# Fact or Fiction Classification Machine Learning & NLP

By

Ibrahim Alhas

A master submitted to
King's College London
for the degree of
MASTER OF SCIENCE

Supervised by Josh Murphy
Faculty of Natural & Mathematical Sciences
Department of Informatics
King's College London
September 2020

## ACKNOWLEDGMENTS

I must chiefly acknowledge Josh Murphy for his supervision

I would like to thank my family and friends for their kind support

## ABSTRACT

The spread of misinformation is of great concern, with high societal impact and importance. The majority of online users are unaware that they may be exposed to fictitious, untruthful information. Historical and current research indicate that misinformation is on the rise, primarily due to the *Truth-Default Theory.* The theory states that people are susceptible to deception, especially on online news websites and social media platforms, because people tend to believe what they see is unconditionally truthful, without validating the integrity of the information. This dissertation had the primary aim of procuring a software algorithm solution that can classify factual from fictitious information. We built knowledge on existing research, and aimed to use *artificial neural networks.* Based on our literature and technical reviews, we proposed a set three solutions, all based on artificial neural network paradigms. We acquired a dataset of size $40,000$ instances, already labelled, albeit with a small class unbalance (approximately $2,000$ more instances for class fictitious). We implemented three distinct neural network architectures, namely *Convolutional*, *Recurrent*, and *LSTM*, which were designed to be nonidentical, for the benefit of exploration and experimentation. The ability to experiment allowed us to try different methodologies and designs, producing reliable and impressive results. In concluding our dissertation, we have procured three solutions for our problem, with each solution achieving a validated accuracy of 97%, 98%, and 99% respectively.

## DEDICATION

This dissertation is dedicated to those that posses the heart and desire to never stop learning

# Table of Contents

# List of Figures

# List of Tables

# Chapter One

# Introduction

In chapter one, we introduce the problem, motivation, and aims and objectives.

## 1.1 Problem & Motivation

When people are situated in a crisis, the trivial ability to acquire relevant information becomes an important necessity. However, not all circulating information may actually be factual, particularly on social media, because there is no automatic methodology established to deal with such a difficult task.

According to (Mian and Khan, 2020a), there is a global rise in the spread of misinformation (fictitious, or, unproven information) that has already damaged the scientific and public communities (Marlin, n.d.). Also, there seems to be an indication that certain (online) news agencies may deliberately create dramatic, fictitious headlines accompanied with unproven news to attract viewers, with no considerations of the consequences, such as causing panic amongst the public (Mian and Khan, 2020a). In addition, there have already been cases where some individuals have attempted to misinform the general public. One such scenario is the circulation of the so called "cures" or "treatments" that can supposedly heal an individual infected with the covid-19 virus. The Food and Drug Administration in the United States has warned against the lack of scientific evidence of such claims. A prime example is the fictitious news in circulation labelled as a "healing product" – which contains chlorine dioxide, a bleaching agent that if consumed, the infected patient would benefit from the supposed health effects, such as the antiviral, antimicrobial, and antibacterial properties as a cure. This claim has been rebuked by scientists as completely untrue, and the scientific community has advised the public to stay vigilant against further fictitious news (U.S.FDA, n.d.) & (Mian and Khan, 2020b). It is therefore beneficial for circulating information, such as news, (particularly on social media) be factual, and not be based upon fiction, nor contain bias. These requirements would benefit the public in terms of health, safety, increased awareness, and reduction or elimination of circulating misinformation, backed by copious amounts of research and evidence suggesting that fictitious, biased or untruthful information may have negative consequences on people (Marlin, n.d.).

The problem ultimately has societal importance. Misinformation could affect a person's ability to make well-informed and justifiable decisions. There is significant research suggesting that the average person in the public is susceptible to the *Truth-Default Theory*, or TDT. TDT, simply put, is as we communicate with others, we tend to operate on a default presumption that the other person is honest (Levine, 2014). This may imply that people could be '*truth biased*' such that they tend to believe others are telling the truth more often than they actually are. TDT simply makes it hard for the general public, especially on social media, to distinguish factual information from fiction because we are prone to believing what we observe or hear. Therefore, the solution to this problem will have positive impacts on society.

The project tries to address a very difficult and ongoing problem. First, the distinction between fact and fiction is hard to distinguish, especially on the Internet. In other words, how can each piece of information be labelled as factual or fictitious? Furthermore, one prime reason people fail to recognise fictitious information is because of TDF, as explained above. It is therefore difficult to envision how computer algorithms may deal with this problem. Second, the design, implementation, and evaluation of a software that can accurately classify fact from fiction will rely heavily on choosing the correct methodology, classification path, dataset, and evaluation metrics. Such tasks are not easy to implement, nor is there a perfect algorithm (as the *no lunch theorem* demonstrates). Thus, the trial-and-error methodology and creative thinking is crucial for tacking such problems as this.

Therefore, we propose a solution by employing an Artificial Neural Network algorithm that can (learn to) classify factual and fictitious information. The initial application of this project was Twitter, for the source of data for training and evaluation. We hope to extend this methodology to incorporate other areas, from finance to health, in future research.

## 1.2 Aims & Objectives

We hold the following aims and objectives of this project, bulletined below:

- We wish to thoroughly research ways in which circulating information can be classified into fact or fiction, based on a learning algorithm, with the help of Natural Language Processing (NLP) tools.
- We wish to use an unbiased, high-quality and sufficiently-sized dataset to produce an ethical and reliable solution for our problem.
- We wish to demonstrate the effectiveness of neural networks on our problem, namely LSTM and CNN, with high accuracy and generalization ability.
- We also wish to propose future research for this project, and to look for alternative solutions to the problem proposed in order to find potentially better solutions.

# Chapter Two

# Background Theories

In chapter two, we introduce the state-of-the-art literature and technical backgrounds. We assume that the reader has at least elementary mathematical knowledge of linear algebra, calculus and probability. We also assume the reader has at least basic knowledge of Machine Learning and Artificial Intelligence.

## 2.1 Literature

### 2.1.1 State-of-the-art Overview

The reviewed literature has provided great insight for this project. We identified algorithms that are already used for related tasks and problems, and that the theory of neural networks out-performing these standard algorithms seems to be evident, as we will see. In addition, the importance of a reliable and robust dataset has been established, which could increase learning performance for neural networks. Lastly, we acknowledged that the combination of existing models, such as *bag-of-words*, or *GloVe* word stemming from NLP, that could produce optimal performances. In addition, the use of sentiment analysis could further increase algorithm learning and performance capabilities.

We now show our findings from the reviewed literature in detail, below.

#### Fact or Fiction: Content Classification for Digital Libraries

In their paper (Finn, Kushmerick, and Smyth, 2001) stated that "the World-Wide-Web is a vast respiratory of information [. . . ], but much of it is hidden to the user". Their motivation was to research a robust process of extracting, classifying, and evaluating a classification algorithm for digital libraries was expressed. Their work attempted to create a naive Bayes model that could automatically classify news articles, where the article is either factual or fictitious, based on the information extracted from the news articles. Their work describes a system that can generalise well from news websites. However, it is important to note that their methodology only works on the extraction of information from URLs, where the algorithm attempts to automatically classify information in the URL by using an article database. The article database contains manually-engineered news articles, in which the classifier uses to base it's probabilistic classifications on.

What their research excelled in is that they identified many important theoretical questions and issues that are important for consideration when attempting to address such a difficult problem. The questions and issues they identified relevant for this project. For example, questions such as how do we, the engineers, classify information in practical means? In other words, how can we distinguish information that is factual or fictitious without the need for large amounts of resources or significant time. Furthermore, how do we explicitly enforce such a distinction in mathematical and programmatic methods? Moreover, their classifier was also generalisable, and this was justified with good results based on an unseen dataset (the use of $k$ cross-fold validation was highly useful). The favourable outcome of a generalised algorithm means that the classifier is flexible on other types of data, or on different platforms other than URLs. In addition, they recognised that some information extracted from the URL may be inconsistent or biased, and this problem was considered when building the classifier. For example, the decision to exclude images and links from the extracted data drastically improved the quality of the data by leaving out information that is considered irrelevant. This would reduce the occurrence of the *curse-of-dimensionality* problem (the curse of dimensionality problem is when there are vast amounts of data, the dimensions of the dataset increase exponentially (Keogh and Mueen, 2017)).

On the other hand, their research has potentially unidentified or unaddressed issues, with limitations. First, the database for the classifier was described as the "core of the system", implying that, without this database, their classification model would not work. Also, the classifier uses manually labelled data. This could imply bias or incorrect labels within the dataset. This is a limitation as the classifier requires hand-engineered instances, which are needed for the classification model to classify new, unseen data. We propose that in order to reduce the likelihood of bias in the dataset, a second engineer confirms there is no bias input.

To conclude, we propose that a more sophisticated algorithm be used instead, such as neural networks. This is because it may be more empirical to train a classifier that has the ability to learn. However, it is important to note that a neural network would require large amounts of data

#### DART: A Database of Arguments and Their Relations on Twitter

(Bosc, Cabrio, and Villata, 2016) have attempted to address the problem of argumentation in tweets. Their work has tried to identify certain types of tweets that are considered as arguments, and identify the relation (if any) between two tweets that are linked together, such as attacking or supporting. More so, their work tried to address the issues around extracting argumentation from (text) data, which includes the domain of facts vs fiction. Lastly, how to differentiate between fact vs fiction in text using the argumentation theory.

Their research has contributed substantially. For instance, their research formed a robust dataset of annotated tweets that range a multiple topics, such as Brexit. Their main contribution was the methodology they proposed in dealing with the limitations on what can be extracted from tweets, such as the 140-characters limit, the quality of the average tweet in terms of language (broken language, incomplete or unrecognisable), and the pairing of tweets for identifying their relations, such as an attacking or supporting tweets.

On the critical perspective, their research has failed to propose a working methodology for the classification of argumentative tweets. There is no methodology that can manually or automatically annotate tweets as supportive or attacking. Furthermore, it is apparent that their dataset was manually labelled, which could contain bias.

All in all, we acknowledged that incorporation of argumentation may lead to better performances because we may understand how factual or fictitious information may be spread, interpreted and used by Twitter users. We also acknowledge the issues of working with Twitter's 140-character tweet limit and whether to incorporate broken words, sentences, emoji symbols and numbers. Also, the source of the tweet could be taken into context, for example, if the tweet originated from a respected and acknowledged twitter account such as BBC News, then this could add more weight to the tweet in the direction that the tweet is probably factual, whereas if the tweet is from a source that has none or little legitimacy, would imply higher probability of fiction. Lastly, the automatic classification of new data based on real-world data seems to be a limitation of this research.

**Beyond Opinion Classification: Extracting Facts, Opinions and Experiences from Health Forums**

(Carrillo-de-Albornoz et al., 2019) have produced significant relevant research. In their article, they described a set of machine learning methods that can classify data from health forums, with high accuracy (over 80 per cent). Their motivation was to develop a model that could automatically classify factual, opinionated, or experience-related data, thus, to make health information more easily accessible for patients, professionals, and researchers. They implemented un/supervised algorithms, namely naive Bayes, linear regression, and decision trees, to classify a set of 3000 health posts in health forums, which were manually annotated as fact, opinion, or experience. We focus on the support vector machines because it produced the highest accuracy out of all algorithms.

The most interesting aspect of their research was that they used the bag-of-words model, which is typically used for natural language processing tasks. Whether to implement NLP tools is an important decision to make when it comes to deciding how the algorithms will understand what words project in semantic definition. Furthermore, the use of word embedding (the measure of similarity between two words) has been used in order to boost the classifiers' ability to assess the context of the data in relation to other words in the corpus (a set of information or data, compiled).

Critically, their classifiers were domain-specific, meaning the classifier would have high accuracy classifying health-related data, but probably low accuracy on other topics, such as news data. In other words, their classifiers lack generalisability on unseen or different data, such as Twitter data. Since Twitter data evidently has more complicated semantic and lexical structures of information, such as emojis, symbols and sentence formatting, a more appropriate word embedding (or stemming) model could have been used, such as GloVe, a model for distributed word representation (Pennington, Socher, and Manning, 2014a). In addition, a variety of machine learning algorithms have been used in this research, excluding neural networks, which may signify little or insufficient ability for learning. We propose that the use of neural networks may be more practical because neural networks can learn to find patterns in data, where other un/supervised algorithms may fail to do. The use of neural networks may lead to increased accuracy and generalisation abilities.

In conclusion, the use of the bag-of-words model in this research has been demonstrated to improve the accuracy of the support vector machine algorithm, instead in conjunction with neural networks for this project for the main benefits of learning ability, and generalisability on different domains, for increasing performances. However, we suggest that the GloVe word embedding model was more appropriate because of the optimal computations of word similarities may result in more accurate classification.

### Combining Machine Learning with Knowledge Engineering to Detect Fake News in Social Networks

(Hinkelmann, Ahmed, and Corradini, 2019) have proposed a combination of machine learning algorithms, and incorporating *Knowledge Engineering* to detect fake news on social media. Their main proposal was combining knowledge discovery with machine learning, which in simple terms, was the attempt at creating rules that tried to mimic the thought processes of engineers These generated rules would then be added to the classifier. The significance of their methodology was that a set of rules were created, then embedded on machine learning algorithms to include the engineers' thought process into the classification model itself, which could then potentially be robust at solving complex problems in a variety of domains, without external intervention. Briefly, a synthetic dataset is created manually with arbitrary data. Next, the dataset is fed into a machine learning algorithm for training (with poor performance at this stage). During training, the data is analysed and assessed on the probability of factual or fictitious news with the following set of meta-data: source of data (i.e. social media, which can be argued that the source of the data may be indicative of bias, so that the classifier could perform better by selecting only legitimate sources, thus producing biased results), author, and topic. The training is saved for the knowledge engineering process. An important aspect of this research is that a proposal of a *"stance detection"* function is suggested. The stance detection assesses the *"confidence"* of the algorithm during training (whether the algorithm is sure of the output it computed). As mentioned prior, the training result is inputted into a *"combination function,"* which holds all the output computations. Intuitively, the combination function is then used to feed the knowledge engineering process so that a set of rules are generated. In other words, the algorithm tries to mimic the human conclusion-processes by extracting meaningful information from data. The significance of this is that the algorithm produces a library of problem-solving rules, and a modest base of collateral knowledge to solve problems without the intervention of an engineer.

Limitation-wise, the algorithms depend on the set of rules created via knowledge engineering. If the rules generated are not interpretable or relevant, then the classifier will perform poorly as a result of poorly designed rules. Also, the algorithm does not handle real-world data, which could lead to biased or unrealistic rules during knowledge engineering. In addition, the collateral knowledge could be impacted negatively if the dataset is insufficient in size or quality.

As for last remarks, their research have introduced the potential combining machine learning and knowledge discovery. We propose that real-world data was used instead of a synthetic, manually created one. The potential is that the classifier may find realistic and relevant complex patterns where the classifier may be unable to identify, due to fake data (due to a lack of good features or algorithm parameters).

### A Short Message Classification Algorithm for Tweet Classification

(Selvaperumal and Suruliandi, 2014) created a method that makes use of tweet features, such as the number of times a tweet was retweeted, and main text. The performance of this method was compared to the classic text classification algorithms, i.e support vector machines and naive Bayes. Their motivation was to improve the accuracy of the method via the use of tweet features for trending tweets, based on tweet popularity (retweets).

One of the most significant decisions they took was the collection of unbiased data from Twitter (via Twitter's data API). The tweets were extracted via URL. Once the dataset was collected, the pre-processing stage, where symbols, use of stop words (words such as "the", "a", "and", and so on, that are filtered out before training in order to improve data quality), and non-English words were removed. However, Pre-processing did not address the possibility of broken words or words with numbers as they were left intact because of the difficulty in detecting and overcoming such occurrences. During evaluation, the method had an increased classification accuracy of 10.2%, compared to the conventional classifiers, such as naive Bayes.

The limitations of their research are that their dataset was small - with approximately five hundred instances. We can subsume that the results were provisional rather than established because conventional algorithms have more data, which increases reliability and consistency of the results. Moreover, during the pre-processing stage, the decision to implement stop words may have altered the results, as some stop words may have relevant or interesting information in a tweet. For example, the more contextual information regarding a tweet, the more justification for the classification.

Their research has allowed us to appreciate the implementation of stop words. Furthermore, the use of context (i.e. tweet features) was demonstrated to improve model accuracy, compared to conventional algorithms, thus, justifying the aims of their research.

## 2.2 Technical

We now wish to explain technicalities of this project, as well as their mathematical foundations (where necessary).

### 2.2.1 What is Machine Learning?

*Machine Learning* (ML) is a subfield of *Artificial Intelligence* (AI), with roots in statistics, pattern recognition, signal processing, natural language processing, and data mining. The goal of machine learning is to enable computers the ability to learn from data, without explicit programming (Alpaydin, 2020). A machine learning model observes data to identify patterns, then uses the patterns to make predictions. These models come in handy when we cannot solve a problem for any apparent reason, such as logistical or high complexity. It is important to note that machine learning algorithms usually require large amounts of data, where data can be numbers, words, images or even interaction (Alpaydin, 2020). Machine learning has three distinct paradigms which we will describe briefly.

*Supervised learning* is the method of supervising the model to predict an outcome from the given input, where input is a dataset with input/output pairs. The goal is to produce accurate predictions on never-before-seen data. There are two distinct branches of algorithms that are classed as supervised learning, and they are *classification* and *regression*. Regression is the prediction of continuous values, with a definition of a function with input variables. In classification, we try to predict a class label. A class label is simply a target/output that is predefined. The model classifies data based on the labels via error-correction. Classification usually falls under binary or multiclass, where binary is two classes, and multiclass is more than two classes. For example, the classification of fact or fiction is binary; classification of spoken languages is multiclass. There are many algorithms for supervised learning, such as neural networks and decision trees. In this project, we utilise supervised learning.

In *Unsupervised learning*, there is no supervision. The model is fed data that does not have input/output pairs (predicted/actual), but only input. The purpose of unsupervised learning is to identify hidden patterns in data that may not be be found via supervised learning, where the unsupervised model is simply exploring the data for patterns and knowledge (Alpaydin, 2020). Unsupervised learning is usually implemented to understand the data as a pre-processing technique, which can be useful for learning a new representation of the data, and also for increasing accuracy and reducing error rates.

*Reinforcement learning* (RL) differs from un/supervised learning paradigms because there no input/output data pairs. Reinforcement learning focuses on exploration (of unknown territory) and exploitation (of known territory). Briefly speaking, the RL model is punished or rewarded based on its performance in an environment (i.e. virtual simulation). If the model makes an error, it is punished; if it does what it was intended to do, then it is rewarded. This alerts the model semantically, that 'this is wrong, don't do it again' or 'this is correct, keep doing it'. Readers interested in more detail can refer to (Alpaydin, 2020).

### 2.2.2 Mathematical Theories

We now explain the mathematics of concepts and processes common in this project.

#### Gradient Descent

*Gradient Descent* is a powerful optimisation algorithm commonly used in ML (among other disciplines) to update the parameters of ML models by iteratively moving in the direction of the local or global minima (the optimal solution) (Amari, 1993a). Because the error surface can be complex with hills and valleys, or even an increasing dimensionality, it is therefore important to use optimisation techniques that will find the minima of an error function. In neural networks, the objective is to minimise the error during training. Therefore, the objective function can be viewed as a loss function. For readers interested in pseudo-code of gradient descent, see Appendix B.

