) and S&P 500 (right).

The daily percentage change is calculated to get stationary time series with mean zero.

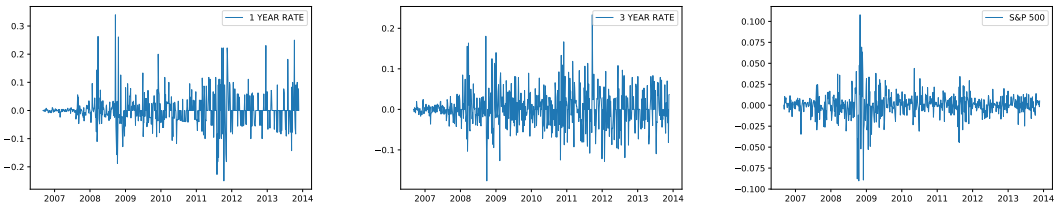


Figure 3.2: Daily percentage change. 1 year rate (left), 3 year rate (middle) and S&P 500 (right).

To calculate ARMA predictions, I fitted a model to the available information at time t. A prediction is then made for time $t + 1$. Since the financial news headlines are the limiting dataset in terms of dates, earlier data can be used for the time series forecasting. The time series are included from 2006-09-01 in order to make a proper prediction for the first sample in the news dataset at 2006-10-20.

After constructing models and forecasting the time series, I compared the forecasts to the actual percentage changes. This is used to create labels for the classification task. If the ARMA prediction is higher than the actual outcome, the label is 0. Otherwise, it is 1. The binary accuracy – the accuracy of the index going up or down – of the ARMA predictions can also be used as a baseline for comparison with other models.

I used the Python package *statsmodels* to fit ARMA models and make predictions. Traditional time series analysis is not the main concern of this project. Therefore, the parameter estimation is rather crude. The AIC is calculated for a few different values of p and q, which suggests an ARMA(1,1) process is adequate.

3.3 Text Vectorization

Some text vectorization techniques are jointly trained with the classification model, and thus not possible to isolate from the model. Others can be trained both jointly and used as a pre-processing layer for another model. This project uses three vectorization techniques. In all of the techniques except for Sentence-BERT, the news titles for one day are concatenated in one document representing the news of this day.

The news dataset contains 27,617 unique words when converted to lower case.

3.3.1 TF-IDF

Term Frequency–Inverse Document Frequency (TF-IDF) is purely a pre-processing technique which outputs large dimensional vectors of the same length as the vocabulary size. This can then be fed into any model. As stated in the previous chapter, this technique does not preserve the order of words, which does not make it suitable to use as input to a recurrent network. We used a tokenizer from the scikit-learn library to transform the texts into TF-IDF vectors.

Even though TF-IDF deals with frequency of words, there is still a computational benefit of having a restricted vocabulary, since this is equal to the input dimension. An appropriate vocabulary size is found by evaluating the validation accuracy for different sizes using logistic regression. This is because logistic regression is optimized quickly and can give an indication of when important information for classifying is lost. For the IMDb dataset, this was found to be 10,000 words.

3.3.2 GloVe

For the word embeddings, I used the GloVe pre-trained word vectors. The vectors have been trained on a dump of Wikipedia from 2014 containing 1.6 billion tokens and the English Gigaword fifth edition dataset, containing 4.3 billion tokens (Pennington et al., 2014). 400,000 uncased words with an embedding size of 300 dimensions are represented in the GloVe dictionary I used. Of the 22,415 words in the news data, 22,107 were available as pre-trained GloVe embeddings.

The tokenization for the GloVe models first converted all words to lower case and reformed some words such as *won't* and *can't* to *will not* and *can not*. This allows for more words to be initialized with pre-trained embeddings since contracted words are handled differently by different tokenizers. The tokenization is done using the tokenizer from Keras.

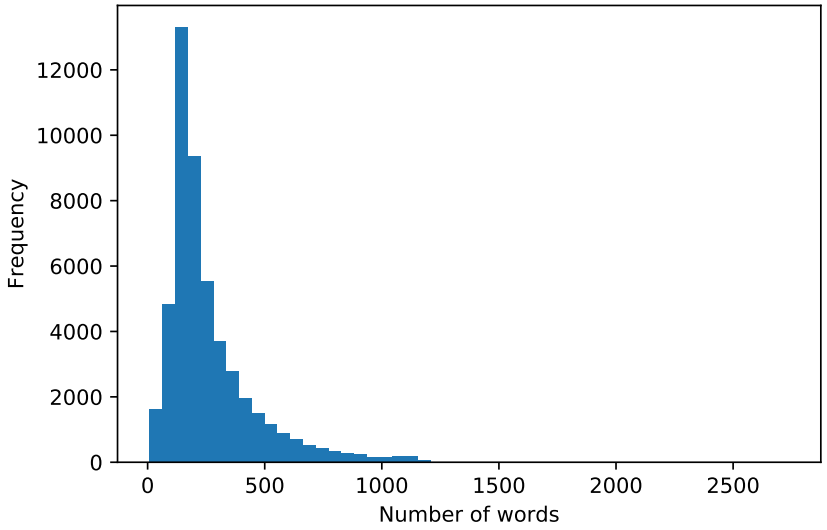


Figure 3.3: Histogram of the number of words in the 50,000 IMDb movie reviews.

For the non-sequential models, the data is converted from $w \times d$ dimensions to $1 \times d$, where w is the number of words in the concatenated news titles for one day and d is the embedding dimensions, here 300. This is done by taking the element-wise average over all d dimensions. In the models using the embeddings as non-trainable pre-processing, this is simply done before the data is fed into the model. In the models where the training of the embeddings are continued, a custom layer in Keras computes the element-wise average after the embedding layer, where the data has been inputted as token indices and the pre-trained embeddings are set in the embedding layer.

For sequential models, the order of words in the titles is kept intact. The number of words in the news titles for one day is shown in Figure 3.4, and the number of words in the reviews of the IMDb dataset in Figure 3.3. In order for each text sample to have the same size, a maximum sentence length is set. Longer sequences are cut of, shorter sequences are padded with zeros. The maximum sequence length is set to 800 for the news data, which well covers the majority of the samples. For the IMDb data, it is set to 500.