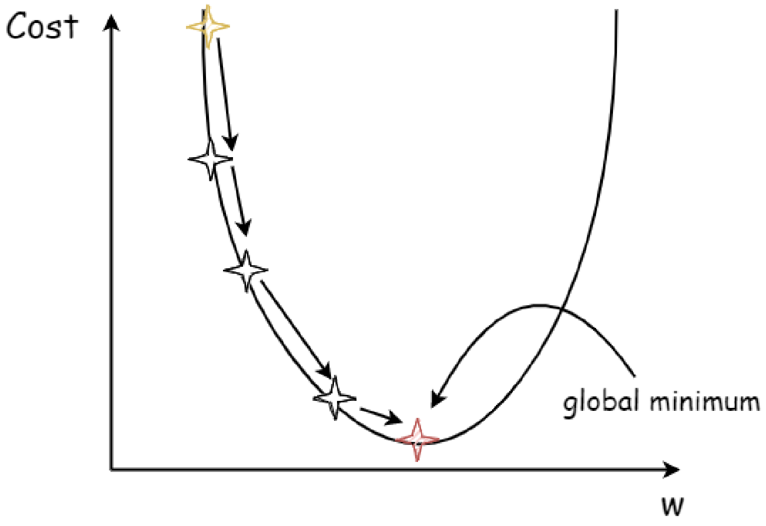


**Figure 2.1** An simple illustration on Gradient Descent. We start at an arbitrary location (yellow star), and reduce the error by moving closer to the global minimum by taking gradients. The global minimum (red star) is the optimal cost.

One problem concerning gradient descent is the *vanishing gradient* problem. Any algorithms that employ gradient descent can be affected with the problem of vanishing gradients. Simply put, the use of multiple activation functions in multiple layers of hidden layers results in the gradients of the loss function to approach zero. This means the neural network model reaches a point where no learning occurs. This is a significant problem, as when a neural network contains multiple layers, there could be an increase in the likelihood of vanishing gradients occurring. However, a solution to this problem is to use the ReLu activation function which eliminates this problem as ReLu does not cause small derivatives, which means that the gradients are also not small – which results in significant change in derivatives, allowing the model to learn (Grosse, 2017).

Intuitively, a gradient is a vector that stores the partial derivatives of multivariate functions. Partial derivatives are used to calculate the slopes at specific points in curvature for multivariate independent functions. Gradient descent will find the parameter $\boldsymbol{w}$ by sequentially taking steps that is proportional to the negative of the gradient. We briefly summarise, mathematically, the concept of gradient descent in neural networks.

**Loss Function**

There are many loss functions to choose from. Namely, *Cross Entropy* measures the performance of the classification model by measuring the differences of two probability distributions for random variables or events (Nasr, Badr, and Joun, 2002), where the output is between $[0, 1]$. An alternative is *Mean Squared Error*, or, MSE. MSE computes the averages of the squared errors. In addition, the output is always positive (Y. Lu, 2019). MSE is used for regression tasks, and therefore, is not suitable for a classification problem. In contrast, cross entropy is used for classification problems. Cross entropy creates a wide decision boundary between the classes, which typically results in less error computed. Cross entropy works by computing the Euclidean ($L2$ norm) distance between the classes. Cross entropy would benefit this project as a wider boundary is optimal for least error computed by the neural network. We illustrate mathematically an example of a loss function, below.

Using loss function MSE, the parameter $w$ is updated to minimise error:

$$w(k+1) = w(k) - \Delta w(k) \tag{2.1}$$

where *w(k)* is the value of $w$ at $k$ iteration. And $\Delta w$ is:

$$\Delta w(k) = \alpha \frac{\partial \xi}{\partial w} \tag{2.2}$$

where $\alpha$ (alpha) is the learning rate, which states how much change is to be taken along the gradient. The $\partial \xi$ over $\partial w$ is the gradient of the loss function $\xi$, proportional to weight $\boldsymbol{w}$. We can derive the gradient for every sample $i$ with the *chain rule*:

$$\frac{\partial \xi_i}{\partial w} = \frac{\partial \xi_i}{\partial y_i} \frac{\partial y_i}{\partial w} \tag{2.3}$$

Since $\xi_i$ is the squared error loss , we can show $\partial \xi_i$ over $\partial y_i$ as:

$$\frac{\partial \xi_i}{\partial y_i} = \frac{\partial (t_i - y_i)^2}{\partial y_i} = -2(t_i - y_i) = 2(y_i - t_i) \tag{2.4}$$

We know that $x_i \times w$ is simply $y_i$, and we can deduce $\partial y_i$ over $\partial w$ to simply:

$$\frac{\partial y_i}{\partial w} = \frac{\partial (x_i \times w)}{\partial w} = x_i \tag{2.5}$$

Lastly, the function update $\Delta w$ for sample $i$ is:

$$\Delta w = \alpha \times \frac{\partial \xi_i}{\partial w} = \alpha \times 2x_i(y_i - t_i) \tag{2.6}$$

For *batch* gradient descent, the gradients are simply summed for each sample $i$:

$$\Delta w = \alpha \times 2 \times \frac{1}{N} \sum_{i=1}^{n} x_i(y_i - t_i) \tag{2.7}$$

Initially, gradient descent is initialised to random-value parameters. The optimisation is implemented after the error is calculated; then the parameters are updated to minimise the error (Vidal et al., 2017).

Let us now describe the driving force of optimisation in neural networks.

### Backpropagation Algorithm

*Backpropagation* allows for the computation of partial derivatives $\partial c$ over $dw$, and $\partial C$ over $\partial b$ of the cost function $C$. By computing the loss in the previous iteration or epoch (i.e. one cycle through the entire dataset), the weights and biases are tuned to generate less error in the next iteration or epoch. Ultimately, backpropagation lets us understand the impact of how changing the weights and biases in a neural network manipulates the loss function (Ruder, 2016). The fundamental requirements of backpropagation are a dataset, compatible neural network architecture (i.e. feed-forward), and a loss function.

There are two forms of backpropagation: *stochastic* (SGD) and *batch* (BGD). Stochastic is an iterative method where the derivatives are calculated after each sample. Batch refers to an epoch of training; then the parameters are tuned. Both methods have their own advantages and disadvantages. Namely, stochastic converges faster compared to batch because the derivatives and parameters are computed and updated after each sample. On the other hand, the batch version computes the error after going through each sample in the dataset (which takes significant computational resources and time) (Amari, 1993b). Furthermore, SGD is less computationally expensive because each sample is computed individually, compared to BGD where the whole dataset is computed at once. However, BGD allows for more stable convergence because parameter updates are computationally efficient, and less updates are computed. A hybrid of SGD and BGD was created to overcome the limitations and cons of these versions, called *mini-batch* gradient descent. Mini-batch combines the computationally efficiency of BGD and the speed of SGD to optimise a neural network. For each iteration, a smaller batch of samples are computed instead of the full batch. This method reduces computational costs of the stochastic method and decreases the likelihood that the model will overshoot or perform premature convergence that may occur for the batch method, which enables for a robust, well balanced, and computationally non-intensive optimisation method.

We describe backpropagation mathematically, in backwards (since the actual algorithm works backwards) (H.K.Lam, 2020). The **training error** for any single instance is:

$$J(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{c} (t_k - z_k)^2 = \frac{1}{2} ||\mathbf{t} \text{ - } \mathbf{z}||^2 \tag{2.8}$$

where $t_k$ is the $k^{th}$ target output; and $z_k$ is the $k^{th}$ actual computed output, and $k = 1...c$; $\mathbf{w}$ is the representation of weights in the network. $||\cdot||$ is the Euclidean norm operator of desired output $\mathbf{t}$ minus actual output $\mathbf{z}$, squared, i.e. $\sqrt{a_1^2 + a_2^2 + a_3^2 + \cdots + a_n^2}$. The weights are updated to minimise the error:

$$w(l+1) = w(m) + \Delta\mathbf{w}(m) \tag{2.9}$$

where $\Delta\mathbf{w}(m)$ is:

$$\Delta\mathbf{w}(m) = -\alpha\frac{\partial J(w(m))}{\partial w(m)} \tag{2.10}$$

where $m$ represents the $k^{th}$ iteration; and $\alpha$ is the learning rate, which is $> 0$. To find the error on the **hidden-to-output** weights for one iteration:

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = -\xi_k\frac{\partial net_k}{\partial w_{kj}} \tag{2.11}$$

where $net_k$ is the net of the output of the weights multiplied by $x$, at $k^{th}$ iteration. We can also define the *node sensitivity* (sensitivity describes how the error changes with the activation function of the node's $net_k$):

$$\xi_k = \frac{\partial J}{\partial net_k} \tag{2.12}$$

Furthermore, we express $\xi_k$ as:

$$\xi_k = \frac{\partial J}{\partial net_k} = \frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = (t_k - z_k)\frac{\partial f(net_k)}{\partial net_k} = (t_k - z_k)f'(net_k) \tag{2.13}$$

We can define $net_k$ as:

$$net_k = \sum_{i=1}^{n_H} y_i w_{ki} + w_{k0} = \sum_{i=0}^{n_H} y_i w_{ki} = W_k^T Y \tag{2.14}$$

where $w_k0$ is the bias, usually equal to 1. So, in summary, the updates for weights for hidden-to-output is:

$$\Delta w_{ki} = -\alpha\frac{\partial J}{\partial w_{ki}} = \alpha\xi_k y_i = \alpha(t_k - z_k)f'(net_k)y_i \tag{2.15}$$

which is what we calculated as the error for the current iteration. The new weights updated for the next iteration $k$ will be:

$$w_{ki}(m+1) = w_{ki}(m) + \Delta w_{ki}(m) \tag{2.16}$$

where $i = 0$ for the bias; $1,...,n_h$; $k = 1,...,c$; $y_0 = 1$. Now, we compute the error for **input-to-hidden** neurons, at the $k^{th}$ iteration:

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} \tag{2.17}$$

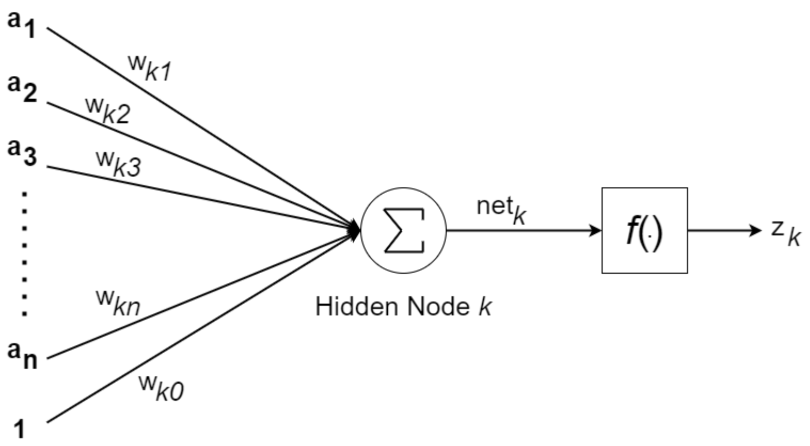


**Figure 2.2** The inputs $a_n$ with their corresponding weights $w_{kn}$. $net_k$ is simply the dot product of $\mathbf{wx}$ in hidden node $k$. $net_k$ is computed in the activation function.

where $J$ and $y_j$ respectively are:

$$J = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2, y_j = f(net_j) \tag{2.18}$$

and $net_j$ and $w_{ji}$ are:

$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0}, w_{ji} = \frac{\partial net_j}{\partial w_{ji}} = x_i \tag{2.19}$$

where $x_0$ is simply the bias set at 1. Therefore:

$$\frac{\partial J}{\partial y_j} = -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial y_j} \tag{2.20}$$

and:

$$\frac{\partial J}{\partial w_{ji}} = -\sum_{k=1}^{c} \xi w_{kj} f'(net_j) x_i \tag{2.21}$$

where $\xi$ is:

$$\xi = \sum_{k=1}^{c}(t_k - z_k) f'(net_k) w_{kj} \tag{2.22}$$

Therefore, the conclusive expression for error in the input-to-hidden neurons at $k^{th}$ iteration is:

$$\Delta w_{ij} = -\alpha\frac{\partial J}{\partial w_{ij}} = \alpha\xi x_i = \alpha f'(net_j)\left[\sum_{k=1}^{c} w_{kj}\xi_k\right] x_i \tag{2.23}$$

where $i$ is equal to 0; $k$ = 1,...,c. And finally, the updated weight ($w_{ji}$) at the next iteration ($k+1$) is:

$$w_{ji}(m+1) = w_{ji}(m) + \Delta w_{ji}(m) \tag{2.24}$$

**Activation Functions**

An *activation function* is a mathematical function that maps an input into an output. In machine learning, there are many activation functions to choose from, each with their own significance, limitations and uses. If activation functions were not used in neural networks, the models would be very simple as no learning of complex patterns from data would take place. Activation functions determine the output of the neural network, their accuracy, and also their computational capacity and efficiency during training. Lastly, the convergence of the model depends heavily on the type of activation function used (Agostinelli et al., 2014).

The *step function* is an elementary activation function. A step function determines whether the neuron is activated based on the input value. This is often referred to as the *threshold* function. If the input of a neuron is above the threshold, the neuron is activated, otherwise it is not.

The mathematical expression of a step (threshold) function is:

$$\omega(x) = \begin{cases} 1 & \text{if } x > threshold \\ 0 & otherwise \end{cases} \tag{2.25}$$

where 1 = neuron activation.

Another activation function is the *sigmoid.* Sigmoid outputs ranges between $[0, 1]$; and it is not zero-centred. By this we mean the value of the function revolves around 0, which means convergence is quicker, but with increased computation cost. This is important as neural networks with $n$ number of layers, where $n$ is $> 2$ - meaning convergence is quicker.

A more popular activation function is *ReLu* (Rectified Linear). ReLu approximates the biological motivations and mathematical justifications in non-linear form, which also solves the vanishing gradient problem. Compared to sigmoid, ReLu is not restricted to $[0, 1]$, and saturates on a single direction, causing resistance to vanishing gradients. In addition, since the derivatives of ReLu are equal to 1, a neural network does not require additional time for computing error during training because of this constant (Agarap, 2018). However, ReLu is prone to the "*dying ReLu*" problem, where the input approaches negative or zero, which is a detrimental problem as backpropagation is impossible without gradients. One solution to this problem is by implementing a variant of ReLu, called *Leaky ReLu* (L. Lu et al., 2019). Leaky ReLu avoids the dying ReLu problem because of positive slopes in the negative region, which enables the flow of gradients for backpropagation. Essentially, Leaky ReLu prevents the dying of neurons because the range of the function is extended to negative regions.

The mathematical expressions for ReLu and Leaky Relu respectively are:

$$\gamma(\mathbf{x}) = max(0, \mathbf{x}), \zeta(\mathbf{x}) = max(\xi x, \mathbf{x}) \tag{2.26}$$

where $\mathbf{x}$ = input, and $\xi$ is $< 1$.

We now wish to introduce the inspiration for the computational neuron.

### 2.2.3 The Biological Neuron

A *neuron* is a processing cell found in the mammalian brain. Neurons are electrically excitable cells (meaning, the cell can get 'activated'), and have the primary function of information processing and transmission. A neuron is composed of various components, such as the dendrites, nucleus, axon, and the cell-body. The brain consists of billions of neurons that are densely interconnected. Neurons transmit information to other cells, such as nerve cells, muscles, or gland cells (A. K. Jain, Mao, and Mohiuddin, 1996a). The biological neuron has inspired for the mathematical foundations of modern artificial neurons networks.

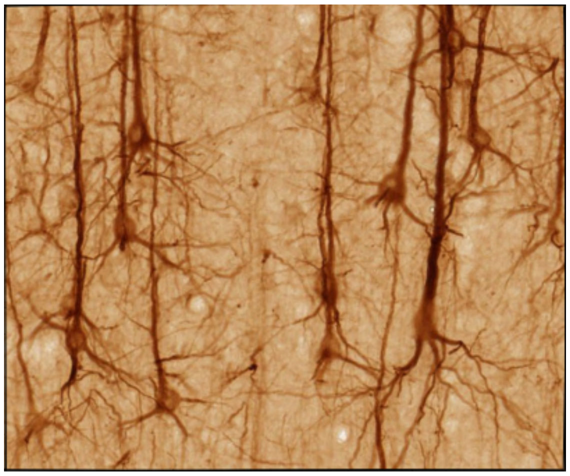


**Figure 2.3** Biological neurons captured via microscope in the cerebral cortex of the Brain. The dendrites, axons, and cell-body of each neuron are clearly visible, as well as their interconnections. Image courtesy of (BrainMaps.org, n.d.).

### 2.2.4 The Perceptron Model

A *perceptron* is the backbone of any artificial neural network. It is one of the simplest supervised learning algorithms developed. (Rosenblatt, 1958) showed that the perceptron was designed to illustrate the properties of intelligent systems, particularly on how an intelligent being senses information, and how information is stored. Furthermore, Rosenblatt theorised a probabilistic model on which numerical information is processed. This model is based on the biological neural network. One of the most important concepts of the perceptron is the *threshold*, in which a neuron must meet if it is to be activated. The theory that a biological neuron could be mimicked mathematically has led to the development of complex learning systems, such as *Deep Learning*, and it all started with the perceptron model. Moreover, (McCulloch and Pitts, 1943) work pioneered the background theories necessary for building artificial neural networks, after Rosenblatt. They demonstrated how biological neurons produce highly sophisticated patterns, and how neurons can be mimicked computationally (Hayman, 1999).

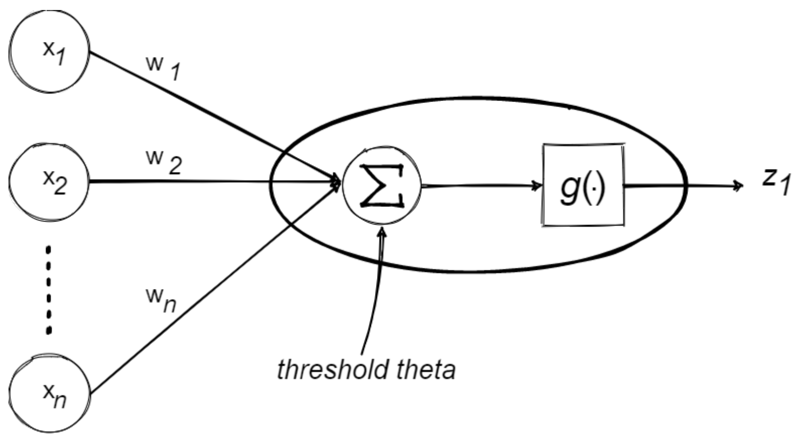


**Figure 2.4** The early Perceptron, proposed by McCulloch & Pitts in their work in 1943. Figure 2.5 depicts a single-layer (single perceptron) perceptron model, comprising of $n$ input features,$n$ weights, summation function, and output $z$. The activation function is undefined, but can be any of the functions we described.

Conversely, the *input* is numeric points (data), that is fed into the *input* layer. The *weights* are the parameters that transform the input data in the *hidden* layer(s) (the multiplication of $xw$). In other words, weights determine the significance of an input in the neural network. Neurons in the hidden layer are what makes learning possible. The hidden neurons have no connection to the input layer, only to other, deeper hidden layers. The summation function produces the dot product of $xw$; the activation function then determines if the value of the neuron is above a threshold value. Finally, the *output* is a *class* in which the neural network computed. We can think of the inputs as the dendrites, the weights as the synapses, and the threshold function can be approximated to the soma of the neuron (where the activation is determined), in biological terms (A. K. Jain, Mao, and Mohiuddin, 1996a).

However, the shortcomings of the perceptron model was soon realised. Namely, because the perceptron is a single-layer model, the ability for complex learning could not be achieved. For example, the perceptron fails to solve any XOR problems because the perceptron computes linear output computations, where XOR problems are not linearly-separable. However, to solve these limitations, the *Multi-Layer-Perceptron* model was conceived, which produces non-linear outputs (Mühlenbein, 1990).

### The Perceptron Convergence Theorem

The *Perceptron Convergence Theorem* states that for a perceptron algorithm (i.e. single or multi-layer), any two classes can be separated if they are *linearly separable.* A decision boundary is computed if the algorithm can successfully divide the classes into their respective class regions. In technical terms, there exists a weight vector, $w$, that the perceptron model will classify all samples $x$ correctly in a finite number of iterations. Because the perceptron is the backbone of any computational neuron, the theorem applies to artificial neural networks.