3.3.3 Sentence-BERT

Reimers and Gurevych, the authors of Sentence-BERT, have made models publicly available on Github, see Appendix A. The package is called *sentence-transformers*. A pre-trained model is used to embed sentences into vectors of length 768. In order to represent a full document or several news headlines, each sentence is embedded independently. The elementwise average is taken to represent the collection of sentences. These features can then be used as input to classification models.

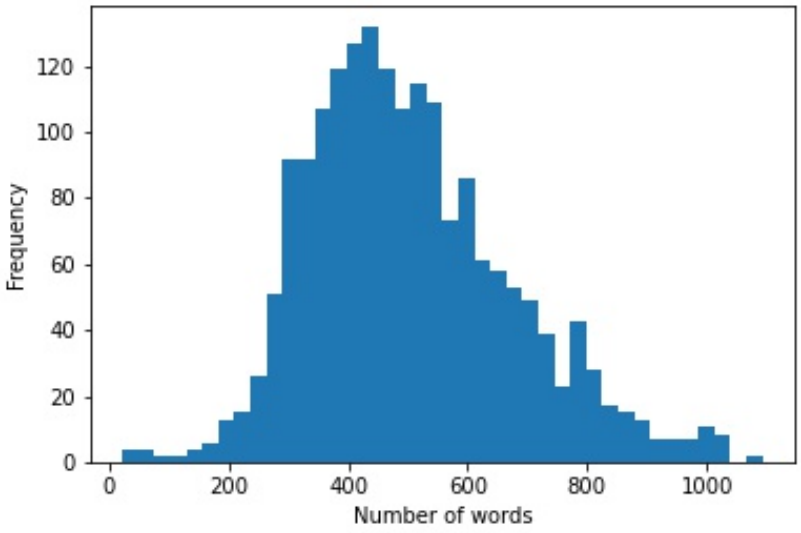


Figure 3.4: Histogram of the number of words in the concatenated news titles for one day.

3.3.4 BERT

The tokenization for the BERT model is more complex than previously described models. As described in Section 2.3.3, each input to BERT is made up of three vectors for every token – *token id*, *segment id* and *mask*. The main part of this is implemented in a tokenizer supplied by Devlin et al. (2018) on Github, see Appendix A. Some processing is done after the tokenization, such as adding [CLS] and [SEP] tokens for the start and end of sequences.

The maximum sequence length for BERT is 512, which implies 65 % of the daily news headlines can not be fully represented. The first 512 tokens of these daily headlines are used in these cases. Shorter sequences are padded with zeros.

3.4 Models

This section explains the models used in the experiments and how they are implemented. Firstly, general feed forward-style models are described, followed by recurrent models, and lastly the transformer model BERT.

Random Classifier. For an imbalanced rather small dataset, a random classifier can provide a basic understanding of what performance to beat to ensure some predictive power in a model. The random classifier can be trained on the training set, or just randomly predict on the test set. I have used two random classifiers which make some use of the training set.

Two approaches were used – *stratified* and *most frequent*. The stratified method respects the distribution of labels in the training data and randomly predicts on the test data with the same distribution. The most frequent-method predicts the most frequent label in the training data on all the test samples.

The random classifiers used in this project only relies on the target variables, so it is not interesting to classify it as sequential or not.

3.4.1 Tree-based & Feed Forward Models

These models are only used with non-sequential data as input, i.e. TF-IDF, average GloVe embeddings and Sentence-BERT. These vectorizations typically have dimensions 5,000–20,000 (TF-IDF), 300 (GloVe) and 768 (Sentence-BERT).

Logistic Regression. We used logistic regression both as a traditional benchmark and a more novel model in combination with GloVe and BERT embeddings. We used the logistic regression model from the python library scikit-learn for the implementation.

Support Vector Machine. We used a support vector machine (SVM) with similar motivation as for the logistic regression – it serves as a well known benchmark model and is rather simple to implement and optimize. We used a support vector machine for binary classification from the scikit-learn library.

Feed-forward Network. A densely connected feed-forward network is evaluated with varying number of hidden layers, nodes and dropout rate. The loss function is binary cross entropy.

The input to the model is either sequential data into an embedding layer (see Figure 3.5) – if the embeddings are continuously trained – or as pre-processed data with static embeddings. For the latter case, this is just a regular multilayer perceptron with rectified linear unit activation function and dropout regularization. The output layer is a single node with a sigmoid activation function.

The model taking sequential inputs has an embedded layer as the first layer. The weights in this layers are set to the pre-trained GloVe embeddings, and are then jointly trained with the other parameters in the model. A maximum sequence length is specified and an average is taken element-wise over all the 300 embedding dimensions. The average is taken where the sum of the absolute value of the embedding vector is non-zero, i.e. the positions in the sequence vector that actually contain a word.

This tensor is then passed into a regular multi layer perceptron as described previously.

The feed-forward network is implemented using the Keras library.

3.4.2 Recurrent Models

The sequential models all allow further training of embedding parameters, since the first input layer is a regular embedding layer. The input dimensions are $w \times d$, where w is the maximum sequence length and d is the dimension of the embedding.

Bidirectional LSTM. The bidirectional LSTM has the same input structure as the feed-forward network, but rather than an averaging layer over all dimensions, an LSTM layer is applied. This preserves the sequential structure of the inputs and enables the model to take the order of words into account.

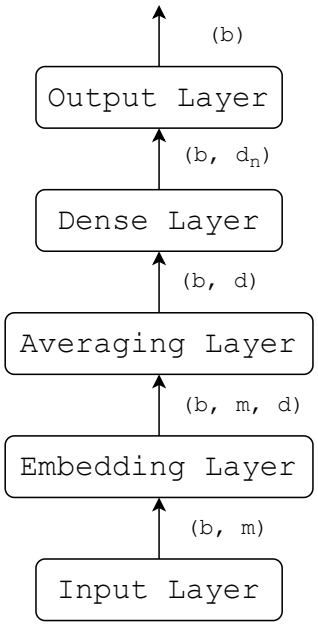


Figure 3.5: Schematic overview of the used feed-forward network with embedded inputs. b is the batch size, m is the maximum sequence length, d is the embedding dimension and d_n is the number of nodes in the dense layer.

The LSTM layer is wrapped in a bidirectional layer, which implies the input sequences are processed both from left-to-right and from right-to-left. Since this configuration takes a lot of time to train, hyper parameters optimization is performed by manually testing a few parameters.