We intuitively describe the theorem below.

**Formally:**

$$\textit{Let there be two } \textbf{\textit{linearly-separable}} \textit{ classes } \mathcal{A} \textit{ and } \mathcal{B}. \tag{2.27}$$

$$\textit{Let there also be a dataset } \mathcal{D}, \textit{ which is divided into two subsets, one for each class.} \tag{2.28}$$

$$\textit{Given the dataset, train the perceptron until a weight vector } \vec{\mathbf{w}} \textit{ is found, where:} \tag{2.29}$$

$$\vec{\mathbf{w}}^T \;\; > 0 \textit{ for all input } \mathcal{P} \; \in \; \textit{Class 1} \tag{2.30}$$

$$\vec{\mathbf{w}}^T \;\; < 0 \textit{ for all input } \mathcal{P} \; \in \; \textit{Class 2} \tag{2.31}$$

The perceptron convergence theorem states that for all inputs $P$ that are $>$ zero, they are classified as class 1, and conversely, for all inputs $P$ that are $<$ zero, they are classified as class 2. Thus, the goal of the perceptron is to find the weights that satisfies the above criteria (2.30 & 2.31) by implementing an error-correction method in a finite amount of iterations, as long as the two classes are linearly separable (Murphy, Gray, and Stewart, 2017).

We have described the backbone of any neural network. We can now delve into the details of artificial neural networks.

### 2.2.5 Artificial Neural Networks

An *artificial neural network*, or simply, neural networks (NNs), summarised by (Priddy and Keller, 2005), is a mathematical model that aims to mimic the structure and functionalities of a biological neural network. Neural networks can be viewed mathematically as a function that takes an input and produces an output, just like its biological counterpart. Neural networks have been used to solve (and continue to do so) a wide variety of problems in diverse fields, such as pattern recognition, optimisation, prediction and natural language processing, to name a few. Neural networks can learn *automatically* from examples, which makes them very attractive. Moreover, the general appeal of neural networks is vast, such as massive parallelism, learning and generalisation ability, adaptivity, and fault tolerance (ability to work with incomplete data) (A. K. Jain, Mao, and Mohiuddin, 1996b).

Parallel processing is useful for this project because multiple layers can simultaneously be implemented and computed, which will enable better learning and performance benefits. Learning ensures hidden structures in data are captured. Generalisability that the trained model can be effective on unseen data (i.e. news topics, or different domains of information source). With the use of backpropagation, neural networks become adaptive problem-solving algorithms, capable of handling difficult problems. Neural networks have a high degree of fault tolerance which decreases general model-related errors, thus, resisting performance decline because the dataset would be likely have incomplete or null instances. In addition, neural networks can function with limited or incomplete data: a benefit for this project as the one of the limitations of this project is access to a sufficient and high-quality dataset.

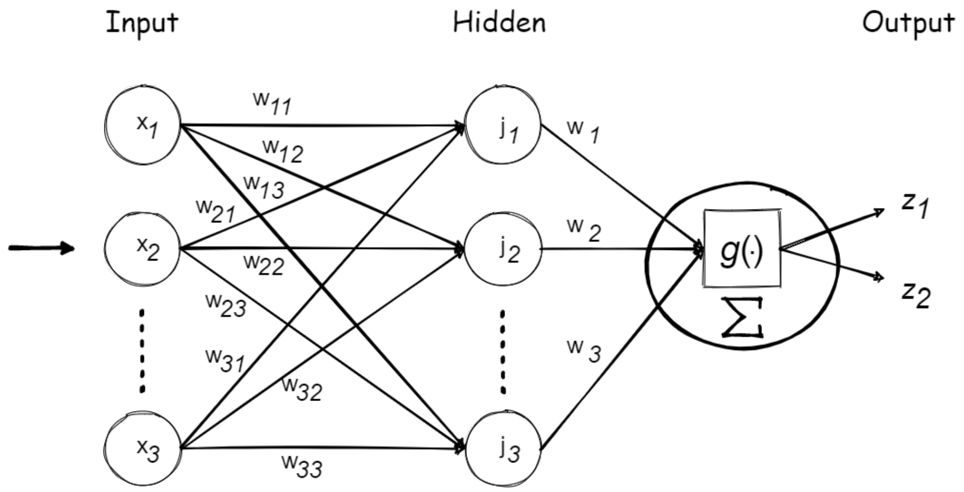


**Figure 2.5** A simple *feed-forward*, *fully-connected* neural network with 3 input neurons, 3 hidden neurons, an unidentified activation function, and 2 outputs. The weights ($w_{ki}$) simply state that the neural connection between input $k$ to hidden neuron $i$ is established (i.e. $w_{12}$). For simplicity, the weights do not have any numeric value, but in real applications, these weights are initialised to arbitrary numeric values. We know that the weights are synonymous to *synapses* in biological neural networks. The hidden layers calculate the output of $\mathbf{wx}$, which mathematically is: $\sum_{i=1}^{n} w_{ki}x_i + b_{ki}$, where $b =$ bias. In the output layer, the computation: $\sum_{j=1}^{j} w_{ki}j_{ki} + b_{ki}$ is calculated to obtain the values of the output weights.

On the other hand, neural networks also have disadvantages. The first is the need for large amounts of data, particularly for supervised learning. This is because a neural network requires many examples (input/output pairs) to find patterns, and make predictions based on these patterns. The second issue is that neural networks are not transparent, meaning we do not truly understand how calculations are computed in the hidden layer, because in the hidden layer, we cannot make observations. This can be an issue when trying to understand or evaluate the effectiveness of a neural network because we cannot evaluate something that we do not know how it works. Lastly, depending on the architecture and task, neural networks can be computationally expensive. The complexity tends to be correlated with the amount of data, the depth of the model (number of total layers), the activation functions used, and complexity of the task, to name a few (Dumitru and Maria, 2013).

### Learning

Learning in neural networks is accomplished with a *hyper-parameter* (a parameter that is pre-defined) called the *learning rate*, typically represented as $\alpha$ or $\eta$. The learning rate dictates the amount of change proportional to the error. Learning is achieved by iteratively tuning weights until convergence. Choosing the learning rate can be tricky, as a high value increases the likelihood of overshooting the minima; and a low value implies slow learning.

The equation for learning is:

$$\mathbf{Z} = \frac{\partial Error}{\partial w_{ij}} \tag{2.32}$$

where *Error* is computed via backpropagation; learning is applied with:

$$\mathbf{w}_{ij} = w_{ij} - \alpha \frac{\partial Error}{\partial w_{ij}} \tag{2.33}$$

where $\alpha$ is the learning hyper-parameter that is applied accordingly to minimise the error, in respect with the *error* in the previous iteration (Anthony and Bartlett, 2009a).

There are many learning methods, each with their benefits and limitations. Namely, a *learning rate schedule* is the concept of changing the learning rate after iterations or epochs. The benefit is that learning does not get stuck in a local minimum, and this is achieved with *decay* and *momentum.* Decay forces learning to decrease in small amounts of change as iterations pass, which allows learning to avoid overshooting the minima, or converging prematurely. Momentum forces learning to avoid settling in the sub-optimal local minima, and this allows learning to find the optimal global minima instead. Both parameters are controlled by the learning-rate hyper-parameter. However, the problem of a learning rate schedule is that it is heavily reliant on the learning rate hyper-parameter, which reduces any adaptive learning capability. Furthermore, the sequential reduction of learning as iterations pass may not be optimal because learning may stop too soon. Also, because the schedule is set in advance, learning does not adapt to the dataset as iterations increase, and this may cause performance decreases. To overcome such sub-optimal solutions, a better learning method called *adaptive learning* was developed. Adaptive learning solves the problem of over-dependency to the learning hyper-parameter because learning changes according to the error produced per iteration. This also helps algorithm adaptation to the dataset which could enhance learning (Anthony and Bartlett, 2009b).

It is therefore more beneficial to use adaptive learning for the justifications above. One method of adaptive learning is *Adagrad.* Briefly, Adagrad works well with sparse data (data instances that are zeros) by adapting to the dataset as iterations pass, which is important because datasets are likely to have sparsity. Adagrad applies learning in an adaptive manner by scaling the learning rate for every dimension, which implements a learning rate on the model that is well-balanced (neither too high, neither too low). One issue of Adagrad is that it employs an aggressive monotonic reduction of learning, which can cause instability (Lydia and Francis, 2019). An alternative method called *Adam* could be used instead. Adaptive Moment Estimation, or Adam, is a method that computes the learning rate proficiently per parameter (Diederik P Kingma and Ba, 2014b). Adam, just like Adagrad, stores the exponentially rising average of gradients. In comparison to Momentum, where the gradient can be viewed as a gradual decline of a ball over the slope, Adam instead acts with friction on the downward fall, controlling the decay of learning with resistance. This shows that the learning rate decreases as the method approaches the minimum. (Diederik P Kingma and Ba, 2014b) demonstrated that Adam performs better than other methods due to this method of resisting a downward fall, with better performance.

### Neural Network Architectures

Since the perceptron, neural networks have evolved into many architectures. Due to project scope limitations, we only present architectures that are likely to draw good results, such as *Convolutional Neural Networks* (CNNs), *Recurrent Neural Networks* (RNNs), and *Long/Short-Term Memory* (LSTMs), a special type of RNN. We will also discuss and evaluate these architectures on the reasons why they may be effective for the project, in chapter four. For readers interested in more information on CNNs, RNNs and LSTMs, see (O'Shea and Nash, 2015) and (Medsker and L. Jain, 2001), respectively.

A convolutional neural network is a *deep learning* architecture (meaning, there are many computing neurons in the hidden layer) that is highly effective at solving computer vision tasks (Britz, 2015). Typical world applications include self-driving cars, satellite-imagery analysis and photo-tagging. Like traditional neural networks, there are input, hidden and output layers, where the weights and biases are learned, with backpropagation. Moreover, the input is an image (in pixels), and the output is the class of the image, such as dog or cat. The main difference from standard feed-forward architectures is that CNN neuron-to-neuron connections are localized, meaning each input region is connected to an output region.

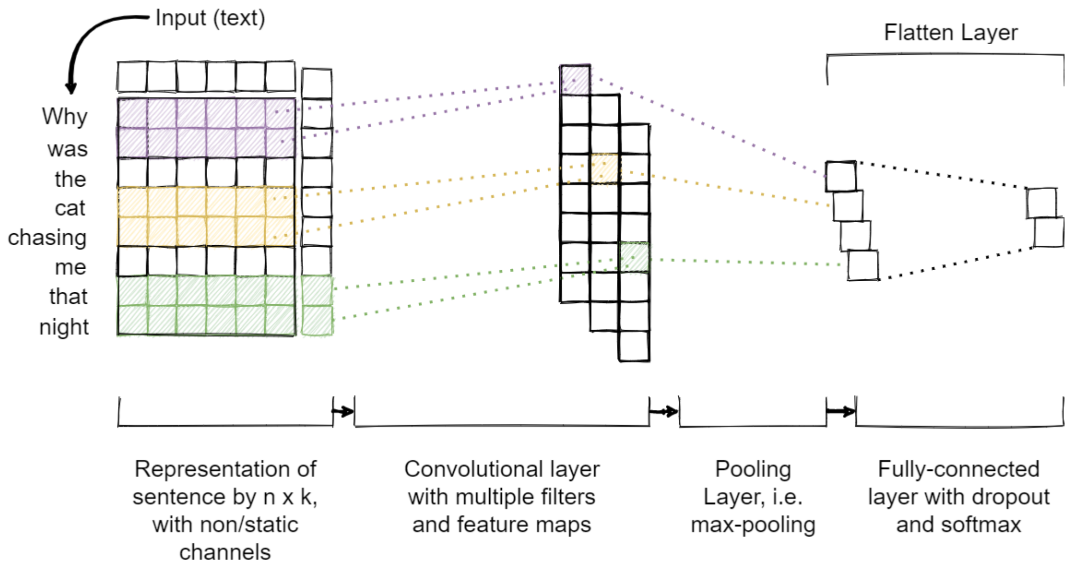

Input (text)
Flatten Layer
Why
was
the
cat
chasing
me
that
night
Representation of sentence by n x k, with non/static channels
Convolutional layer with multiple filters and feature maps
Pooling Layer, i.e. max-pooling
Fully-connected layer with dropout and softmax

**Figure 2.6** A simplified standard CNN model with a text processing task. Briefly, each word is represented by channels. The convolution layer convolutes each word channel (the multiplication of weights and inputs). This is then passed to a pooling layer (i.e. max-pooling), where the spatial size of each feature is reduced to reduce parameter size. The result is then passed (typically) to a fully-connected layer.

Typically, the activation functions used in CNN's are non-linear, i.e. sigmoid. In brief, CNNs employ layers of convolutions, where each layer has an activation function applied. What CNNs excel at is that they can automatically learn filter values of a given task. When it comes to NLP tasks, CNNs have been demonstrated to work surprisingly good. Instead of pixels as input, the CNN model is fed sentences or a corpora of documents, which are represented as a matrix. In the matrix, each row is a *token* (a character or word). These vectors form *word embeddings*, which are low-dimensional word representations.

| **Advantages** | **Disadvantages** |
|---|---|
| Computationally fast | Requires large amounts of data |
| Parameter sharing ability | Complicated due to many hyper-parameters |
| Identifies hidden patterns accurately | Prone to over-fitting |
| | Black-box model |

**Table 2.1** The advantages and disadvantages of CNNs (O'Shea and Nash, 2015).

Recurrent neural networks are sequential models (i.e. reading left to right) that use information (inputs) that are dependent on each other. RNNs are primarily used for natural language processing tasks, such as speech recognition. In comparison to traditional architectures, such as feed-forward CNNs, RNNs perform computations that rely on previous information for new outputs. In other words, RNNs hold (limited) memory on previously computed information. For example, for word prediction tasks, RNNs will capture and hold memory on previous words to make predictions on the next set of possible words as input. There are architecture variations of RNNs, such as *one-to-one* (traditional neural network for binary classification), *many-to-one* for sentiment classification, and *many-to-many* for machine translation (language translation). The most appropriate variation for this project is one-to-one, where we have one input and one output. Furthermore, since traditional backpropagation cannot work on RNNs, a special version of backpropagation called *Backpropagation Through Time* (BPTT) is implemented on RNNs. Conversely, BPTT is implemented at every point in time $t$, where the derivative with respect to weight $w$ is computed (Lillicrap and Santoro, 2019). The expression for BPTT with loss function $J$ is:

$$\frac{\partial J^T}{\partial w} = \Sigma_{t=1}^{T} \frac{\partial J^T}{\partial w}|_t \tag{2.34}$$

| **Advantages** | **Disadvantages** |
|---|---|
| Capable of sharing weights | Deep models are intractable |
| Processing ability of any length | Inability to compute long sequences of data |
| Ability to memorise past information | Computationally slow and expensive |
| Model size tractable with increasing inputs | Prone to the vanishing/exploding gradients |
| | Black-box model |

**Table 2.2** The advantages and disadvantages of RNNs (Medsker and L. Jain, 2001).

Long Short-Term Memory, or LSTM, is a variation of traditional RNNs, with nearly identical architectures. LSTMs are capable of tracking a long history of previous inputs, thereby defeating RNNs at effectively using historical information. Another benefit that LSTMs offers is the ability to compute long-term memory. Since we have a dataset with

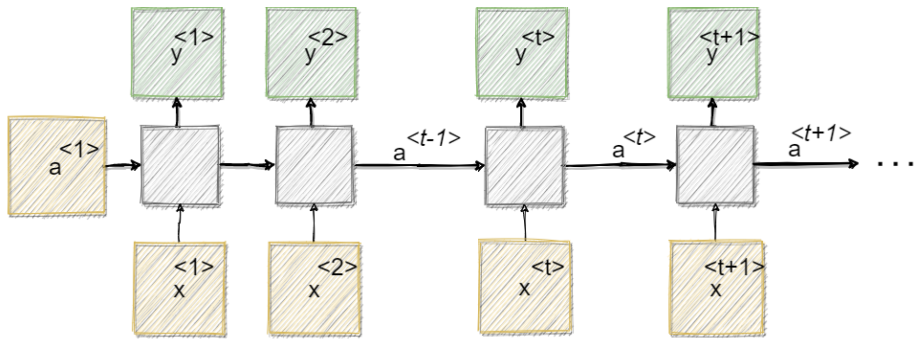


**Figure 2.7** The vanilla RNN architecture. At each time step, the previous word is put into context for the next word prediction/output. The yellow squares resemble individual inputs, the red squares compute these inputs; the result is an output (green squares). The activation as well as BPTT is computed at each time step $t$.

text, the importance of long-sequence computation is critical, as the ability to compute these long-sequences will imply that important information is not left out from the first word of the sentence. Similar to RNNs (which has standard units that perform computations and activation's), LSTMs have special units called the *memory cell*, where historical information is kept for long periods of time. Because LSTMs keep historical information (without intractability problems), the results are better because past inputs are put into context for present and future inputs. There are also more control parameters in LSTM layer structures, the flexibility of the model increases, and this leads to better results. More over, LSTMs solves the vanishing/exploding gradient problem by implementing an *additive gradient* structure, which controls the flow of gradients, via a constant. This constant prevents the gradients from exploding or vanishing (Hochreiter and Schmidhuber, 1997). Likewise, the BTPP gradient descent method is also implemented on LSTM.

| **Advantages** | **Disadvantages** |
|---|---|
| Memory capability | Complex model |
| Solves vanishing/exploding gradients | Computationally slow & memory expensive |
| Can handle noisy data | Prone to over-fitting |
| Can handle long sequences of data | Black-box model |
| Flexible and tractable model | |

**Table 2.3** The advantages and disadvantages of LSTMs (Medsker and L. Jain, 2001).

### 2.2.6 Datasets

A *dataset* is a collective data structure of related information. A dataset can be collective of any type of information. In supervised learning, dataset instances are input/output pairs, where each instance is annotated with a category or class. The classifier uses this label to make predictions, and reduce error (via backpropagation). Datasets are comprised of rows and columns, with each row represents an instance; a column represents a feature. An instance is simply an example data, whereas a feature is an aspect of a particular instance.

| Sepal Length | Sepal Width | Petal Length | Petal Width |
|---|---|---|---|
| 4.1 | 2.1 | 1.1 | 0.1 |
| 6.0 | 0.4 | 1.4 | 0.3 |
| 5.1 | 3.1 | 0.9 | 0.2 |
| 4.7 | 2.4 | 2.1 | 0.5 |
| 5.2 | 2.9 | 2.4 | 0.2 |

**Table 2.4** The classic iris dataset extract. The columns are the features; the rows are the instances. In a real-world application, we are likely to have hundreds or thousands of instances and multiple features.

With datasets, important aspects must be considered, particularly for learning algorithms, such as neural networks. The size of the dataset should be relatively sufficient so that the algorithm does not *under-fit* (the algorithm does not learn or generalise). Also, *over-fitting* is important to avoid because the algorithm may simply model the dataset extremely well, leading to negative performances when exposed to un-seen data. In addition, the sentiment of the dataset should be as neutral as possible (Torralba and Efros, 2011). This is because if the dataset is biased, the algorithm will tend to be biased. As this project has significant societal importance, it is therefore both ethical and practical to use unbiased data.

### Pre-Processing

The quality of a dataset is determined by the usefulness of the information it contains (Shanker, Hu, and Hung, 1996). Furthermore, the better the dataset in terms of quality and quantity, the better the performance of the algorithm. Since Twitter data is likely to have all sorts of qualities, it is useful to clean the dataset before any training in order to create a reliable and robust dataset. It is common that real-world datasets may contain missing data, where one or more instances have no values, or are blank (i.e. $NaN$ and $Null$ placeholders). If such occurrences are not dealt with, then unreliable results may be computed. However, it is relatively easy to pre-process data, as most machine learning tools and platforms offer libraries to handle such tasks. Table 2.5 gives an illustration of a dataset with missing values.

| Sepal Length | Sepal Width |
|---|---|
| 4.1 | 2.1 |
| $NaN$ | $Null$ |
| $NaN$ | 3.1 |
| 2.5 | 1.4 |

**Table 2.5** The iris dataset extract again; this time with missing data.

**Dataset Partitioning**

We introduced the importance of using a reliable and high-quality dataset. However, it is equally important for a dataset to be partitioned into training and testing subsets. For unbiased results, and to promote generalisability, we must test our algorithm with un-seen data. It is important to note that whilst the partitioning of a dataset is crucial, it can also be inflicting as the test dataset may hold valuable information from the algorithm. So, the test dataset must be sufficiently sized enough to provide accurate estimations of generalisability, whilst not too big to hold back valuable information. It is therefore important to find a good balance trade-off. In real-world applications, this trade-off is usually 60 : 40, with 60% for training and 40% for testing. This trade-off is usually optimal for small-to-medium sized datasets as there is enough information for the algorithm to benefit from, whilst simultaneously using enough samples to compute an accurate estimation of generalisability.

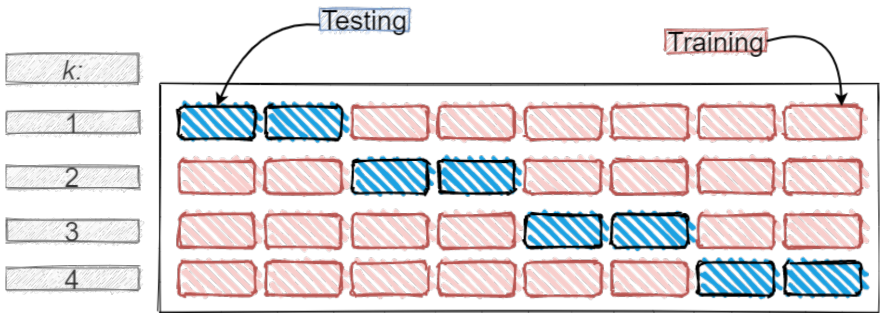


**Figure 2.8** The k-fold cross validation. Here, $k = 4$ (iterations). The blue squares represents the testing subset, where the red is the training subset.

However, a more robust method of data partitioning is *k-fold cross validation*, which is usually used for algorithm evaluation. Simply put, cross validation is the reservation of a subset that is kept un-seen during training, but is used during testing and validation. This significantly reduces bias input (both from engineer and algorithm), and improves reliability and generalisability. The $k$ is a placeholder for the $n$ number of iterations the method is applied for (i.e. $k = 4$) (Shanker, Hu, and Hung, 1996).

### 2.2.7 Model Evaluation

Evaluation metrics are indicative of the performance an algorithm. The right performance metric to use is important, as the wrong one will render the evaluation unreliable or inaccurate. One of the most widely used evaluation metrics for neural networks is the *Confusion Matrix*. Simply put, for classification tasks, the confusion matrix tells us how the system is performing in terms of in/correctly classifying samples (Carlini and Wagner, 2017). The confusion matrix compares the predicted (desired) outcomes with the actual outcomes. For example, if the predicted outcome is the same as the actual outcome, then this is a *true-positive*, meaning the desired outcome was also computed correctly by the model. On the other hand, if the predicted outcome does not match the actual outcome, then this is a *false-positive*, because the model computed an undesired outcome. True-positives and true-negatives are indicative that algorithm performance is good. Conversely, false-positives and false-negatives imply that the model is performing poorly. Thus, the higher numbers of true-positives and true-negatives, the better model performance.

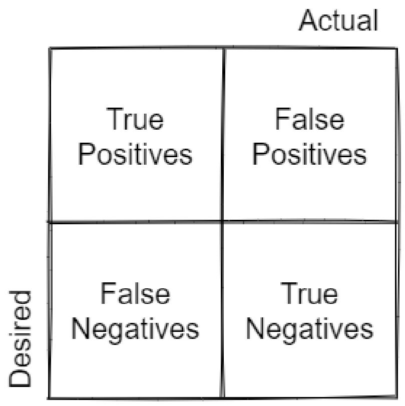


**Figure 2.9** A confusion matrix visualisation.

In addition, there are other performance metrics used for measuring the performance of the model, such as:

$$\textbf{Accuracy} = \frac{TP + TN}{P + N} \tag{2.35}$$

where $P$ are positive instances; $N$ are negative instances. Accuracy is useful because the computation takes all instances into consideration. However, accuracy can be a misleading metric for imbalanced datasets because of the bias input of true-positives.

$$\textbf{Precision} = \frac{TP}{TP + FP} \tag{2.36}$$

Precision only takes relevant instances into consideration. The purpose of this metric is to find out how much of the true-positives are indeed classified correctly.

$$\textbf{Recall} = \frac{TP}{TP + FN} \tag{2.37}$$

Recall is the measure of sensitivity, where the proportion of actual positives are classified correctly.

$$\textbf{F1 Score} = 2\frac{Recall \times Precision}{Recall + Precision} \tag{2.38}$$

Sometimes called the *harmonic mean*, the F1 Score is simply the weighted average of precision and recall, where both instances are considered (positives and negatives). By computing this metric, we obtain an unbiased, neutral measure. The F1 Score is indicative of how well the model is generalisable.

### 2.2.8 The Programming Language Python

*Python* is an interpreted scripting and object-oriented programming language. It has high-level semantics, and data structures with dynamic typing and binding functionalities. One useful feature of Python is there is no compilation of code, which translates to increased functionality (Van Rossum et al., 2007). Python is used to create a plethora of tasks, such as programs, websites, desktop applications, games, software, and data analysis. Furthermore, it is the favoured programming language for developing machine learning, data mining, and any artificial intelligence related algorithm or program. There is also an abundance of libraries and platforms that support Python, such as scikit-learn and TensorFlow, which allow for flexibility of programming. Lastly, Python makes it easy to express high-level abstract concepts conveniently with libraries, such as TensorFlow. The project utilizes Python version 3.7. For writing our solutions in code, we used *Pycharm*, which is free to use.

### 2.2.9 Natural Language Processing

NLP helps computers understand language. Certain NLP processes and tools were used to procure a solution to our problem. Briefly, we present these processes and tools.

#### Tokenization And Stemming

*Tokenization* is the process of breaking down a sentence into pieces. The broken-down pieces are called *tokens*. During tokenization, punctuation is removed to simplify the tokens. For example, the sentence ”let there be light” is tokenized into "let", "there", "be", "light". Tokenization helps neural networks understand text. Tokenization is also the backbone for other NLP models, such as word embedding. Once tokenization has been accomplished, we then involve a process called *stemming*, which is the process of grouping alike words together.

#### Text Embedding/Vectorization

*Word embedding* involves vectorizing words into real numbers. With vectorized words, we can then extract meaningful patterns. Methods of vectorizing words are the bag-of-words (Yin Zhang, Jin, and Zhou, 2010), word2vec (Ma and Yanqing Zhang, 2015), and GloVe (Pennington, Socher, and Manning, 2014b) models. We briefly explain these models below.

Word2vec produces vectors that represent the text numerically. Word2vec groups words of similarity to each other in a vector-space. In addition, the similarity of words leads to associations of words in the vector-space. The output is a vocabulary, where every word has an attached vector, which is then used as input to the neural network. The similarity measure is computed with the cosine trigonometry measure (the Euclidean norm). The main disadvantage of word2vec is that it fails to identify the representations of out-of-vocabulary words.

GloVe, just like word2vec, captures representations of words in vectors. Training is performed on word-to-word co-variance statistics, which produces a co-variance matrix of words. The cosine similarity measure is computed between words. Essentially, the co-variance matrix (how two things are related to each other) for word-to-word is computed. Unlike word2vec, GloVe computes the global statistics of how words co-vary, which reduces the reliance of local statistics. GloVe has two disadvantages: the co-variance matrix is computationally expensive to compute; GloVe, just like word2vec, fails to capture representations of unseen, or, out-of-vocabulary words. Figure 2.9 visualizes how Word2vec and GloVe map words in feature space.

The cosine similarity measure is expressed as:

$$S = \frac{w_1 \cdot w_2}{||w_1||||w_2||} = cos\theta \tag{2.39}$$

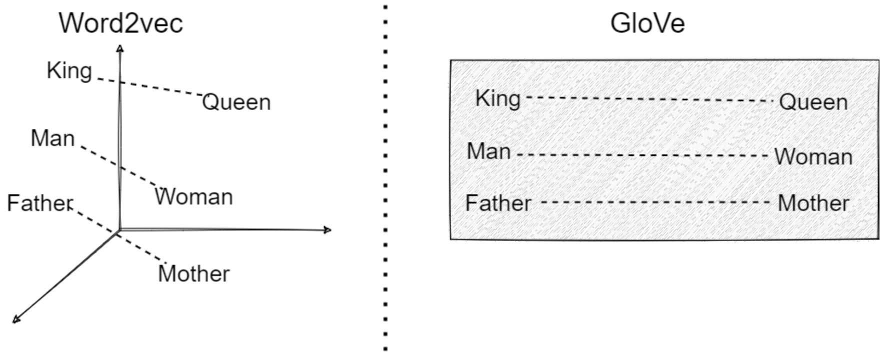


**Figure 2.10** A visualisation of Word2vec and GloVe in feature-space.

Bag-of-words is the traditional model for extracting features, and forming vocabularies from text data. The reasons this model was not considered for this project is because the bag-of-words model does not compute the co-variance of words, because it assumes words are independent of each other. This was considered detrimental to the performance of the solution because it does not compute co-variance of words. In other words, it does not capture context, which can be used to calculate the probability of another word appearing next. Another reason is because it creates a large vocabulary, where the dimensionality of the feature space increases proportionally, which can become computationally expensive.

### Stop Words

*Stop words* are commonly found words in sentences. Stop words are filters that remove such common words, such as ”and”, ”the”, ”but”, ”I”, and so on. By implementing stop words, the quality of the processed text data increases as words that contain none or minimal meaning or importance are removed.On the other hand, removing stop words may lead be detrimental to performance. For example, for sentiment analysis (the assessment of the polarity of a given text or sentence), the removal of stop words may yield to sub-optimal results because certain removed words may change the sentiment of the sentence (Nadkarni, Ohno-Machado, and Chapman, 2011).

### 2.2.10 Machine Learning Libraries

We present the various open-source, freely distributed machine learning libraries/packages that were used to design and implement our solutions.

#### Google's TensorFlow

*TensorFlow* is a powerful library used by academia and industry for designing and implementing machine learning models. The ability to design and implement large-scale neural networks is made easier with TensorFlow, with the support of many libraries, such as keras and TFlearn. Conveniently, TensorFlow applications are produced mainly with Python programming. TensorFlow allows for the creation of data-flow graphs, which visualise the flow of data via graphs to assist in understanding the machine learning model further. Also, we can optimize our models with TensorBoard, where we can analyse how our model is performing. The most important benefit of using TensorFlow is abstraction, where the library takes care of the very technical aspects of the model behind the scenes, allowing the developer to focus entirely on the overall logic and implementation of the model. This, in turn, would greatly help on reducing time spent on implementing the algorithm proposed in this project. Lastly, the power requirements of TensorFlow, in terms of system configuration, are relatively low, meaning that a low-end machine is sufficient to model very complex and sophisticated models (Ertam and Aydın, 2017). This was important as the project has limitations regarding the machine capabilities.

#### Keras

*Keras* is an open-source, high-level machine learning API that integrates easily with TensorFlow for implementing flexible neural network models. Python is the primary programming language on Keras, which supplements the project's decision to use Python for programming. Keras improves development and implementation processes because there are standalone modules and features, such as customizable neural network architecture layers, loss functions, optimizer, and activation functions, with increased computation speeds (Moolayil, Moolayil, and John, 2019). Also, keras offers callback features that help avoid under/overfitting in neural networks.

#### Scikit-Learn

*Scikit-learn* is an open-source library for machine learning tasks. Scikit-learn allows for easier dataset management, deployment and evaluation via built-in functions and packages such as *Numpy* and *Pandas*. However, Scikit-learn lacks the capability to support complex and sophisticated ML models, such as neural networks. Furthermore, Scikit-learn allows for simple and fast ML tasks with small datasets. Since we have a dataset of around $40,000$ instances, we require a library that could manage a large dataset, such as TensorFlow (Pedregosa et al., 2011). We used *Numpy* for linear algebra for easier and quicker calculations (i.e. multiplication of weights by input), and *Pandas* for working with our dataset (i.e. dataset manipulation), both packages are easily integrated into scikit-learn.

# Chapter Three
# Ethics & Professionalism

Before we present the solutions to our problem, we wish to discuss the issues concerning ethics, professionalism and academic honesty.

## 3.1 Ethics

Ethics is the concern of moral principles that supervise the behaviour of an activity. We, as academic researchers, have an obligation to conduct research that is morally ethical. As such, we have kept this obligation in mind throughout the project.

## 3.2 Professionalism

Throughout this project, the rules and guidelines of the British Computer Society were consulted, namely the "*Code of Conduct*". These rules govern the applications of Information Technology, with the goal of providing a competent and ethical practice for computing applications (Conduct and Practice, n.d.). Whilst most of the rules do not apply to this project, only two does in section **public interest**. This section states that we uphold the rules: "*have due regard for public health, privacy, security and well-being of others and the environment,*" and "*conduct your professional activities without discrimination on the grounds of sex, sexual orientation, marital status, nationality, colour, race, ethnic origin, religion, age or disability, or of any other condition or requirement*". The first rule states that we must consider the safety and security of others and the environment, and that our project does not cause harm. The second rule concerns the dataset we acquired, and it simply means that we must not use the dataset to discriminate or abuse others on the grounds stated above. In addition, the dataset was only used for our research purposes, and no attempt was made to sell/distribute the dataset, nor use it for any other purposes.

## 3.3 Academic Honesty

The project contains various technical implementations and literature research. We clearly referenced and acknowledged work of others, documented code to state original authors (where required), and only used codes or libraries that are freely distributed (open-source).

# Chapter Four
# Methodology & Implementation

In chapter four, we discuss our methodology in detail. We outline the solution to the problem in technical and programmatic detail, with an overview of overall solution designs. We also detail the various problems we encountered and solved.

## 4.1 Overview

During methodology and implementation, we focused on *flexibility* with *validation.* We aimed to design and implement flexible solutions with validations. We chose flexibility because it allows for a greater degree of exploration, which could help find optimal solutions, instead of sub-optimal ones. Thus, we successfully produced three distinct neural network designs. For our project, we created a solution/model life cycle (Appendix A). We present our solutions as Models. Figure 4.1 comically demonstrates the abstract process of machine learning.

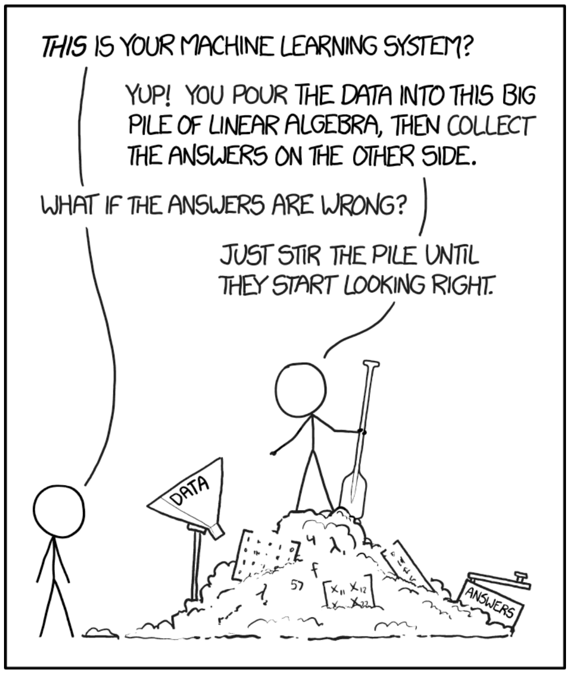

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

**Figure 4.1** www.xkcd.com/1838.

First, we acquire a dataset for training/testing. We then analyzed the dataset for quality and information. If the dataset was poor quality (i.e. contained a lot of useless or missing instances), we would perform data pre-processing (cleaning and denoising). Since we are working with text data, it was important to clean the dataset from punctuation and special characters. This is because such occurrences would be useless during word embeddings as they hold no informational value. Once pre-processing is completed, the neural network is defined (i.e. defining the layers, weights, activation functions, etc). We then split the dataset into training and testing subsets. We then compile and fit the model, initiating training. We then evaluate the performance of our model using relevant metrics. At this stage, we can understand how well our model is performing (i.e. accuracy). If our model does not achieve a baseline 95% accuracy, we tuned the model. We then test our model on our test data, to validate our training accuracy and loss outputs, and to look for any signs of over/under-fitting. This is to reduce the likelihood where our model memorizes the dataset, leading to poor performances on unseen data, thus promoting generalizability. The emphasis on model evaluation is strict here, as we wish to create reliable and validated results, increasing model robustness. If the model shows overall robustness, we can then package the model.

## 4.2 Dataset Acquisition

We obtained a dataset that was originally created for research purposes (free to use), from a github repository. The dataset consists of two *.csv* files, labelled "*True*" (for factual samples), and "*Fake*" (for fictitious samples), in the English language. The dataset is already labelled. It is important to note that alternative datasets were searched for, but, due to time limitations, we could not spend a lot of time to search for alternatives. It was decided that a dataset was to be acquired as soon as possible so that more time would be spent on developing a solution. As a result, we were able to produce three solutions instead of one, which would have been the scenario if we spent more time searching for datasets. Also, because creating a dataset is a tedious and long process, (not to mention possibility of un/intentional bias input) we could not form a dataset synthetically.

### Understanding the Dataset

Data analysis was carried out in order to understand and evaluate the dataset. During analysis, we found the total number of instances is approximately 40,000. The dataset is labelled into two classes - factual and fictitious. Instances for each class is nearly balanced, but with 2,000 more instances for class fiction. We also found out that the dataset contains missing instances, which we removed to reduce sparsity. The dataset has the features *title*, *text*, *subject* and *date*. The title feature contains instances of news articles, whilst the text feature contains tweet instances. We found that some instances contained *source of publication* (i.e. "Washington (Reuters)"), where most of the factual instances had sources of origin. We used this information, by extracting source of origin (publisher) information, and formed a new feature called *publisher*.

| Title | Text | Subject | Date |
|---|---|---|---|
| As U.S. budget fight... | Washington - The head of... | Politics | December 31, 2017 |

**Table 4.1** The first instance from the factual class subset.

The addition of the publisher feature increased model performance because our model learnt that factual information tends to contain sources of origin. Conversely, we found that fictitious information contained more uppercase letters, words, special characters compared to factual information. We wish to use this knowledge in future research because there seems to be a correlation where fictitious information tends to have more words and a higher number of uppercase letters and special characters. However, avoided this implementation for this project because Twitter tweets always have the same source of publisher, where news articles would contained multiple sources (i.e.”Seattle/Washington (Reuters)”). This would have created bias because the algorithm would use this exclusive information only for news articles, and exclude Twitter tweets. We also identified that factual information contained less common words than fictitious information. We therefore realized that the use of stopwords may be required to increase text value. The features *subject* and *date* were tricky because they are dependent on other features, but do not give useful information. However, the addition of the publisher feature added value as the source of publication is a good indication of information validity.

## 4.3 Model Design Overview

Since we have a problem that involves textual data, it was natural to use RNNs as our solution. However, during our evaluation, we realised that a variant of RNN, called LSTMs, was better fit to handle our problem. LSTMs offer long-sequence handling, which is crucial as we are dealing with words that require historical inputs to be memorized and put into future contexts. RNNs fail to compute long sequences of information. This is because RNNs can only either hold information for extended periods of time, or efficiently use gradient descent for learning (Bengio, Simard, and Frasconi, 1994). LSTMs can accomplish long sequence handling because the vanishing/exploding gradients problem is prevented from occurring, simultaneously remembering historical information (for all our models, we avoided using sigmoid or tanh activation functions in hidden layers, to avoid vanishing/exploding gradients). CNNs have been demonstrated to work with text data, with impressive results (Kim, 2014). We wished to explore the capabilities of CNNs on our problem.