3.4.3 Transformer Models

One transformer model has been implemented, tuned with two different final layers. A pre-trained BERT-base model from TensorFlow hub is used and implemented as a Keras layer. The two final layers of the BERT base model are fine tuned. One model adds a sigmoid output node to the BERT-base model. The other model appends a hidden layer with 256 nodes and dropout regularization. The training is done with hyperparameters recommended by Devlin et al., a batch size of 16 and an Adam optimizer with learning rate 2e-5. The benchmarking on the IMDb dataset is trained for 4 epochs. The financial dataset is significantly smaller (1,846 samples vs 20,000 samples) and is thus trained using an early stopping callback conditioned on validation accuracy.

3.5 Hyperparameter Optimization

The selection of hyperparameters was carried out using either a grid search or a random search over a selected range of parameters. Grid search was used for random forest and random search was used for other models where hyperparameter optimization was made. The

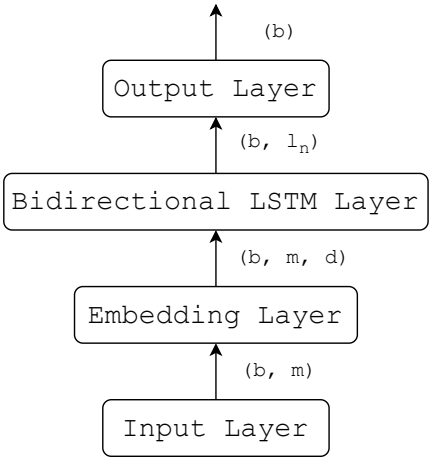


Figure 3.6: Schematic overview of a bidirectional LSTM network with sequential input data. b is the batch size, m is the maximum sequence length, d is the embedding dimension and l_n is the number of nodes in the LSTM-layer.

search was done with either 10 or 3 fold cross-validation depending on the computational feasability. More demanding models like bidirectional LSTM and BERT were fitted by manual tuning of hyperparameters, i.e. using values that seem reasonable.

The neural network-style models were trained using an early-stopping callback. This implies that the model is trained until the validation accuracy has not improved for a number of epochs. The final model is then chosen as the model with the highest validation accuracy.

The parameters for the random forest model were tested exhaustively for the ranges listed in Appendix B. The set of parameters which gave the highest F1-score on the validation set was chosen.

Once the best set of hyperparameters have been found, the models are evaluated on an unseen test set. The recorded metrics are accuracy and F1-score.

The tested ranges and concluded sets of parameters for a selection of models are listed in Appendix B.

3.6 Implementation Notes

All optimizations are carried out on in a Jupyter notebook environment using python 3.7.4 and TensorFlow 1.15. Python is run on a local computer with 16 GB RAM and an Intel i7-10510U CPU, 1.80 GHz.

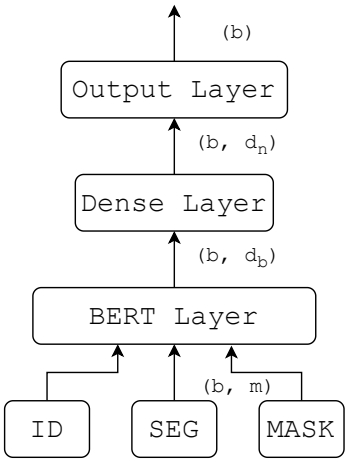


Figure 3.7: Schematic overview over the architecture of the BERT model. b is the batch size, m is the maximum sequence length, d_b is the BERT embedding dimension and d_n is the number of nodes in the dense layer.

Chapter 4
Results

In this section, the results on the benchmark dataset are briefly presented followed by the results on the financial data. The results from the movement of indices are displayed in Section 4.2. Finally, the results from the prediction of the ARMA-forecasts are presented in Section 4.3.

The performance is presented as accuracy and macro average F1-score on the test set for all models.

4.1 Benchmark Evaluation

The results on the IMDb dataset for benchmarking are presented in Table 4.1. While most of the models trained rather quickly, it is worth mentioning that BERT and the bidirectional LSTM both needed > 24 hours to be optimized on the validation set. BERT with a sigmoid output performed the best on this dataset with an accuracy of 91.6%, followed closely by BERT with a feed forward layer. There is however only a small gap to SVM, logistic regression and multi-layer perceptron with features generated by SBERT, trained at a fraction of the time needed for BERT. Another note is that the GloVe embeddings performed quite poorly in the pretrained setting, but had a large improvement when the embeddings were continuously trained.

4.2 Index Direction Predictions

This section presents the results from predicting the direction of an index on a daily basis. Two experiments have been made. The first one uses the news from day k to predict the movement from day $k-1$ to k. The second uses the news from day k to predict the movement from day k to $k+1$. Naturally, the first task should be easier since more information is available. It is however not trivial, since the targets are whether the median average of the

Model		Acc
TF-IDF	Random forest	85.6
	SVM	89.5
	Logistic regression	89.1
	MLP	89.2
GloVe (pretrained)	Random forest	80.7
	SVM	84.1
	Logistic regression	85.2
	MLP	84.1
GloVe (jointly trained)	MLP	88.2
	Bidirectional LSTM	90.1
Sentence-BERT	Random forest	87.1
	SVM	90.0
	Logistic regression	90.7
	MLP	90.1
BERT	Sigmoid	**91.6**
	MLP	91.0

Table 4.1: Performance of used methods on the IMDb dataset.

bid and ask rate over the full day has increased or decreased. It is dependent on the market movement of the whole day, while the news titles are published continuously throughout the day. Confusion matrices are displayed for a selection of models in Figures 4.1 and 4.2.

The dataset contains 1,846 days, partitioned into 960 samples for training, 240 sampled for validation and 646 samples for testing. The label distribution is presented in Table 4.2. The distribution was selected to be the same in the training and test set.

	0	**1**
1 year rate	0.61	0.39
3 year rate	0.54	0.46
S&P 500	0.55	0.45

Table 4.2: Label distribution for the three series in the index prediction task.

As seen in Table 4.3, a lot of the results for the one year rate are similar. This is due to the models simply predicting the most frequent class for every sample in the test set. Such a behavior was seen repeatedly, especially for the two treasury rates. The highest performing model on the test set for the three year treasury rate seen in Table 4.3 is SBERT with random forest, logistic regression and MLP. The predictions are however heavily skewed towards label 0 and can not be assumed to be significantly better than a random classifier. The best performance over the random baseline is achieved on the S&P index. Both the F1-score and the accuracy on the test set is improved compared to the baseline for some models.

The results in Table 4.4 generally show a little lower performance on the test set across all time series, compared to Table 4.3. A random classifier using the 'stratified' method achieves the best F1-score on the two treasury rates. The S&P series does however show an absolute

Model		1 year		3 year		S&P	
		F1	Acc	F1	Acc	F1	Acc
Random classifier	Most frequent	**0.56**	**68.9**	0.44	59.4	0.37	53.6
	Stratified	**0.56**	57.4	0.50	53.3	0.50	50.8
TF-IDF	Random forest	0.41	68.7	0.42	58.4	0.55	**59.6**
	SVM	0.41	**68.9**	0.40	59.4	0.51	57.3
	Logistic regression	0.41	**68.9**	0.43	58.4	0.55	58.0
	MLP	0.41	68.4	0.37	59.4	**0.56**	57.7
GloVe (pretrain)	Random forest	0.42	67.6	0.40	58.5	0.55	58.2
	SVM	0.41	**68.9**	0.37	59.4	0.35	53.6
	Logistic regression	0.41	68.6	0.40	59.0	0.53	57.3
	MLP	**0.56**	**68.9**	0.45	59.3	0.48	56.5
GloVe (jointly trained)	MLP	0.41	**68.9**	0.37	59.4	0.47	55.7
	Bidirectional LSTM	0.42	67.5	0.37	59.1	0.49	49.2
Sentence-BERT	Random forest	0.41	66.6	0.51	**60.4**	**0.56**	56.8
	SVM	0.41	**68.9**	0.37	59.4	0.54	57.0
	Logistic regression	0.53	67.2	**0.55**	59.4	**0.56**	57.1
	MLP	0.41	**68.9**	**0.55**	58.4	**0.56**	56.8
BERT	Sigmoid	0.41	**68.9**	0.38	59.4	0.48	50.2
	MLP	0.41	**68.9**	0.37	59.4	0.38	54.3

Table 4.3: Performance of used methods on classifying whether the price of an index has increased from day $k-1$ to k given the news titles from day k.

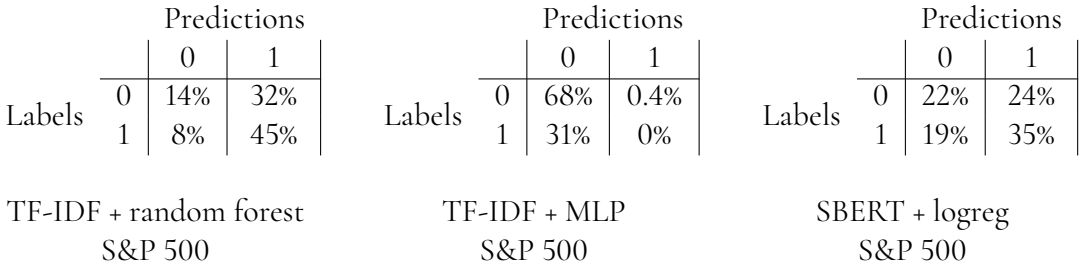


Figure 4.1: Confusion matrices for some models for the current day prediction task presented in Table 4.3. The metrics are from 646 test samples.

improvement over the random baseline. TF-IDF and random forest give the highest accuracy on the test set. The highest F1-score is achieved by TF-IDF + logistic regression together with pretrained Glove embeddings combined with logistic regression or a multilayer perceptron.

4.3 ARMA Direction Predictions

This section presents the results from the ARMA direction prediction task. The label is 0 if the ARMA prediction is higher than the actual outcome. Otherwise it is 1.

Model		1 year		3 year		S&P	
		F1	Acc	F1	Acc	F1	Acc
Random classifier	Most frequent	0.41	68.9	0.37	59.4	0.35	53.6
	Stratified	**0.50**	55.3	**0.49**	52.2	0.49	51.5
TF-IDF	Random forest	0.41	68.9	0.40	58.4	0.54	**59.1**
	SVM	0.41	68.9	0.38	58.4	0.49	56.3
	Logistic regression	0.41	68.9	0.40	55.9	**0.55**	58.2
	MLP	0.41	68.4	0.47	55.9	0.47	55.9
GloVe (pretrain)	Random forest	0.41	67.6	0.43	58.4	0.52	53.1
	SVM	0.41	68.9	0.37	59.4	0.35	53.6
	Logistic regression	0.41	68.9	0.44	59.6	**0.55**	58.2
	MLP	0.41	68.9	0.37	59.4	**0.55**	55.4
GloVe (jointly trained)	MLP	0.41	68.9	0.37	59.4	0.35	53.6
	Bidirectional LSTM	0.41	68.9	0.38	58.5	0.48	48.6
Sentence-BERT	Random forest	0.43	**69.2**	0.44	**59.9**	0.50	52.9
	SVM	0.41	68.9	0.37	59.4	0.36	54.0
	Logistic regression	0.47	66.6	0.51	56.3	0.53	54.3
	MLP	0.41	68.9	0.38	59.8	0.37	53.1
BERT	Sigmoid	0.41	68.9	0.40	58.7	0.47	53.3
	MLP	0.41	68.9	0.37	59.1	0.39	45.5

Table 4.4: Performance of used methods on classifying whether the price of an index has increased from day $k-1$ to k given the news titles from day $k-1$.

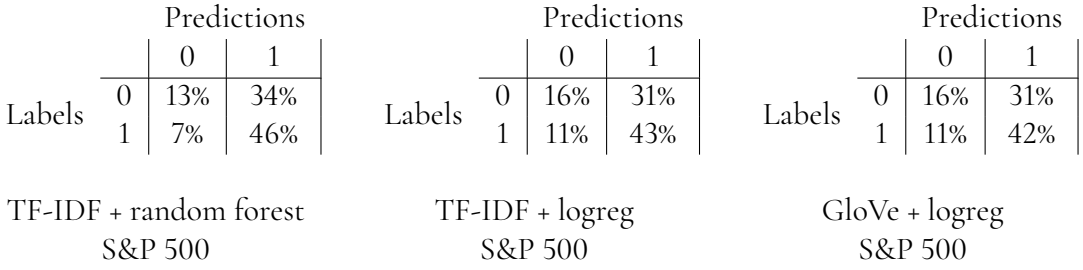


Figure 4.2: Confusion matrices for some models for the next day prediction task presented in Table 4.4. The metrics are from 646 test samples.