Our initial aim was to implement a single LSTM neural network with Word2vec. We then realized that we could also implement another LSTM architecture with GloVe. Rather than choosing one architecture, we wanted to implement both models, and evaluate and compare them. This allowed us to explore the solution sphere. We were also able to implement a CNN architecture, which is a different architecture from RNN/LSTMs. Having settled on solution architectures, we decided to use the same dataset for all models, to avoid model bias and favouritism. We also expanded on this decision and implemented the same pre-training process for all models (pre-processing). Furthermore, the new dataset feature (publisher) was included for all models because this feature was important to the performance of the model. On an important note, every model was evaluated on the same performance metric set (i.e. accuracy, loss). It was important that a fixed number of epochs were implemented for all models, to ensure that no model received favoritism or a disadvantage. Lastly, we implemented the same train/test split ratio of 2:1 for every model. Having said that, we believe that each model should be independent, and therefore needed to be flexible in structure. Namely, other model parameters, such as word embedding size, activation functions, model layer sizes and hidden neurons, should be experimental for each model. Thus, we believe we created a fair and unbiased environment for implementation and evaluation.

### 4.3.1 Model 1: LSTM w/ Word2vec

For our first solution, we implemented an LSTM architecture, with a *Sequential* structure. The sequential is a keras model structure, along with other structures, such as *Functional API*, which offers more sophisticated implementations. As we wanted our solution to be simple yet effective, performing computations in a step-by-step manner, the sequential model served this purpose. However, the functional API structure would have also worked well. Figure 4.2 shows our Model 1 design.

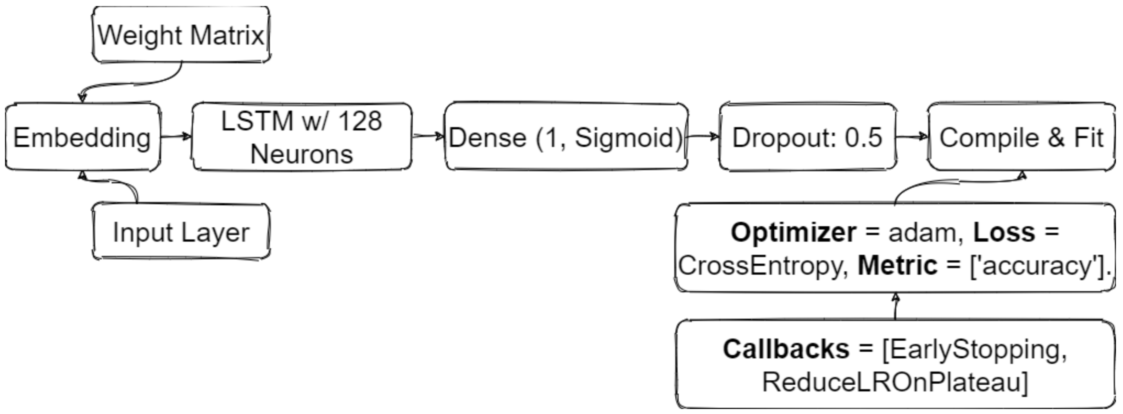

Weight Matrix
Embedding
LSTM w/ 128 Neurons
Dense (1, Sigmoid)
Dropout: 0.5
Compile & Fit
Input Layer
**Optimizer** = adam, **Loss** = CrossEntropy, **Metric** = ['accuracy'].
**Callbacks** = [EarlyStopping, ReduceLROnPlateau]

**Figure 4.2** Model 1 sequential structure. We found that a single dense layer was sufficient for this model (having experimented with denser layers).

During pre-processing, the features *date* and *time* were removed for the reasons stated prior. The extracted feature *publisher* was added into our dataset, and *title* and *text* features were combined into one to reduce redundancy and improve input flow. We also removed instances with missing values, and implemented stopwords to improve data quality. We then implemented word2vec in the embedding layer. We determined the embedding dimension to be 200, but have experimented with 100 and 350. However, we identified that the training times increased with larger dimension values (i.e. 300). Moving on, in the LSTM layer, there are 128 hidden neurons, with a sigmoid activation function. We have experimented with the number of hidden neurons here as well, but found that 128 neurons provided a medium-size structure - not too little, not too much, with impressive results. We found that the sigmoid function was a better fit for our data, and it performed well in terms of model accuracy, and produced very few misclassifications when compared to relu. In addition, we believe that the tanh function would also have produced similar results. During training, we realized that the model was over-fitting (where the model fails to perform as good on unseen data as it did on training data). To fix this problem, we added a *Dropout* layer, to drop 50% of the input from the previous layer for the next layer/epoch. Furthermore, as a precaution, we shuffled the inputs after every epoch to reduce the likelihood that the same input is used for the next epoch. During training, we noticed that learning was very slow after the $2^{nd}$ epoch. One possible cause was that batch size was too small, which resulted in very slow learning steps. We were also observing a slowly decreasing loss. To avoid learning plateau from occurring, we implemented a *callback* feature, called *ReduceLROnPlateau*, which monitors *validation accuracy.* Intuitively, as training epochs increase with no improvement to learning, the learning rate is reduced in tiny amounts (i.e. 0.0001), which promotes change. Thus, learning plateauing no longer occurred as the model successfully adapted to the dataset after each epoch. Lastly, the important hyper-parameters, optimizer and loss, were very interesting to experiment with. We found that optimizers (the tuning of weights) such as Adam, RMSProp and Adadelta, worked best with our data, whilst SGD and Nadam were underperforming. Adam was slightly better than other optimizers as it was able to adapt to our dataset quickly. According to (Diederik P. Kingma and Ba, 2014a), Adam is compatible with large datasets with large neural network parameters (of which we have both), with very little computational memory requirements [...]. By implementing Adam, training times were reduced, simultaneously with errors. For the loss function, we used *Binary Crossentropy*, which calculates the loss of a sample with the computation of averages:

$$\textbf{Loss J} = -\frac{1}{OutputSize}\Sigma_{i=1}^{OutputSize}\ y_i \cdot log\ \hat{y}_i + (1 - y_i) \cdot log\ (1 - \hat{y}_i) \tag{4.1}$$

where $\hat{y}_i$ is the $i^{th}$ scalar value output; $y_i$ is the correspondent target; output size is total number of scalar values in output (Peltarion, n.d.).

We used binary crossentropy because it was empirical to use a binary classification system, with only two possible outputs. Furthermore, we found that the sigmoid function significantly reduced mistakes when crossentropy was implemented. Finally, we monitored the accuracy of our model during training.

### 4.3.2 Model 2: LSTM w/ GloVe

Model 2 was an experimental model with impressive results. Model 2 architecture was similar to Model 1, but with a few differences. We wanted to experiment with GloVe word embedding because it allows the computation of co-occurrence of words, which drastically improves the vocabulary and informational value of data. We demonstrate the co-occurrence of two words in GloVe, in table 4.2.

| Probability & Ratio | $k = solid$ | $k = gas$ | $k = water$ | $k = fashion$ |
|---|---|---|---|---|
| $P(k\| ice)$ | $1.9 \times 10^{-4}$ | $6.6 \times 10^{-5}$ | $3.0 \times 10^{-3}$ | $1.7 \times 10^{-5}$ |
| $P(k\| steam)$ | $2.2 \times 10^{-5}$ | $7.8 \times 10^{-4}$ | $2.2 \times 10^{-3}$ | $1.8 \times 10^{-5}$ |
| $P(k\| ice) / P(k\| steam)$ | 8.9 | $8.5 \times 10^{-2}$ | 1.36 | 0.96 |

**Table 4.2** The probability and ratio of words *ice* and *steam*, calculated in GloVe (Pennington, Socher, and Manning, 2014a).

In table 4.2, we can observe that the word *ice* has a shared property *water*. We can also observe that ice co-occurs with *solid* more frequently than *gas*, where steam co-occurs (naturally) with gas more frequently. The word *fashion* has the least co-occurrence probability with ice or steam. This is a useful feature of GloVe that improves the quality of our solution, as the co-occurrence of words (with accurate estimation) may produce better contexts (Pennington, Socher, and Manning, 2014a). Figure 4.3 demonstrates Model 2.

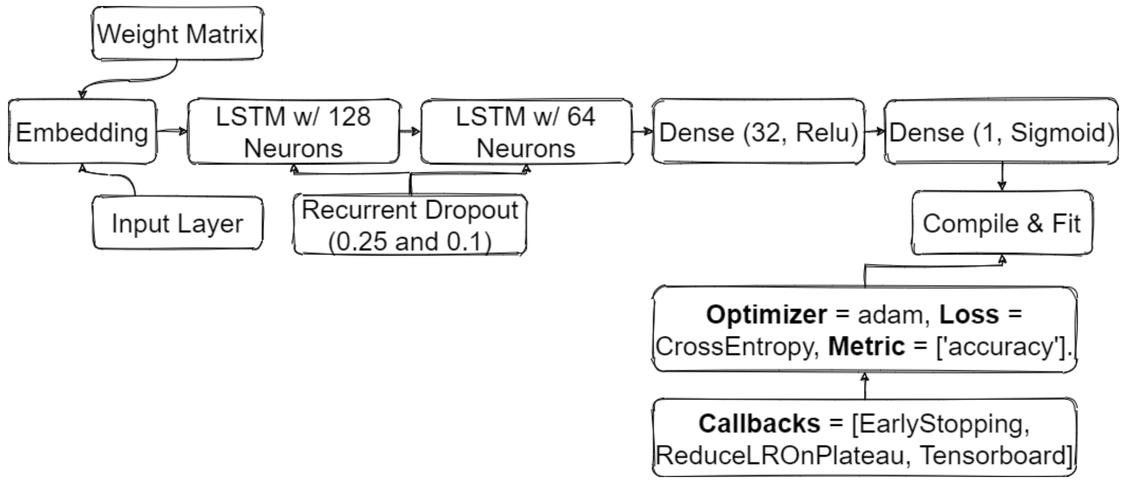

Weight Matrix
Embedding
LSTM w/ 128 Neurons
LSTM w/ 64 Neurons
Dense (32, Relu)
Dense (1, Sigmoid)
Input Layer
Recurrent Dropout (0.25 and 0.1)
Compile & Fit
**Optimizer** = adam, **Loss** = CrossEntropy, **Metric** = ['accuracy'].
**Callbacks** = [EarlyStopping, ReduceLROnPlateau, Tensorboard]

**Figure 4.3** Model 2 sequential structure. We wanted to experiment with denser layers, more activation functions and dropouts.

In terms of model architecture and structure, we experimented with a slightly more deeper neural network. Model 2 is identical to Model 1 up until the structure of the neural network. However, the dimension size of word embeddings was increased to 200, instead of 100, to see if there are increases in performance. In regards to neural network layer structure, we implemented deeper and denser layers, with more dropouts. Namely, we added an extra LSTM layer and Dense layers. We experimented with the sigmoid and relu functions, but settled on relu as it gave a 4% increase in performance compared to sigmoid. However, we added a sigmoid activation in the final layer as the final output function. Interestingly, the more denser layers we added, the more the performance decreased. Also, we noticed increasing signs of over-fitting with more hidden layers. Thus, we settled a slight increase in layer density. We experimented with the same set of optimizers and loss functions (as we did for Model 1) and found that Adam was still the better performer for weight optimization, with RMSProp coming second. For loss function, we tried multiple functions, and settled on sigmoid, because the use of sigmoid in the hidden layer would produce vanishing gradients. However, as LSTMs implement BPTT, exploding/vanishing gradients are avoided. Figure 4.4. demonstrates visually the two most used activation functions in our models.

Model 2 was not without errors. For example, learning plateau occurred after the 4$^{th}$ epoch, where we implemented the callback ReduceLROnPlateau. The most critical problem occurred during GloVe word embedding, where the neural network was crashing before training, without any explicit indication of cause. To fix this problem, we implemented a checkpoint system (in code) to monitor the exact cause of the crash. Thus, we found that the cause was word vectorization, where we couldn't train word vectors. We fixed this problem by using a *.txt* file that contained pre-trained word vectors (freely distributed and available to see at (Halasz, n.d.). We then used the pre-trained vectors in the Embedding layer). The pre-trained vectors also decreased computational time and resource dependency. To conclude, Model 2 produced impressive results, in all performance evaluations, with no signs of over/under-fitting.

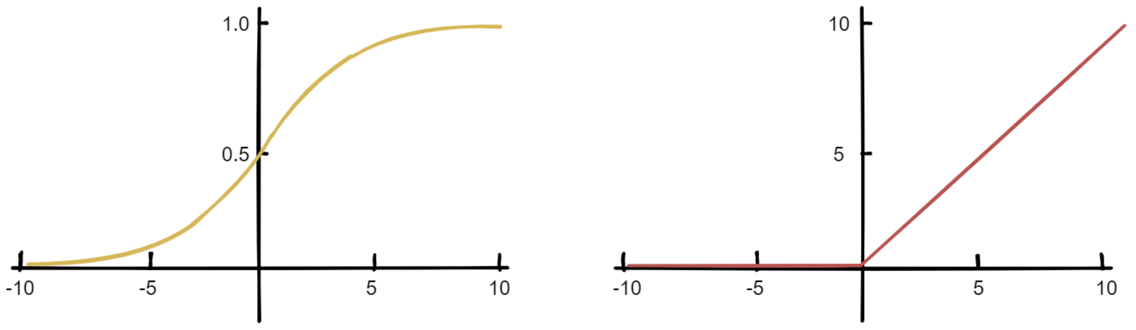


**Figure 4.4** Activation functions sigmoid and relu, with their output ranges.

### 4.3.3 Model 3: CNN

We were very interested in how CNNs could (if ever) solve our problem. We wanted to explore the solution sphere further. Our research has enabled us to experiment with a successful application of CNNs with text input, with impressive results.

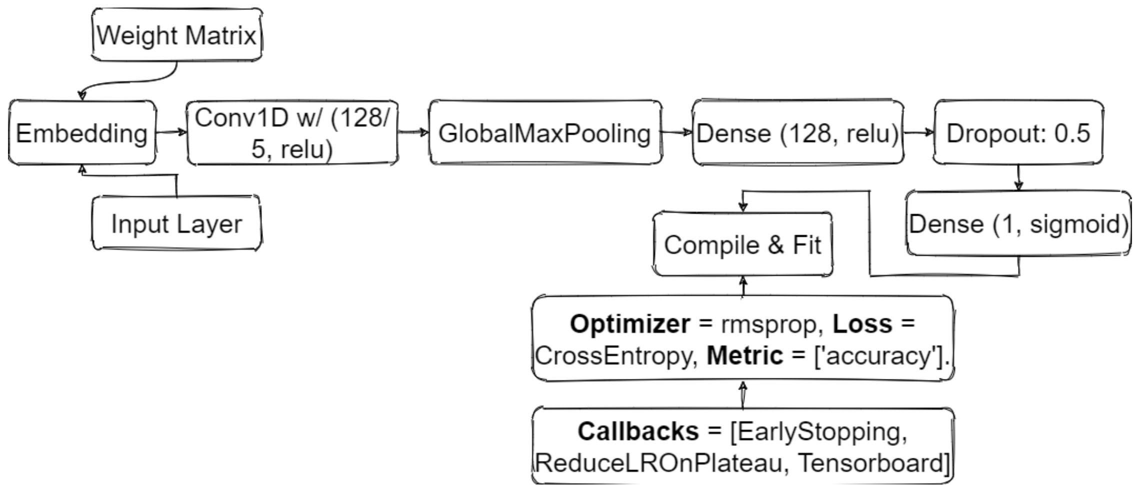

Weight Matrix
Embedding
Conv1D w/ (128/ 5, relu)
GlobalMaxPooling
Dense (128, relu)
Dropout: 0.5
Input Layer
Compile & Fit
Dense (1, sigmoid)
**Optimizer** = rmsprop, **Loss** = CrossEntropy, **Metric** = ['accuracy'].
**Callbacks** = [EarlyStopping, ReduceLROnPlateau, Tensorboard]

**Figure 4.5** Our Model 3 structure.

We experimented with our Model 3, by increasing/decreasing the number of hidden layers, neurons per hidden layer, input size, input dimension, input shape and max length, epoch iterations. We quickly settled on our activation, as well as optimizations and loss functions. During experimentation, we acknowledged that a semi-shallow CNN, with multiple hidden layers and hidden neurons, with relu activation function, gave Model 3's highly impressive and interesting results. Model 3 initially showed signs of high over-fitting early on during development, where we implemented various normalization techniques. More over, learning plateau occurred in Model 3 than all the models combined.

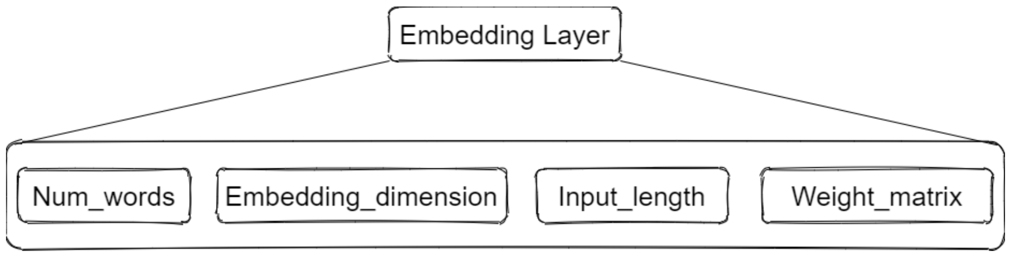


**Figure 4.6** The Embedding Layer, used for all models.

Figure 4.6 shows the Embedding Layer architecture. *Num words* (set for $25,000$) is the total words to be used as input, *Embedding dimension* is the hyper-parameter pre-specified which sets the dimension of the input (200), *Input length* is the maximum length of an input (i.e. including letters, numbers, special characters, etc) set to 5000, and *Weight matrix* are the weights that are computed (or learnt) during word embedding.

The core concept of CNNs are *convolutions*, where mathematical combinations of relationships are computed to produce new, third relationships (Kuo, 2016). The third relationship is the combinatorial information from the two original relationships. Convolutions extract features via the application of filters. Each convolution layer can be controlled via the hyper-parameters *Filter Count* (for feature extraction), *Input Size* (the total number of text input), and *Stride* (total number of inputs to take at once). After convolutions, an activation function (usually non-linear) is applied to the output of the convolution. We used a one-dimensional convolutional layer because this sets our input into a one-dimensional axis, with an output of two-dimensions. This is important because we want our text to be in a single-dimension for time-series (i.e. computing each word in sequences), and also because our input are text and not pixels, where we could use 3 dimensions to represent an image. This also reduces the complexity of the input since we are computing one-dimensional input. We applied *padding* to the input so that the feature map does not shrink (where useful information is lost). This method was also implemented for Models 1 and 2. Furthermore, padding improves model performance because it ensure filter size and stride is compatible with input. To reduce the likelihood of over-fitting, we apply a *Pooling Layer*. Pooling reduces the complexity of the dimensions, whilst keeping a significant portion of information from convolutions by only taking the most important and useful features. We used *GlobalMaxPooling* as our pooling operator because it captured the most interesting features from our dataset. However, other pooling operators, such as MaxPooling, would work too.