An ARMA(1,1) model is fitted to the time series for every date with available news. The accuracy of the prediction for the entire dataset is shown in Table 4.5. These are mainly shown to give an idea of the accuracy of the ARMA-model, but also gives a baseline for comparison with the task in Section 4.2.

The same structure as in the previous section is applied for this task. One evaluation uses the news from day k to predict the validity of the forecast for day k, see Table 4.7. The other evaluation concerns using news from $k-1$ to predict the validity of the ARMA-forecast for day k, see Table 4.8. Confusion matrices for the best performing model for every time series are displayed in Tables 4.3 and 4.4. The labels used in this task are slightly less skewed than

Index	Acc
1 year rate	65.9
3 year rate	56.0
S&P 500	47.6

Table 4.5: Accuracy of the ARMA-predictions.

the labels in Section 4.2, as seen in Table 4.6.

Index	0	1
1 year rate	0.61	0.39
3 year rate	0.54	0.46
S&P 500	0.55	0.45

Table 4.6: Label distribution for the three series in the ARMA prediction task.

Model		1 year		3 year		S&P	
		F1	Acc	F1	Acc	F1	Acc
Random Classifier	Most Frequent	0.38	61.1	0.35	53.8	0.35	55.0
	Stratified	0.47	54.4	0.51	50.4	0.48	52.9
TF-IDF	Random forest	0.46	62.0	0.50	53.3	0.46	55.7
	SVM	0.46	62.3	0.49	56.6	0.45	56.6
	Logistic regression	0.49	62.3	0.51	55.0	0.48	54.7
	MLP	0.38	61.1	0.54	54.9	0.45	56.4
GloVe (pretrain)	Random forest	0.52	**63.1**	0.53	55.8	0.49	53.6
	SVM	0.38	61.1	0.35	53.8	0.35	55.0
	Logistic regression	0.48	61.7	0.54	55.8	0.52	57.8
	MLP	0.52	62.2	0.35	53.8	0.50	55.0
GloVe (jointly trained)	MLP	0.38	61.1	0.35	53.8	0.36	54.6
	Bidirectional LSTM	0.39	60.9	0.50	51.5	0.48	56.4
Sentence-BERT	Random forest	0.55	62.3	0.53	54.6	0.59	60.9
	SVM	0.38	61.1	0.50	55.2	0.52	58.8
	Logistic regression	**0.57**	61.6	**0.57**	**57.4**	**0.62**	**62.6**
	MLP	0.51	62.5	**0.57**	56.9	0.55	59.4
BERT	Sigmoid	0.48	60.2	0.37	52.9	0.36	54.6
	MLP	0.44	61.2	0.48	52.4	0.43	53.6

Table 4.7: Performance of used methods on classifying whether the ARMA-prediction of an index on day $k-1$ was higher or lower than the outcome on day k, given the news titles from day k.

The results in this section are a bit more aligned than in the previous task. SBERT is the feature extraction technique that generally provides the best performance for this task. The performance over the random baseline is highest for the S&P data. SBERT combined with

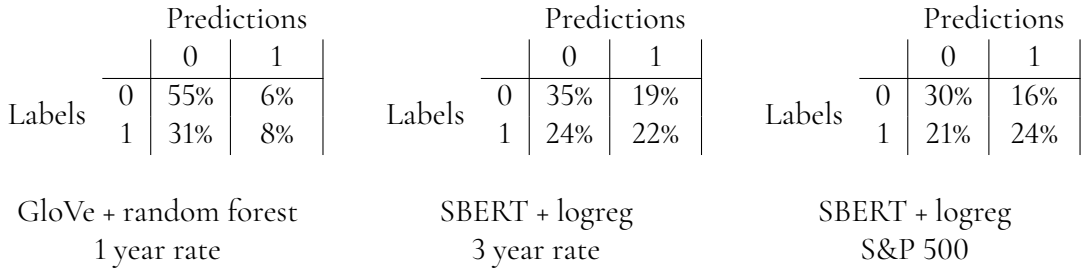


Figure 4.3: Confusion matrices for some models related to the results presented in Table 4.7. The metrics are from 646 test samples.

Model		1 year		3 year		S&P	
		F1	Acc	F1	Acc	F1	Acc
Random classifier	Most frequent	0.38	61.1	0.35	53.9	0.36	55.1
	Stratified	0.48	52.8	0.53	53.1	0.48	50.3
TF-IDF	Random forest	0.45	61.3	0.50	54.5	0.55	60.7
	SVM	0.42	61.5	0.43	53.7	0.50	58.0
	Logistic regression	0.47	61.0	0.52	56.7	0.55	60.2
	MLP	0.57	61.8	0.36	53.6	0.46	56.3
GloVe (pretrain)	Random forest	0.50	60.2	0.51	52.3	0.50	57.7
	SVM	0.38	61.1	0.35	53.9	0.36	55.1
	Logistic regression	0.52	62.2	0.54	55.7	0.48	55.6
	MLP	0.58	60.7	0.44	55.0	0.51	56.3
GloVe (jointly trained)	MLP	**0.60**	61.3	0.42	55.0	0.37	55.4
	Bidirectional LSTM	0.43	61.1	0.45	52.8	0.40	54.2
Sentence-BERT	Random forest	0.56	63.6	**0.55**	56.5	0.59	61.5
	SVM	0.38	61.1	0.53	**57.4**	0.51	59.3
	Logistic regression	0.58	63.2	0.54	55.0	**0.63**	**64.1**
	MLP	0.53	**63.8**	0.46	55.4	0.46	57.4
BERT	Sigmoid	0.48	60.9	0.35	53.5	0.50	54.3
	MLP	0.53	58.6	0.35	53.8	0.41	54.4

Table 4.8: Performance of used methods on classifying whether the ARMA-prediction of an index on day $k-1$ was higher or lower than the outcome on day k, given the news titles from day $k-1$.