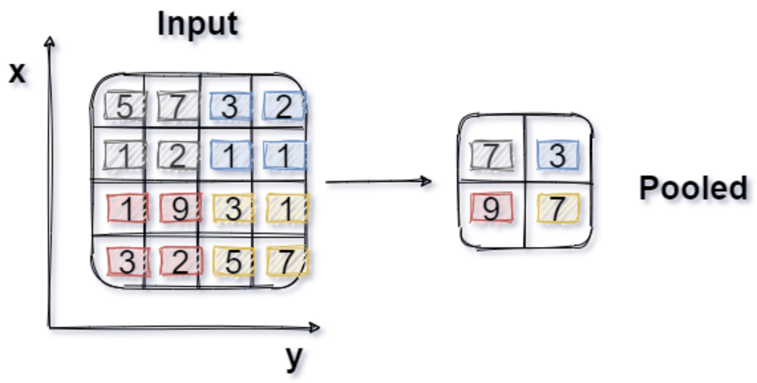


**Figure 4.7** GlobalMaxPooling takes the maximum value of an area.

In terms of neural network architecture, having experimented with shallower and denser hidden layers, we identified that we needed to increase the number of hidden layers and reduce the hidden neurons. We initially experimented with five hidden layers, all with relu activation functions, and dropout normalization implemented in intervals of two hidden layers. We realized that our model was over-fitting the data as there were too many hidden layers that were capturing the same information repeatedly. We reduced the epoch iterations from five to three, which reduced over-fitting slightly. This was a big issue, as we needed an architecture that was not complicated yet dense enough to capture unique/useful information. We then reduced the hidden layers to two, which significantly increased model performance and reduced over-fitting. We then increased epoch iterations to four and identified that our model was no longer learning after the third epoch, as a result of large learning steps. We then reduced the batch size from 300 to 256, which helped the neural network adapt to the dataset in a controlled and efficient manner, taking adaptive learning steps, resulting in stable learning phase. As usual, we implemented the same callback features with the addition of *Tensorboard*. Tensorboard allows for a detailed look at a neural network, which helps in understanding model problems, such as over-fitting. We added tensorboard as we were constantly observing over-fitting and learning plateau. Tensorboard helped in overcoming these problems by giving detailed information during training and evaluation. Overall, our Model 3 was the most impressive, with outputting very respectable results.

### 4.3.4 Solution Summary

Due to the huge combinatorial nature of neural networks, as well as time limitations, we were only able to experiment with a few combinations for each solution. However, we were able to produce three distinct solutions with impressive experimental results. Model 1 was the introductory solution with a traditional and simple structure, with Word2vec word embedding. Model 2 was the attempt of experimenting with a more sophisticated and advanced neural network, with GloVe word embedding. Finally, we implemented a CNN architecture as our Model 3, which proved to be the most interesting and impressive solution. In terms of general neural network design, we implemented a robust system of stop conditions for all our models (e.g early stoppage to avoid over-fitting, dropouts, etc). Lastly, we implemented a pattern of increasing sophistication everytime a new model was designed, to understand whether neural network depth and complexity could result in increased performances. To conclude, we believe we justified our design choices for our solutions, were necessary, in adequate detail.

We have described our solutions in detail. Now comes the important job of evaluating and understanding our neural network solutions.

# Chapter Five

# Results & Evaluation

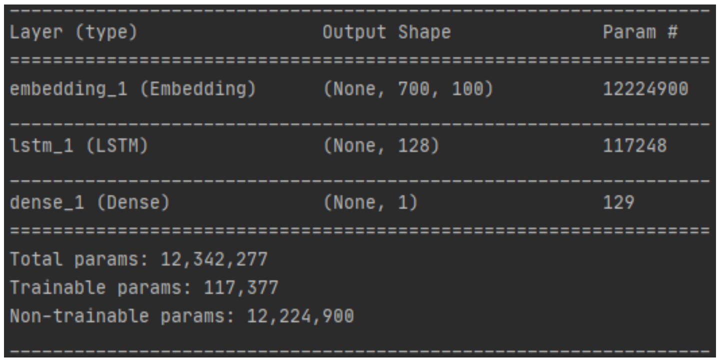
In chapter five, we evaluate the performance of our solutions using relevant evaluation techniques. We also benchmark our experimental results to other relevant work.

## 5.1 Overview

We demonstrate below that neural networks out-perform traditional non-learning algorithms in the state-of-the-art literature. For example, each of our solutions have out-performed the best algorithm proposed or implemented in our literature review. Due to the stochastic nature of neural networks, we present our results produced from the latest runs of our models. However, our results have stayed consistent with multiple testing iterations, resulting in identical or very similar results each time. Recall from chapter Three, subsection *Model Evaluation*, we presented various evaluation methods and processes. To avoid model bias and favouritism, we used the same performance evaluation metrics for all solutions. We present our results and evaluations below.

### 5.1.1 Model 1: LSTM w/ Word2vec

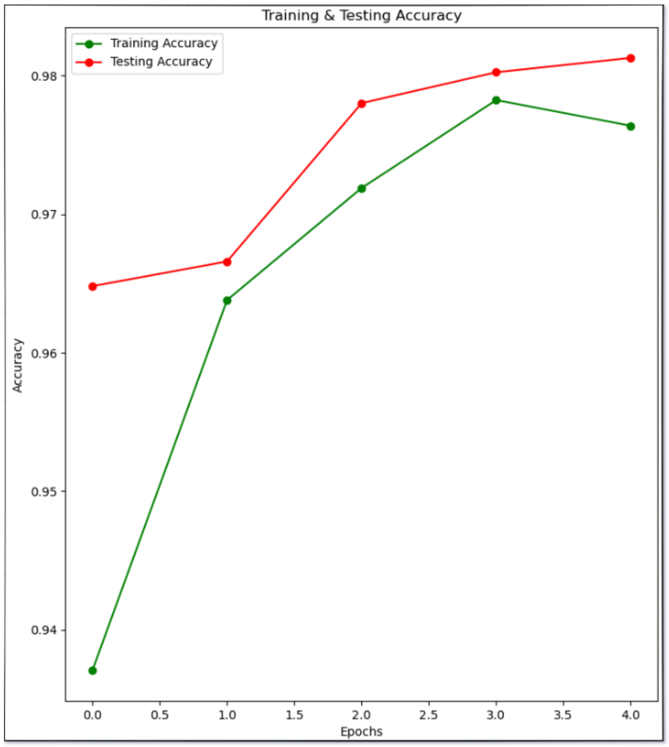
Model 1 has been demonstrated to be a capable and robust solution, with above-standard results. Figure 5.1 shows our Model 1 neural network architecture.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 700, 100)          12224900
_________________________________________________________________
lstm_1 (LSTM)                (None, 128)               117248
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================
Total params: 12,342,277
Trainable params: 117,377
Non-trainable params: 12,224,900
_________________________________________________________________
```

**Figure 5.1** Model 1 structure and parameters.

The total param(eters) signify Model 1's complexity (parameters are input weights and biases). In our embedding layer, we see that a very large number of weights were generated, in the first layer alone. However, when we go deeper, we see that the weights decrease in the LSTM layer, as a result of layer depth and lower number of neurons. Finally, as we implemented a layer with a single perceptron, the result is a layer with a small number of weights and bias. However, weights sharing was not implemented in this model, which may the reason why there are $117,377$ trainable parameters (where backpropagation is implemented for weight optimization). More over, the number of non-trainable parameters (non-trainable parameters are weights that can not be optimized via backpropagation) are over $12,000,000$, suggesting that the model is sophisticated.



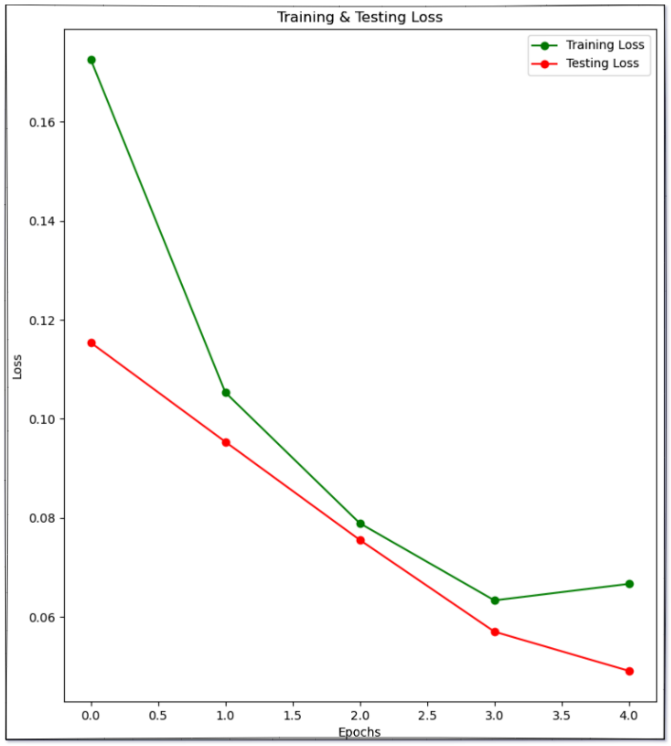


**Figure 5.2** Model 1 training and testing results.

Model 1 shows interesting results in terms of accuracy and loss. Training and testing accuracy start approximately 0.93 and 0.96 respectively. We see a sharp increase in accuracy during training where significant learning occurs from epochs zero to one. Conversely, we see a similar increase of accuracy during testing but not as much, which suggests the model has not adapted to the unseen data efficiently. However, accuracy decreases slightly from epoch three during training, but continues to increase steadily during testing. As our testing accuracy is higher than training accuracy, we assume that Model 1 performs really well with no signs of under/over-fitting. In terms of loss, we observe good results that show our model is efficiently learning, and reaches optimality. Our training loss starts high (0.17), but reduces drastically during the first few epochs. We observe the same when the loss is tested on our test data, with a steady decline of misclassifications. We see that more epochs could have been implemented to further reduce loss. Interestingly, loss during training at epoch three increases (more errors are produced), which may suggest that the model is moving towards the wrong direction of the optimization space. This could be the result of too steep learning steps, which can be resolved with smaller learning steps, forcing the optimizer to take the correct direction (reducing errors). Lastly, loss continues to reduce during testing which suggests that our model is efficient at optimization for unseen data.

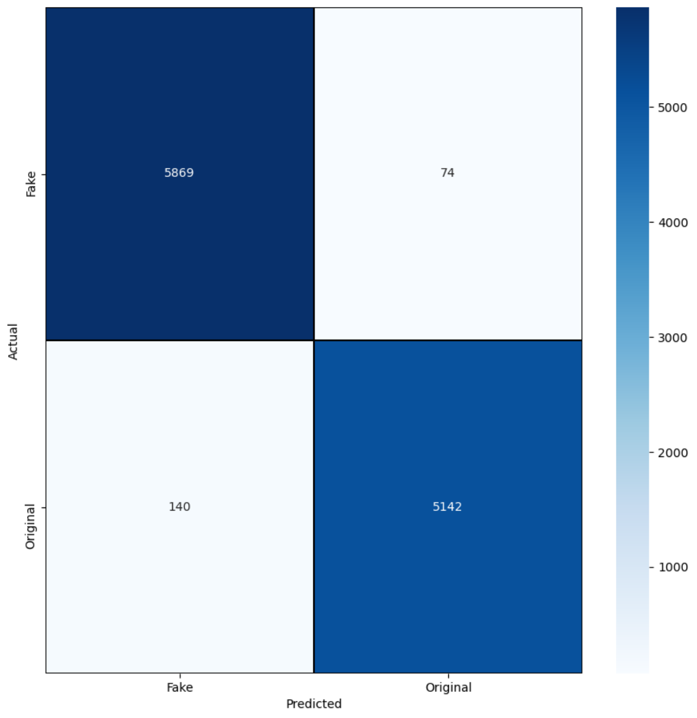
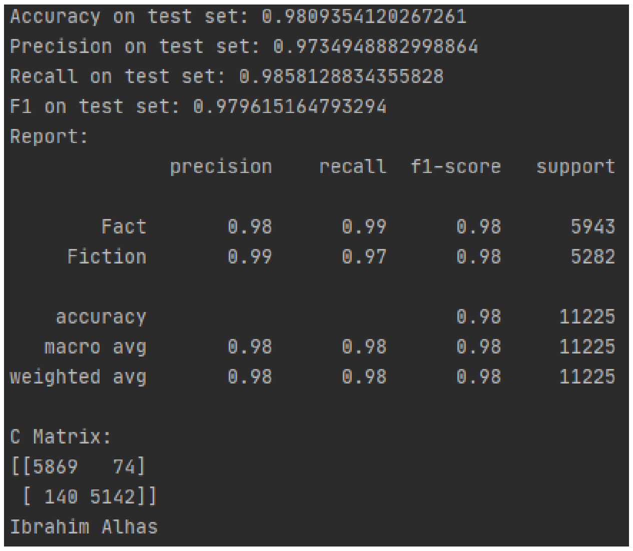


**Figure 5.3** Model 1 confusion matrix.

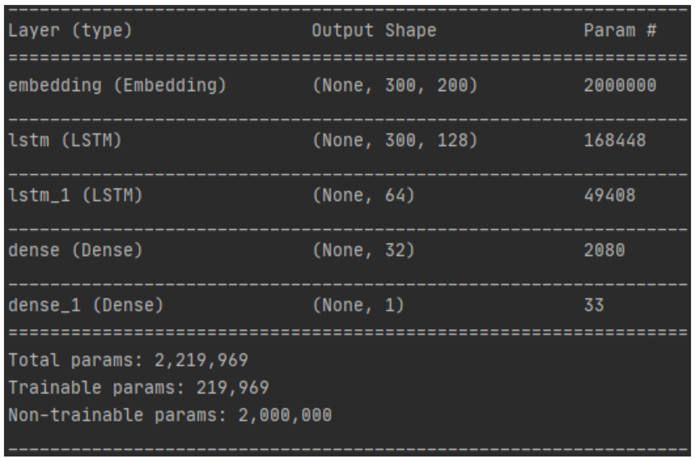
Figure 5.3 shows the confusion matrix for Model 1. An impressive number of samples have been correctly classified, whilst a minor subset has been incorrectly classified. This may be the cause of the sudden increase in loss. We can acknowledge that the model found it the hardest to correctly classify samples that are actually what they appeared to be (i.e. our model falsely classified a sample as fiction even though it was a fact). Figure 5.4 shows the precision, recall and f1-score performance results. Our model performs well on all performance metrics, with a good precision score that suggests how frequent the model is correctly classifying samples. We used the F1 score to see how generalisable our model is, with a very respectable 90% score, suggesting high generalizability. Overall, our model has a validated accuracy of 99%.



```
Accuracy on test set: 0.9809354120267261
Precision on test set: 0.9734948882998864
Recall on test set: 0.9858128834355828
F1 on test set: 0.979615164793294
Report:
              precision    recall  f1-score   support

        Fact       0.98      0.99      0.98      5943
     Fiction       0.99      0.97      0.98      5282

    accuracy                           0.98     11225
   macro avg       0.98      0.98      0.98     11225
weighted avg       0.98      0.98      0.98     11225

C Matrix:
[[5869   74]
 [ 140 5142]]
Ibrahim Alhas
```

**Figure 5.4** Model 1 overall performance results.

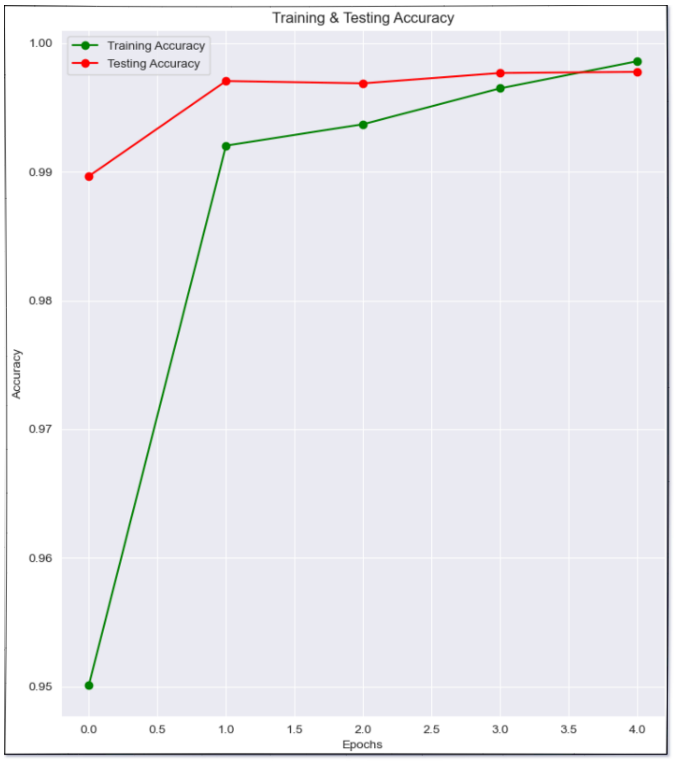
### 5.1.2 Model 2: LSTM w/ GloVe



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 300, 200)          2000000
_________________________________________________________________
lstm (LSTM)                  (None, 300, 128)          168448
_________________________________________________________________
lstm_1 (LSTM)                (None, 64)                49408
_________________________________________________________________
dense (Dense)                (None, 32)                2080
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 2,219,969
Trainable params: 219,969
Non-trainable params: 2,000,000
_________________________________________________________________
```

**Figure 5.5** Model 2 structure and parameters.

Model 2 has been a successful experimental model in all aspects. Figure 5.5 shows neural network parameters. Interestingly, there are lesser total parameters than Model 1, but with more layers. Also, the number of trainable parameters (weights) are slightly more compared to Model 1. Figure 5.6 shows Model 2 accuracy and loss progress during training and testing. We see that training accuracy at epoch zero started at 0.95, which implies that the neural structure and architecture was robust to begin with. We can observe that model accuracy was most improved during epoch transition zero to one. Model 3 slowly but continuously improved accuracy from epoch one to four, indicating that the model could have benefited from more epochs of training (we were unable to do so as our machine crashed due to insufficient memory). However, at epoch four, the accuracy was very close to optimal (1.0). On the other hand, we can observe that the accuracy of the testing subset began with 0.99, a very good sign that the model is generalized, and has performed well on unseen data. Interestingly, accuracy increases significantly between epochs zero and one. Testing accuracy levels out after epoch one, whilst we see a steady incline in training accuracy until epoch four. However, we notice that testing accuracy has converged at epoch two, where there are no significant increases in accuracy. On the other hand, training accuracy shows no sign of accuracy plateau, which may likely continue until epoch ten, based on the slope of the accuracy. At epoch four, testing accuracy has fallen below training accuracy, which may imply that the true accuracy of the model is not optimal, but sup-optimal. The testing accuracy was approximately 0.997, where the training accuracy is 0.998. The differences are minimal, suggesting that the model performs as good on (unseen) testing data as it did on training data.



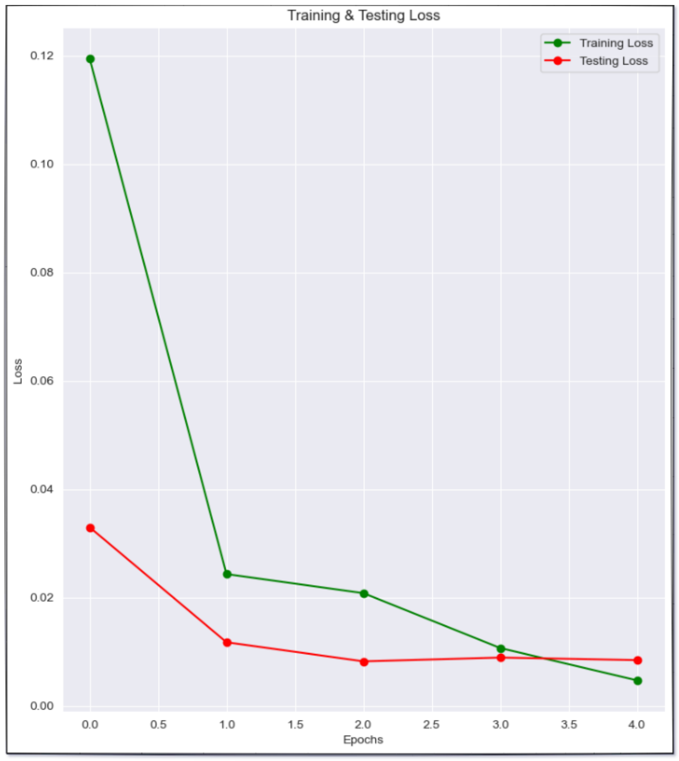


**Figure 5.6** Model 2 training and testing results.

Conversely, our training value for loss starts at 0.12, where loss for testing is 0.04, which implies that the model has significantly optimized itself efficiently, via BPTT. Recall from chapter Two, subsection Gradient Descent, that we are interested in the lowest value of loss (error), which is the global minima. Figure 5.6 confirms our model has reached an optimal loss value at approximately 0.001, for training. We see a sharp decline of errors from epochs zero to one, which is true for training and testing data. Loss during training continues to decline from epoch two, which suggests the model could have also benefited from more epochs for reaching (possibly) a loss value of 0.0. However, loss during testing has stopped reducing (suggesting a plateau) from epoch two, which continues until epoch four. A similar scenario occurs for loss as both, training and testing, have a common epoch (3.5) where the model overestimates loss during training, albeit by a small value difference. Overall, training and testing results are nearly identical, which implies our model does not under/over-fit, with efficient optimization and learning taken place.