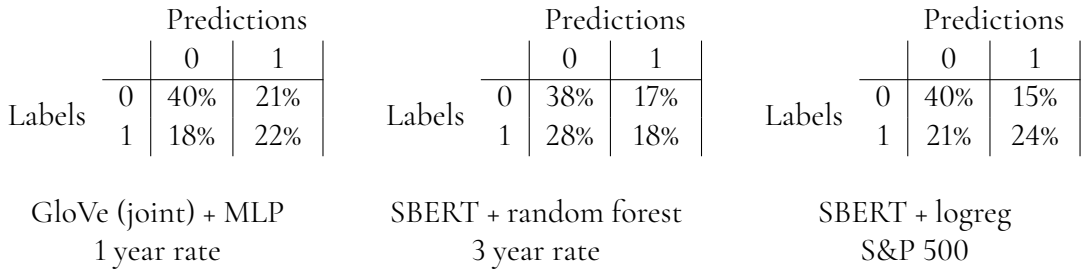


Figure 4.4: Confusion matrices for some models related to the results presented in Table 4.8. The metrics are from 646 test samples.

logistic regression provides the best results for S&P for both the current day predictions in Table 4.7 and the next day predictions in Table 4.8.

Chapter 5
Discussion

In this chapter, I discuss the findings in Chapter 4 and explore potential implications of the results.

Firstly, I evaluate the models in the IMDb task briefly. This is followed by evaluations of the models in relation to the financial tasks in Sections 5.2 and 5.3. A section on the used data follows. In the end of the chapter, I draw some general conclusions and formulate recommendations for further research.

5.1 Evaluation of IMDb performance

The results on the IMDb dataset showed that BERT with a sigmoid output was the most fitting model for the task with 91.6% accuracy. The BERT model with an MLP output performed lower, perhaps due to overfitting on the training data. SBERT with logistic regression was also quite close at a fraction of the training cost for BERT.

Another interesting note is that continued training of GloVe embeddings gave a significant improvement over static embeddings – from 84.1% to 88.2%.

The best model of the traditional, non-transformer based models was the Bidirectional LSTM at 90.1%. This model was also computationally demanding, needing more than 24 hours to converge.

It is safe to say that the collection of models are capable of capturing meaningful information from text data, at least for this type of text and task.

5.2 Index Direction Evaluation

This section discusses the results presented in Tables 4.3 and 4.4.

5.2.1 Model Evaluation

No model applied to the one year and three year rates gave any significant performance improvement over the random baselines. Reflections about the difference between time series is further elaborated in Section 5.4.

Several models did however show a predictive ability on the S&P 500 index, both for current day prediction and next day prediction. The simplest of the feature extraction techniques – TF-IDF – provided several results in the top segment for both tasks. Random forest and logistic regression performed well for both tasks, while SVM was less successful.

A few thoughts on why TF-IDF gave good results on the tasks follows.

Concatenated titles. The fact that the headlines are added up into a long string might be a benefit for TF-IDF compared to sequential models such as bidirectional LSTM and BERT. While the order of words in each sentence matters, the order of sentences should not matter too much. TF-IDF does not take order into account, and is therefore insensitive to whether the sentences are concatenated in a certain order.

Not very semantically demanding. The purpose of news titles are generally to distill the information in an article into a few words. It seems likely that titles are clear and succinct, which gives unambiguous formulations. Words like "outperform" and "excel" seem likely to be features that imply an upward trajectory, as opposed to "bankcruptcy" or "illicit".

It is also noteworthy that the BERT model was at least 100 times more computationally demanding than the models using TF-IDF or Sentence-BERT as features. Nevertheless, it did not give any impressive results.

Comparing the two tasks – predicting today's vs. tomorrows index movement – suggests that predicting the movement of today can be done with a slightly higher accuracy. There is however not a significant difference given the small test set. However, it is a reassuring result that the accuracy for current day prediction is slightly higher than for next day prediction.

This task can be compared with the foundings of Ding et al. (2014) who compiled the Reuters dataset I used. They scored 58.93% accuracy on predicting the direction of the S&P 500 index. This is to be compared with the best model I found on next day prediction – TF-IDF and random forest scoring an accuracy of 59.1%. Ding et al. (2014) only used 174 samples for testing, so it is not apparent that my results are an improvement.

5.2.2 Task Evaluation

The task of predicting the movement of financial indices in general is obviously highly popular, since there is an apparent possibility of profit. The effective market hypothesis introduced by Malkiel and Fama claims that the price of an asset reflects all available information regarding that asset. This would imply that analyzing historical news is pointless, since this information is already reflected in the price. An argument for why natural language processing is useful in the context of the effective market hypothesis is the sheer speed of the information retrieval. Even though the models' understanding of language is not as good as a humans, it is capable of processing a lot of new information quickly. By assessing if and how the new information has any implications for the price of an asset, the model can play a role

in finding a suitable price of an asset. It can also be useful for reducing costs, compared to an employee.

5.3 ARMA Direction Evaluation

This section discusses the results presented in Tables 4.7 and 4.8.

5.3.1 Model Evaluation

Similarly to the index direction task, the models reached the highest performance over the random baseline for the S&P 500 index. The model results are however less ambiguous for the ARMA direction task than for the index direction task. SBERT is the feature extraction method that gives the best performance. When paired with logistic regression, it seems to give decent results for all tasks and series. The confusion matrices in Figures 4.3 and 4.4 also show that the model seems to make reasonable predictions, compared to TF-IDF + MLP for S&P 500 shown in Table 4.1 which just predicts the most frequent label.

The accuracy of 64.1 % seen in Table 4.8 for SBERT + logistic regression is quite uncertain due to the small data set. This does however suggest that the news titles have some predictive power on the validity of the ARMA-prediction. A use case for this could be as a warning system for when a more sophisticated time series model seems to make a suspicious prediction. The NLP model can issue a warning when some probability threshold α is exceeded. For instance, using SBERT as feature extractor and logistic regression with a threshold value of $\alpha = 0.8$, an accuracy of 74 % is achieved. This implies the model only makes predictions if its more than 80 % certain. Given more data, this could possibly be a usable tool.

5.3.2 Task Evaluation

The purpose of this task was to evaluate if there is information in financial news headlines which improves the performance of a traditional time series model. While the short response is *yes*, a few clarifications should be made about the limitations of conclusions that can be drawn.

Small data set. The data set is small – 1,846 samples in total. This implies high variance in how well the test error approximates the actual error rate on unseen data. Furthermore, it increases the risk of overfitting to the training data. This is elaborated further in Section 5.4.

Limited time series model. The fitted ARMA(1,1)-model is rather simple and seldom used in practice for financial forecasting. Evaluating financial news in conjunction with a more sophisticated model with dependency structure between assets would be more realistic.