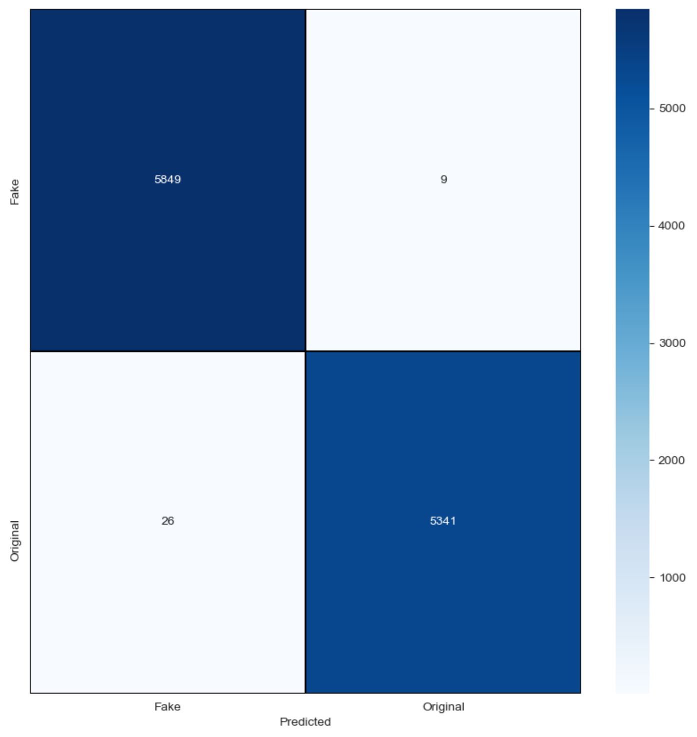
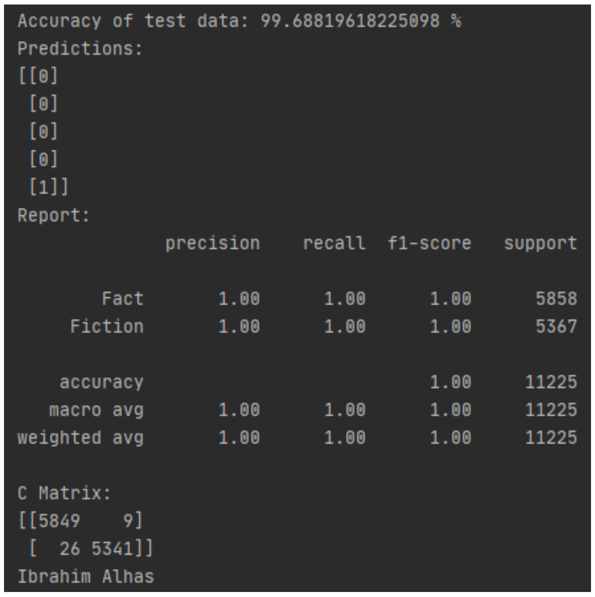


**Figure 5.7** Model 2 confusion matrix.

Looking at figure 5.7, we identify the reduced sum of misclassifications produced by Model 2, compared to Model 1. We also identify that Model 2 has had difficulty classifying false positives, just like Model 1. However, Model 2 has nearly halved the sum of misclassifications compared to Model 1. We see that Model 2 has achieved great progress classifying true negatives. Figure 5.8 demonstrates an overall highly efficient and robust neural network, with all performance metrics at their optimal values of 1.0. However, Model 2 has a lower accuracy of 0.1% compared to Model 1, at 0.97%.

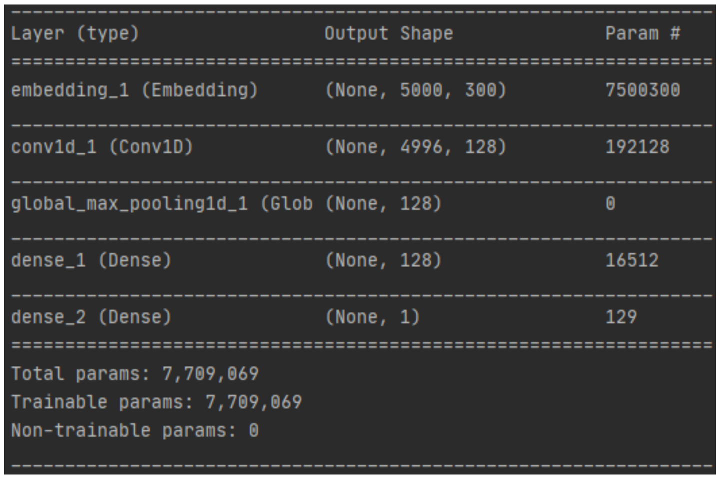


```
Accuracy of test data: 99.68819618225098 %
Predictions:
[[0]
 [0]
 [0]
 [0]
 [1]]
Report:
              precision    recall  f1-score   support

        Fact       1.00      1.00      1.00      5858
     Fiction       1.00      1.00      1.00      5367

    accuracy                           1.00     11225
   macro avg       1.00      1.00      1.00     11225
weighted avg       1.00      1.00      1.00     11225

C Matrix:
[[5849    9]
 [  26 5341]]
Ibrahim Alhas
```

**Figure 5.8** Model 2 overall performance results.

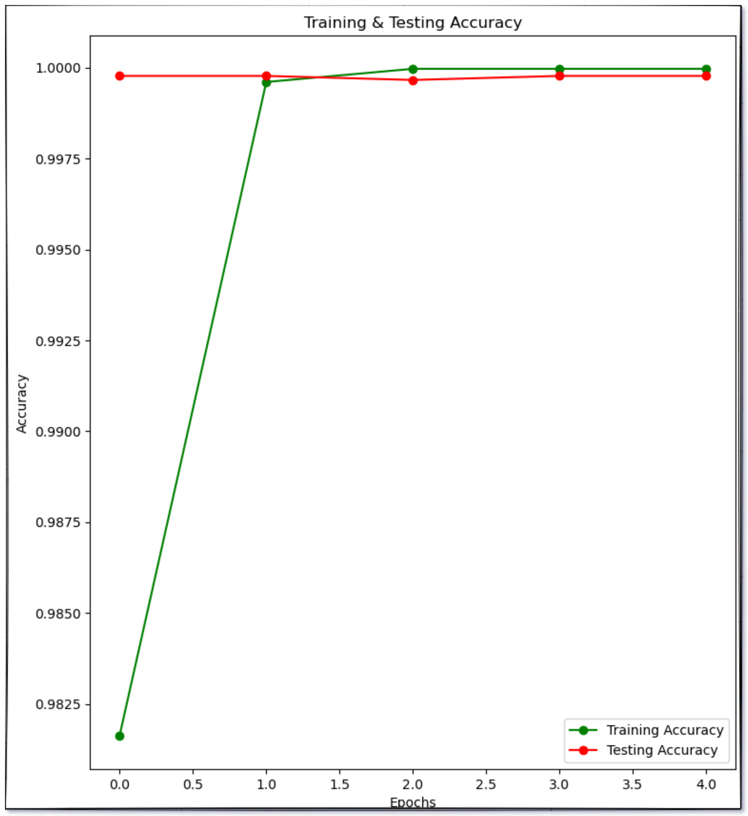
### 5.1.3 Model 3: CNN

Model 3 surprisingly achieved the highest accuracy rate of 0.99, compared to our other models and literature review work. Figure 5.9 presents Model 3 structure. Our CNN model has a slightly dense architecture, with more parameters compared to Model 2. Interestingly, there are no non-trainable weights, suggesting that all weights were optimized, which may be the cause of the CNN's exceptional results.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 5000, 300)         7500300
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 4996, 128)         192128
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 129
=================================================================
Total params: 7,709,069
Trainable params: 7,709,069
Non-trainable params: 0
_________________________________________________________________
```

**Figure 5.9** Model 3 structure and parameters.



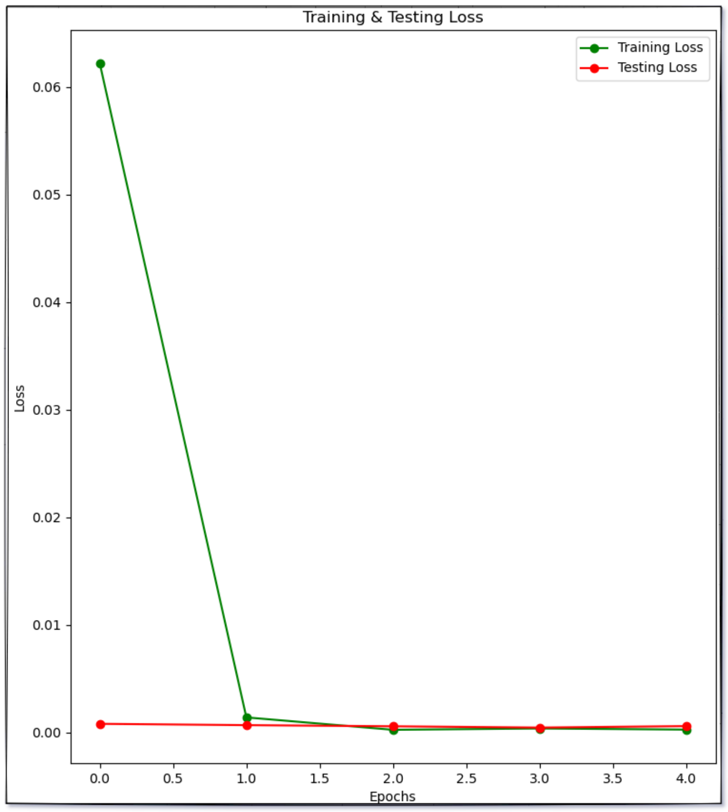


**Figure 5.10** Model 3 training and testing results.

Figure 5.10 shows our last model results. We see a very linear relationship in accuracy and loss, where both have converged after epoch two, which suggests a maximum epoch of three was sufficient. Training and testing results compliment each other, with very similar patterns. Conversely, testing accuracy and loss start at very good values, which implies our model has exceptional generalizability. We also see that during testing, we reach optimal accuracy and loss values, which demonstrate efficient optimization, compared to the other models. According to Figure 5.11, our model only misclassified a single sample from true negatives and false positives categories. Compared to Model 1 and 2, Model 3 has exceptional classification precision and recall, which Figure 5.12 confirms.

### 5.1.4 Solutions Summary

Thus, Model 3 was our most effective and robust solution to our problem, by achieving important requirements for qualification, such as high accuracy, precision, efficiency, and generalizability. Model two was a successful experimentation, with the GloVe word embedding method, which produced better precision and recall results, compared to Model 1 with Word2vec. Model 1 was our introduction to solution sphere, with less generalizability ability, compared to Model 2 and 3, but still with high accuracy and optimization performance. Comparing our solutions to literature review, we have produced solutions that out-perform

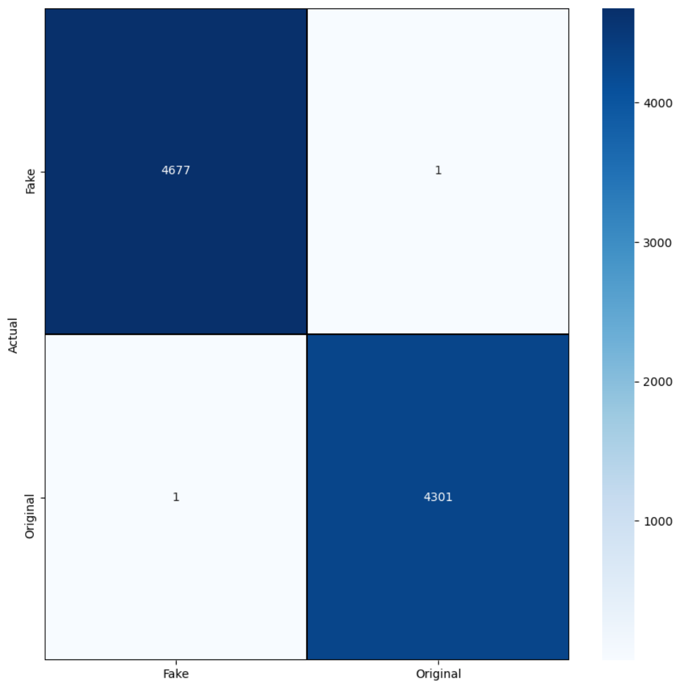
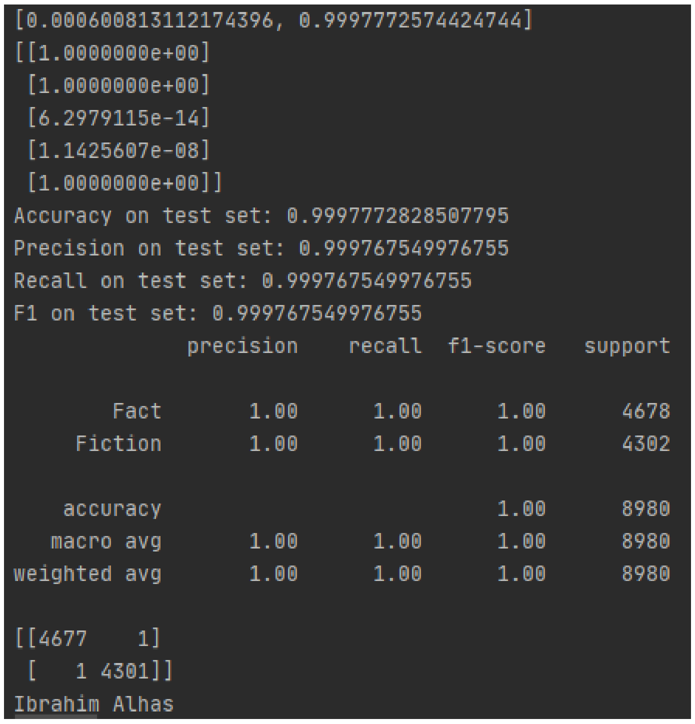


**Figure 5.11** Model 3 confusion matrix.



```
[0.000600813112174396, 0.9997772574424744]
[[1.0000000e+00]
 [1.0000000e+00]
 [6.2979115e-14]
 [1.1425607e-08]
 [1.0000000e+00]]
Accuracy on test set: 0.9997772828507795
Precision on test set: 0.999767549976755
Recall on test set: 0.999767549976755
F1 on test set: 0.999767549976755
              precision    recall  f1-score   support

        Fact       1.00      1.00      1.00      4678
     Fiction       1.00      1.00      1.00      4302

    accuracy                           1.00      8980
   macro avg       1.00      1.00      1.00      8980
weighted avg       1.00      1.00      1.00      8980

[[4677    1]
 [   1 4301]]
Ibrahim Alhas
```

**Figure 5.12** Model 3 overall performance results.

# Chapter Six
# Conclusion

We conclude our project with contributions made, project limitations, and future research.

## 6.1 Contributions

We have contributed three distinct models (two LSTM architectures, with different word embedding methods, and a CNN architecture) that each justify as a solution, with impressive performance results of 0.97%, 0.98%, and 0.99% respectively. We confirmed that our solutions also are generalisable, and can be used on other datasets of similar composition, based on our dataset's composition of tweets and news articles extracted from social media and news websites. We have also demonstrated the capability of CNNs for solving our problem, and have showed that it performs better compared to our LSTM models.

## 6.2 Project Limitations

The project suffered a few notable limitations and problems. The first problem was the limited capability of our machine for solution implementation. We lacked an adequately capable machine to design and implement our solutions, with frequent software crashes, due to insufficient memory and CPU capacities. This caused a few setbacks, such as limiting the number of epochs that we could allow our solutions to run for, as our machine crashed typically after the $6^{th}$ epoch of training, in all our models. This suggests that the model design is not the cause of the crashes, but of the machine. In addition, the ability to try different parameter combinations for each model suffered greatly because the computational time and cost of training was very long. The second limitation concerns the dataset, where the quality of our dataset lacked a good selection of features. We tried to justify this limitation with advanced parameter tuning, to increase performance. However, the size of our dataset was modest, with approximately $40,000$ instances, which countered this limitation. As a note, we researched for other datasets but failed to find any other dataset in time. We also applied for developer access into Twitter's API for the purpose of extracting tweets to form a dataset, with bespoke features and flexible size requirements, but at the time of writing this report, we did not receive any response. Therefore, we could not find a dataset with a better set of features. In addition, due to time limitations, we could not procure a dataset ourselves.

## 6.3 Future Research

As a short term goal of this project, we wish to implement our solutions on a different, unseen dataset to investigate the generalisability performance of our models. We also wish to train our models on a better dataset, with better features and quality, to improve our results further. In addition, we wish to implement different architectures and model life cycles, to search for optimal settings. In the medium term, we wish to investigate how feature engineering could be implemented to our solutions to find the best possible set of features, for optimal performances. Also, we wish to implement a combined CNN/LSTM model as we believe that the strengths of both architectures would yield exceptional performances. In the long run, we wish to apply sentiment analysis into our model, as we believe that sentiment analysis could help bolster solution confidence by identifying language-based features, such as slang, irony/sarcasm, context background, and bias.

# APPENDICES

# Appendix A

# Machine Learning Algorithm Life Cycle

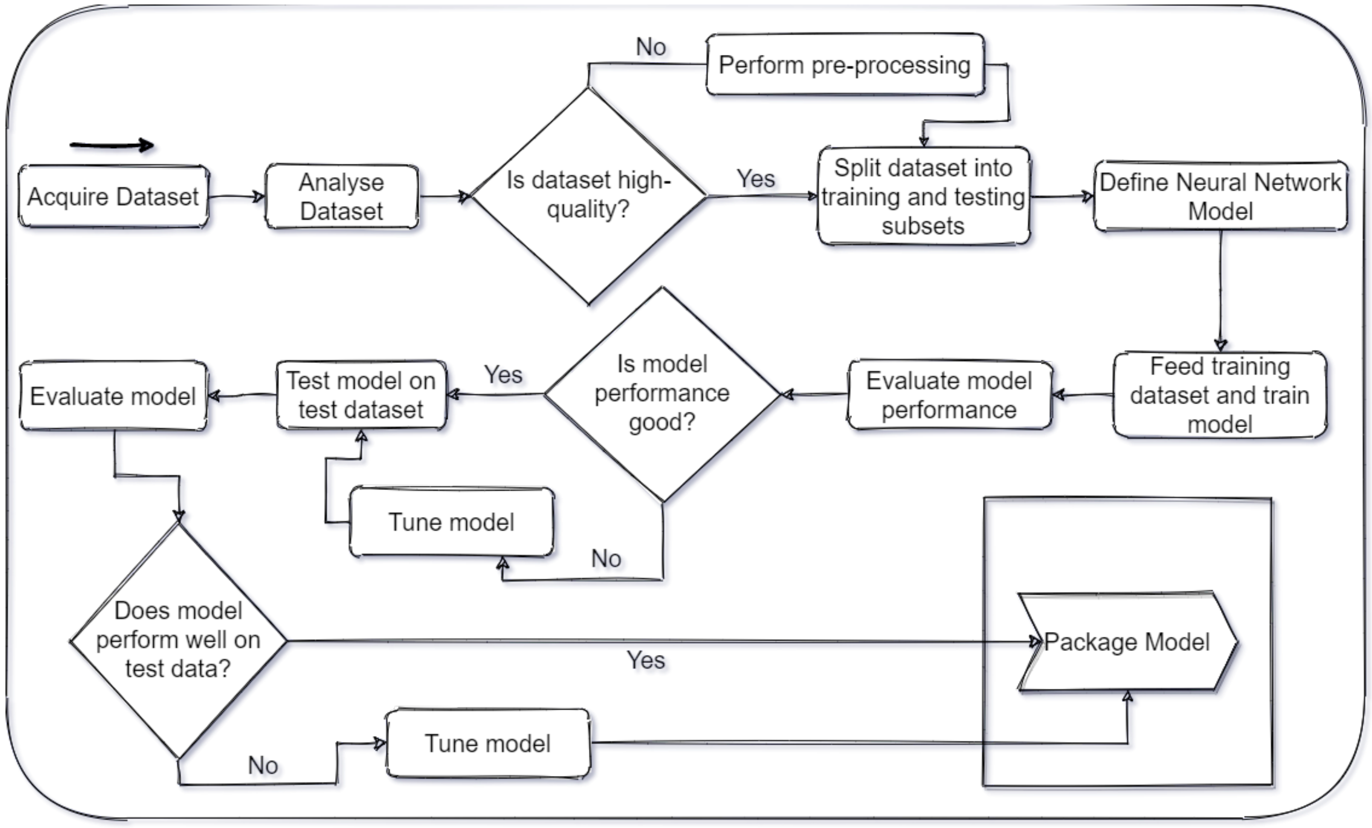

No
Perform pre-processing
Acquire Dataset
Analyse Dataset
Is dataset high-quality?
Yes
Split dataset into training and testing subsets
Define Neural Network Model
Evaluate model
Test model on test dataset
Yes
Is model performance good?
Evaluate model performance
Feed training dataset and train model
Tune model
No
Does model perform well on test data?
Package Model
Yes
Tune model
No

**Figure A.1** The ML algorithm life cycle used for every model implementation. We tried to keep the cycle of models simple and easy to follow, with an emphasis on model testing and evaluation.

# Appendix B

# Gradient Descent Pseudo-code

We describe Stochastic and Batch Gradient Descent, in pseudo-code (Vidal et al., 2017).

**Algorithm 1:** Stochastic Gradient Descent