However, several models did perform above the random baseline both in terms of F1-score and accuracy.

5.4 Data Evaluation

This section discusses the used data sets and implications for how the results can be interpreted.

5.4.1 News Data

The news data was the limiting factor in terms of size. Even though the number of articles was rather large (~114,000), it was restricted by the fact that it only covered roughly 1,800 days.

Another uncertainty about the news data set is the way the headlines were collected. Some days have more news than others and it is unclear which criteria were used to gather the collection. Also, predictions on a daily basis are problematic in terms of size since one year only has 365 days. In order to get a substantial data set, one has to collect data quite a lot of years back. Chances are that the way news are written and the frequency with which they arrive are different now compared to 20 years ago. This implies the data samples are not identically distributed, which is problematic for generalization. Still, it is possible that general patterns occur, which is consistent over time.

Finally, the time which the news are gathered, from 2006-10-20 to 2013-11-19, was a quite turbulent time for the global economy. The global financial crisis took place 2007-2008 and is likely not representable for the relation between news and financial indices in general.

5.4.2 Financial Data

Collecting financial data was a lot easier than collecting news data. The quality of the data is generally better and it is easily available for long time spans.

The results on the S&P 500 index were more convincing than for the two treasury rates. This does make sense with regards to the used news data which mainly concerns financial news regarding large companies. The relation between news and treasury rates does not seem as apparent, which is in line with my findings.

It would be interesting to include a more specific time series for an asset or a stock. Using the price of oil or gold, as well as the stocks of large companies such as Google, Exxon or Apple could make for interesting results and discussion.

5.5 Conclusions

While it is difficult to make general and certain conclusions given the small size of the data set, the results are certainly intriguing. My findings confirm that news titles do have some predictive power on the S&P 500 index, as well as on the validity of ARMA-models fitted to both treasury rates and S&P 500. The performance is however most noticeable for S&P 500.

Another main takeaway is that the method for feature extraction seems highly problem-dependent. The BERT model performed the best on the IMDb data, whereas it performed poorly on all other tasks. However, SBERT produced good results for the all of the tasks. BERT and SBERT are largely similar, but obviously had big differences in performance. My

conclusion from this is that for an unknown problem, it is not always best with a large, computationally heavy model. In fact, TF-IDF with logistic regression performed better than BERT with a sigmoid output node in almost all tasks, except for the benchmarking task.

5.6 Future Work

A few notes on what aspects of this project that would be interesting to look into further. A lot of the uncertainty in this work comes down to the small data set, and the quality of the data. This is a natural starting point for further investigation. Some aspects which have repeatedly come to mind during the work with this report are listed below.

More data. This is one of the largest obstacles I experienced in this project – the lack of data. Using more data would allow for better understanding of how the model generalizes. The models might also overfit less and thus capture more true complex patterns in the training data.

Better data. As stated in Section 5.4.1, there are some uncertainties about the data and how it was collected. A more structured gathering of data with a higher quality assurance would make conclusions more robust.

Visualizing models. A deeper understanding of which parts in the text data that are important would be beneficial. This would give more intuition to how the model works, and could be used to validate the functionality of the model.

Better time series model. The ARMA(1,1) model used in this project is rather simple. A natural extension would be to introduce a more advanced model for time series modelling and see if including news data still improves the predictions.

Specific application. The task evaluated in this project was mainly chosen as indicators for if the data holds any predictive power on time series. The practical applications for the tasks are limited. It would be interesting to further investigate how this could be used in practice, and assess the validity of such an application.

References

Alammar, J. (2018). The illustrated transformer. `http://jalammar.github.io/illustrated-transformer/`.

Arora, A., Datta, A., and Ding, V. (2019). Using news titles and financial features to predict intraday movements of the DJIA. Project report.

Chollet, F. (2017). *Deep Learning with Python*. Manning Publications Company.

Chollet, F. et al. (2020). Keras. `https://keras.io`.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Ding, X., Zhang, Y., Liu, T., and Duan, J. (2014). Using structured events to predict stock price movement: An empirical investigation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1415–1425.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Hamilton, J. (1994). *Time series analysis*. Princeton University Press, Princeton, N.J.

Heston, S. L. and Sinha, N. R. (2017). News vs. sentiment: Predicting stock returns from news stories. *Financial Analysts Journal*, 73(3):67–83.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. (2019). Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*.

Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867.

Li, X., Xie, H., Chen, L., Wang, J., and Deng, X. (2014). News impact on stock price return via sentiment analysis. *Knowledge-Based Systems*, 69:14–23.

Lindgren, G. (2014). *Stationary stochastic processes for scientists and engineers*. CRC Press, Taylor & Francis Group, Boca Raton.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA. Association for Computational Linguistics.

Malkiel, B. G. and Fama, E. F. (1970). Efficient capital markets: A review of theory and empirical work. *The journal of Finance*, 25(2):383–417.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill Education.

Othan, D., Kilimci, Z. H., and Uysal, M. (2019). Financial sentiment analysis for predicting direction of stocks using bidirectional encoder representations from transformers (BERT) and deep learning models. In *International Conference on Innovative and Intelligent Technologies*.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

Reimers, N. and Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

Ting, K. M. (2017). *Precision and Recall*, pages 990–991. Springer US, Boston, MA.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Xie, Q., Dai, Z., Hovy, E. H., Luong, M., and Le, Q. V. (2019). Unsupervised data augmentation. *CoRR*, abs/1904.12848.

Xing, F. Z., Cambria, E., and Welsch, R. E. (2018). Natural language based financial forecasting: a survey. *Artificial Intelligence Review*, 50(1):49–73.

Appendices

Appendix A
Useful Links

- Link to my GitHub repository. `https://github.com/backmag/NLP-finance`
- The used BERT-model from TensorFlow Hub – `https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/2`
- Link to GitHub repository where the used Reuters dataset is hosted – `https://github.com/duynht/financial-news-dataset`
- Link to the GitHub repository *sentence-transformers*, where the SBERT-model I used is available – `https://github.com/UKPLab/sentence-transformers`
- Pre-trained GloVe word vectors provided by Pennington et al. – `https://nlp.stanford.edu/projects/glove/`

Appendix B
Hyperparameter Optimization

The tested range of hyperparameters and the optimal set found are listed in this appendix for a few models. The tuning was done for random forests and multilayer perceptrons. The purpose of this appendix is to give a detailed explanation of the methodology, rather than present all tested sets of hyperparameters.

B.1 Random Forest

Two hyperparameters were optimized for the random forest model, in the following ranges.

max_depth : None, 50, 100

n_estimators : 250, 500, 750, 1000

The best hyperparameters found through a grid search are displayed in Table B.1.

		max_depth	n_estimators
1 year rate	TF-IDF	None	250
	GloVe	50	1000
	SBERT	100	250
3 year rate	TF-IDF	100	250
	GloVe	None	1000
	SBERT	100	250
S&P 500	TF-IDF	50	500
	GloVe	100	500
	SBERT	None	500

Table B.1: Used hyperparameters for random forest model on current day predictions as presented in Table 4.3.

		max_depth	n_estimators
1 year rate	TF-IDF	None	500
	GloVe	50	250
	SBERT	50	250
3 year rate	TF-IDF	None	250
	GloVe	50	750
	SBERT	None	750
S&P 500	TF-IDF	50	500
	GloVe	None	250
	SBERT	50	750

Table B.2: Used hyperparameters for random forest model on next day predictions as presented in Table 4.4.

B.2 Multilayer Perceptron

For the regular feed-forward network, the hyperparameters in the following ranges were tested.

batch size : 8, 16, 32

nodes1 : 50, 100, 250, 500

nodes2 : 0[1], 50, 100, 250, 500

dropout rate (dr) : 0.1, 0.3, 0.5

learning rate (lr) : 1e-3, 1e-4, 1e-5

optimizer : RMSprop, Adam

Out of the combinations of parameters above, 50 sets are randomly selected without replacement. These are then trained using an early-stopping callback with patience 2. A validation split of 0.2 is used, and the set of hyperparameters which yields the highest validation accuracy is chosen.

[1] 0 implies only one hidden layer.

		batch size	nodes1	nodes2	dr	lr	optimizer
1 year rate	TF-IDF	32	50	0	0.3	1e-5	Adam
	GloVe	8	50	0	0.5	1e-5	Adam
	SBERT	32	500	100	0.3	1e-3	RMSprop
3 year rate	TF-IDF	32	100	100	0.1	1e-5	RMSprop
	GloVe	16	50	0	0.3	1e-5	RMSprop
	SBERT	32	50	0	0.1	1e-3	Adam
S&P 500	TF-IDF	16	250	50	0.5	1e-3	RMSprop
	GloVe	32	500	500	0.1	1e-3	Adam
	SBERT	32	250	250	0.5	1e-4	Adam

Table B.3: Used hyperparameters for multilayer perceptron on current day predictions as presented in Table 4.3.

		batch size	nodes1	nodes2	dr	lr	optimizer
1 year rate	TF-IDF	32	50	250	0.1	1e-5	Adam
	GloVe	8	50	0	0.5	1e-5	RMSprop
	SBERT	8	100	0	0.1	1e-5	RMSprop
3 year rate	TF-IDF	16	50	0	0.1	1e-3	RMSprop
	GloVe	8	500	100	0.1	1e-3	Adam
	SBERT	32	50	50	0.1	1e-4	RMSprop
S&P 500	TF-IDF	16	250	100	0.1	1e-4	RMSprop
	GloVe	8	500	0	0.3	1e-4	RMSprop
	SBERT	8	50	500	0.3	1e-5	RMSprop

Table B.4: Used hyperparameters for multilayer perceptron on next day predictions as presented in Table 4.4.

EXAMENSARBETE Forecasting Financial Indices from Financial News
STUDENT Gustaf Backman
HANDLEDARE Pierre Nugues (LTH), Edvard Sjögren (Kidbrooke Advisory)
EXAMINATOR Marcus Klang (LTH)

Slå börsen med morgontidningen?

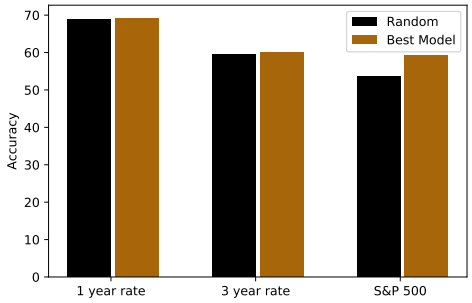
POPULÄRVETENSKAPLIG SAMMANFATTNING **Gustaf Backman**

Snabb och träffsäker hantering av data är av stort intresse inom många områden idag. De senaste åren har särskilt metoder inom automatisk språkhantering sett stora framsteg. I detta arbete visar jag hur deep learning och nyhetstitlar kan användas för att förutspå rörelsen på börsen med en träffsäkerhet på 59 %.

För att förstå hur en finansiell tillgång utvecklar sig över tid använder idag företag ofta tidsseriemodeller som bara tar hänsyn till tidigare värden av exempelvis en aktie eller ränta. Dessutom sitter någon och läser vad som händer i världen och tolkar hur det påverkar finansiella index. Detta tar tid och känns en aning ineffektivt, kanske kan det automatiseras och förbättras?

Mitt examensarbete provar olika sätt att tolka rubriker från finansiella nyheter i USA och undersöker hur de påverkar det amerikanska börsindexet S&P 500 samt två obligationsräntor. Jag har använt olika modeller för att göra om text till siffror. Både äldre varianter som i princip bara räknar vilka ord som finns i rubriker från en dag, samt modernare modeller såsom BERT, utvecklad av Google 2018. Dessa modeller för texttolkning kombineras sedan med någon typ av matematisk modell för att avgöra om ett index går upp eller ner, exempelvis logistisk regression.

Mina resultat visar att det går att förutspå om börsindexet S&P 500 går upp eller ner följande dag, åtminstone jämfört med att låta en bläckfisk gissa som i fotbolls-VM 2010. Vad gäller de två obligationsräntorna kan man dock lika gärna singla slant enligt mina resultat.



Träffsäkerhet på om ett index går upp eller ner kommande dag. Min modell är inte bättre än slumpen för de två räntorna, men bättre för S&P.

Den bästa modellen för S&P 500 var TF-IDF, en sorts normaliserad variant av ordräkning, tillsammans med en random forest-klassifierare. Båda dessa modeller har ganska många år på nacken, men verkar fungera bra till just denna applikation.

Dessa metoder kan exempelvis användas för att i realtid signalera när en nyhet verkar särskilt viktig och hur det påverkar köp/sälj-läget för en aktie eller annat index.