```
initialize $w, \eta, m$ and criteria $\theta$;
$m = 0$;
while $m \leftarrow m + 1$ do
    $x^m \leftarrow$ arbitrary pattern;
    $w_{kj} \leftarrow w_{kj} + \eta \xi_k y_i$;
    $w_{ji} \leftarrow w_{ji} + \eta \xi_j x_i$;
    until: $|\Delta J\ (w(m))| = |J(w(m)) - |J(w(m-1))| < \theta$;
    return w
end
```

**Algorithm 2:** Batch Gradient Descent

```
initialize $w, \eta, e$ and criteria $\theta$;
while $e \leftarrow e + 1$ (epoch increment) do
    $e$ (epoch), $\Delta w_{ji}$, $\Delta w_{jk} \leftarrow 0$;
    while $m \leftarrow m + 1$ do
        $x^m \leftarrow$ select pattern $m$;
        $\Delta w_{kj} \leftarrow \Delta w_{kj} + \eta \xi_k y_i$ (accumulated $\Delta w_{kj}$ for all input patterns);
        $\Delta w_{ji} \leftarrow \Delta w_{ji} + \eta \xi_j x_i$ (accumulated $\Delta w_{ji}$ for all input patterns);
        until: $m = n$, where $n$ is number of individual patterns in dataset)
    end
    $w_{kj} \leftarrow w_{kj} + \Delta w_{kj}$;
    $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$;
    until: $|\Delta J\ (w(r))| = |J(w(r)) - |J(w(r-1))| < \theta$;
    return w
end
```

# Appendix C

# Source Code:

*All the source-code, including freely distributed/open code.*

```python
#   Author: Ibrahim Alhas - ID: 1533204.

#   MODEL 1:    Word2vec Embeddings (self-trained).
#   This is the final version of the model (not the base).

#   Packages and libraries used for this model.
#   ** Install these if not installed already **.
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime
from time import time
# plt.style.use('ggplot')
import seaborn as sns
import nltk
import re
import tensorboard as board
import nltk
import gensim
import tensorflow as tf
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense, Embedding, LSTM, Conv1D, MaxPool1D,
    Dropout
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score,
    recall_score, precision_score, confusion_matrix, \
    f1_score, roc_curve

#   We use STOPWORDS from NLTK package.
warnings.filterwarnings('ignore')
nltk.download('stopwords')
nltk.download('punkt')

#   Basic conData visualisation
    ------------------------------------------------------------------------------------------------

false = pd.read_csv("Fake.csv")
print(false.head())
plt.figure(figsize=(8, 5))
sns.countplot("subject", data=false)
plt.show()
```

```python

true = pd.read_csv("True.csv")
print(true.head())
sns.countplot("subject", data=true)
plt.show()

#   Cleaning the conData ------------------------------------------------------------------------------------

#   We incorporate the publishers feature from title and text instances, and place it into the dataset manually.
#   First Creating list of index that do not have publication part. We can use this as a new feature.
unknown_publishers = []
for index, row in enumerate(true.text.values):
    try:
        record = row.split(" -", maxsplit=1)
        # if no text part is present, following will give error
        print(record[1])
        # if len of piblication part is greater than 260
        # following will give error, ensuring no text having "-" in between is counted
        assert (len(record[0]) < 260)
    except:
        unknown_publishers.append(index)

#   We print the instances where publication information is absent or different.
print(true.iloc[unknown_publishers].text)

#   We want to use the publication information as a new feature.
publisher = []
tmp_text = []
for index, row in enumerate(true.text.values):
    if index in unknown_publishers:
        #   Append unknown publisher:
        tmp_text.append(row)
        publisher.append("Unknown")
        continue
    record = row.split(" -", maxsplit=1)
    publisher.append(record[0])
    tmp_text.append(record[1])

#   Replace text column with new text + add a new feature column called publisher/source.
true["publisher"] = publisher
true["text"] = tmp_text
del publisher, tmp_text, record, unknown_publishers

#   Validate that the publisher/source column has been added to the dataset.
print(true.head())

```

```python
#   Check for missing values, then drop them for both datasets.
print([index for index, text in enumerate(true.text.values) if str(text).
    strip() == ''])
true = true.drop(8970, axis=0)
fakeEmptyIndex = [index for index, text in enumerate(false.text.values) if
     str(text).strip() == '']
print(f"No of empty rows: {len(fakeEmptyIndex)}")
false.iloc[fakeEmptyIndex].tail()

# Also noticed false news have a lot of CPATIAL-CASES. Could preserve
    Cases of letters, but as we are using Google's
# pretrained word2vec vectors later on, which haswell-formed lower cases
    word. We will contert to lower case.
# The text for these rows seems to be present in title itself. Lets merge
    title and text to solve these cases.
# Looking at publication Information


#   Basic visualisation of conData, i.e. subjects such as politics.
for key, count in true.subject.value_counts().iteritems():
    print(f"{key}:\t{count}")
sns.countplot(x="subject", data=true)
plt.show()

# Pre-processing
    ---------------------------------------------------------------------------------


#   We set the labels for each conData instance, where factual = 1,
    otherwise 0.
true["class"] = 1
false["class"] = 0

#   Combining title with text columns.
true["text"] = true["title"] + " " + true["text"]
false["text"] = false["title"] + " " + false["text"]

#   Because subjects are not the same for both datasets, we have to drop
    them to avoid bias.
true = true.drop(["subject", "date", "title", "publisher"], axis=1)
false = false.drop(["subject", "date", "title"], axis=1)

# Combining both datasets.
conData = true.append(false, ignore_index=True)
del true, false

# Download stopwords and punkt.
nltk.download('stopwords')
nltk.download('punkt')

# Removing STOPWORDS, punctuations, and single-characters
    ----------------------------------------------------
y = conData["class"].values
```

```

# Converting input to acceptable format for gensim.
X = []
stop_words = set(nltk.corpus.stopwords.words("english"))
tokenizer = nltk.tokenize.RegexpTokenizer(r'\w+')
for par in conData["text"].values:
    tmp = []
    sentences = nltk.sent_tokenize(par)
    for sent in sentences:
        sent = sent.lower()
        tokens = tokenizer.tokenize(sent)
        filtered_words = [w.strip() for w in tokens if w not in stop_words
     and len(w) > 1]
        tmp.extend(filtered_words)
    X.append(tmp)
del conData

#   Vectorization using Word2vec
    ----------------------------------------------------------------------------------

#   We set the dimensions of the words with the following parameter:
embedDimensions = 100
epochs = 5

#   The function that converts the words into vectors.
w2v_model = gensim.models.Word2Vec(sentences=X, size=embedDimensions,
    window=5, min_count=1)

#   Print current vocabulary size generated via vectorization.
print(len(w2v_model.wv.vocab))

#   We pass these vectors into the LSTM model as integers instead of words
    .
#   Keras is useful for embedding words.

#   Since we cant pass words into the embedding layer directly, we have to
     tokenize the words.
#   Tokenization
    :---------------------------------------------------------------------------------

tokenizer = Tokenizer()
tokenizer.fit_on_texts(X)
X = tokenizer.texts_to_sequences(X)

#   We check the first 10 instances of vectors to validate they have been
    converted to vectors.
print(X[0][:10])

# Our word mappings are in the dictionary.
indexingWords = tokenizer.word_index
for word, num in indexingWords.items():
    print(f"{word} -> {num}")
    if num == 10:
        break

#   The variant of RNN we use here is many-to-one (as stated in the report
    ). This is because we have multiple inputs,
#   ...but only a probability of factual or fictitious for each input.

#   We keep words that are 700 words in length, and truncate anything
    above 700.
maxLength = 700

#   Apply the max length of words here:
X = pad_sequences(X, maxlen=maxLength)

#   Embedding Layer creates 1 more vector for unknown, padded words. This
    Vector is sparse with 0s.
#   Thus our vocab size increases by 1.
vocabularySize = len(tokenizer.word_index) + 1


#   We define a function that creates our weight matrix for our neural
    network.
def get_weight_matrix(model, vocab):
    # Vocabulary size + 1
    vocab_size = len(vocab) + 1
    # Definition of  weight matrix dimensions with zeros.
    weight_matrix = np.zeros((vocab_size, embedDimensions))
    #   Vocabulary stepping, where we store vectors using tokenization
    number mapping
    for word, i in vocab.items():
        weight_matrix[i] = model[word]
    return weight_matrix


# We create a matrix of mapping between word-index and vectors. We use
    this as weights in embedding layer
# Embedding layer accepts numecical-token of word and outputs
    corresponding vercor to inner layer.
# It sends vector of zeros to next layer for unknown words which would be
    tokenized to 0.
# Input length of Embedding Layer is the length of each news (700 now due
    to padding and truncating)

# Getting embedding vectors from word2vec and usings it as weights of non-
    trainable keras embedding layer
embedVectors = get_weight_matrix(w2v_model, indexingWords)

#   A custom callbacks function, which initially included tensorboard.
mycallbacks = [
    tf.keras.callbacks.ReduceLROnPlateau(monitor='val_accuracy', patience
    =2, verbose=1, factor=0.5, min_lr=0.00001),
    tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=2,
    restore_best_weights=True),  # Restoring the best
    #   ...weights will help keep the optimal weights.
```

```python
]

# Defining Neural Network
model = Sequential()
model.add(
    Embedding(vocabularySize, output_dim=embedDimensions, weights=[
    embedVectors], input_length=maxLength,
              trainable=False))
# LSTM
model.add(LSTM(units=128))
model.add(Dense(1, activation='sigmoid'))
# model.add(Dropout(0.5))
#   Compiling our model here with hyperparameters, such as loss function.
#   We implemented various optimizers but adam was the best one.
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['
    accuracy'])
del embedVectors

model.summary()

print("Model weights:")
print(model.weights)

# Train test split ratio.
X_train, X_test, y_train, y_test = train_test_split(X, y)

#   We fit and train the model, with the hyperparameters dictated below, i
    .e. epochs.
history = model.fit(X_train, y_train, validation_split=0.2, epochs=epochs,
     batch_size=256, shuffle=True,
                    callbacks=mycallbacks)

#   Produce a figure, for every epoch, and show performance metrics.
epochs = [i for i in range(5)]
fig, ax = plt.subplots(1, 2)
train_acc = history.history['accuracy']
train_loss = history.history['loss']
val_acc = history.history['val_accuracy']
val_loss = history.history['val_loss']
fig.set_size_inches(20, 10)

ax[0].plot(epochs, train_acc, 'go-', label='Training Accuracy')
ax[0].plot(epochs, val_acc, 'ro-', label='Testing Accuracy')
ax[0].set_title('Training & Testing Accuracy')
ax[0].legend()
ax[0].set_xlabel("Epochs")
ax[0].set_ylabel("Accuracy")

ax[1].plot(epochs, train_loss, 'go-', label='Training Loss')
ax[1].plot(epochs, val_loss, 'ro-', label='Testing Loss')
ax[1].set_title('Training & Testing Loss')
ax[1].legend()
ax[1].set_xlabel("Epochs")
```

```
ax[1].set_ylabel("Loss")
plt.show()

#   We evaluate our model
    .-------------------------------------------------------------------------------------
print("Evaluation:")
print(model.evaluate(X_test, y_test))

#   We predict a few instances (up to 5). For all instances with
    probability 0.5 and over, it is fiction; else factual.
pred = (model.predict(X_test) >= 0.5).astype("int")
print(pred[:5])

binaryPred = []
for i in pred:
    if i >= 0.5:
        binaryPred.append(1)
    else:
        binaryPred.append(0)

#   We print performance metrics.
print('Accuracy on test set:', accuracy_score(binaryPred, y_test))
print('Precision on test set:', precision_score(binaryPred, y_test))
print('Recall on test set:', recall_score(binaryPred, y_test))
print('F1 on test set:', f1_score(binaryPred, y_test))

#   We print the confusion matrix.
print("Report:")
print(classification_report(y_test, pred, target_names=['Fact', 'Fiction'
    ]))

print("C Matrix:")
cm = confusion_matrix(y_test, pred)
print(cm)

print("Ibrahim Alhas")

cmm = pd.DataFrame(cm, index=['Fake', 'Original'], columns=['Fake', '
    Original'])
plt.figure(figsize=(10, 10))
sns.heatmap(cm, cmap="Blues", linecolor='black', linewidth=1, annot=True,
    fmt='', xticklabels=['Fake', 'Original'],
            yticklabels=['Fake', 'Original'])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

#   End---------------------------------------------------
```

**Listing C.1** Solution 1 Code

```
#   Author: Ibrahim Alhas - ID: 1533204.

#   MODEL 2:    GloVe Embeddings with pre-trained vectors (included in
    folder).
#   This is the final version of the model (not the base).

#   Packages and libraries used for this model.
#   ** Install these if not installed already **.
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import datetime
from keras.callbacks import TensorBoard
from time import time
from tensorflow.python.keras.callbacks import TensorBoard
import nltk
from sklearn.preprocessing import LabelBinarizer
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize, sent_tokenize
from bs4 import BeautifulSoup
import re, string, unicodedata
from tensorflow.keras.preprocessing import text, sequence  # tensorflow
from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score, f1_score, roc_curve
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import TensorBoard
from string import punctuation
from nltk import pos_tag
from nltk.corpus import wordnet
from tensorflow import keras as tff
# import keras
from sklearn.feature_extraction.text import CountVectorizer
from collections import Counter
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding, LSTM, Dropout
from keras.callbacks import ReduceLROnPlateau
import tensorflow as tf

#   Importing dataset. Note that the dataset is divided into true (factual
    ) and fake (fictitious) subsets.
#   ** Make sure that the directory is correct, otherwise it will give a "
    no such file or directory" error **.
true = pd.read_csv("True.csv")
false = pd.read_csv("Fake.csv")

#   Basic data visualisation
    ----------------------------------------------------------------------------------

#   We see that the title column is from news articles, and the text
    column forms the twitter tweet extracts.
print(true.head())
print(false.head())
```

```

#   We set the labels for each data instance, where factual = 1, otherwise
     0.
true['category'] = 1
false['category'] = 0

#   We merge the two divided datasets (true and fake) into a singular
    dataset.
df = pd.concat([true, false])

#   We can see that the dataset is a bit unbalanced, with more instances
    for fiction.
print(sns.set_style("darkgrid"))
print(sns.countplot(df.category))

#   Checking for missing values (i.e. Nan).
print(df.isna().sum())
df.title.count()
df.subject.value_counts()

#   We merge the columns title and text into one column.
#   We also delete the columns subject, date and title.
df['text'] = df['text'] + " " + df['title']
del df['title']
del df['subject']
del df['date']

#   We use STOPWORDS for this model.
stop = set(stopwords.words('english'))
punctuation = list(string.punctuation)
print(stop.update(punctuation))

#   DATA CLEANING
    ------------------------------------------------------------------------------------

#   We incorporate the publishers feature from title and text instances,
    and place it into the dataset manually.
#   First Creating list of index that do not have publication part. We can
     use this as a new feature.
unknown_publishers = []
for index, row in enumerate(true.text.values):
    try:
        record = row.split(" -", maxsplit=1)
        # if no text part is present, following will give error
        print(record[1])
        # if len of piblication part is greater than 260
        # following will give error, ensuring no text having "-" in
    between is counted
        assert (len(record[0]) < 260)
     except:
        unknown_publishers.append(index)

#   We print the instances where publication information is absent or
```

```
    different.
print(true.iloc[unknown_publishers].text)

#   We want to use the publication information as a new feature.
publisher = []
tmp_text = []
for index, row in enumerate(true.text.values):
    if index in unknown_publishers:
        #   Append unknown publisher:
        tmp_text.append(row)
        publisher.append("Unknown")
        continue
    record = row.split(" -", maxsplit=1)
    publisher.append(record[0])
    tmp_text.append(record[1])

#   Replace text column with new text + add a new feature column called
    publisher/source.
true["publisher"] = publisher
true["text"] = tmp_text
del publisher, tmp_text, record, unknown_publishers
# -
#   Validate that the publisher/source column has been added to the
    dataset.
print(true.head())

#   Check for missing values, then drop them for both datasets.
print([index for index, text in enumerate(true.text.values) if str(text).
    strip() == ''])
true = true.drop(8970, axis=0)
fakeEmptyIndex = [index for index, text in enumerate(false.text.values) if
     str(text).strip() == '']
print(f"No of empty rows: {len(fakeEmptyIndex)}")
false.iloc[fakeEmptyIndex].tail()


def strip_html(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()


# Removing the square brackets
def remove_between_square_brackets(text):
    return re.sub('\[[^]]*\]', '', text)


# Removing URL's
def remove_between_square_brackets(text):
    return re.sub(r'http\S+', '', text)


# Removing the stopwords from text
def remove_stopwords(text):
    final_text = []
    for i in text.split():
        if i.strip().lower() not in stop:
            final_text.append(i.strip())
    return " ".join(final_text)


#   Remove noise for better performance.
def denoise_text(text):
    text = strip_html(text)
    text = remove_between_square_brackets(text)
    text = remove_stopwords(text)
    return text


#   We apply the denoise function on our column in the dataset.
df['text'] = df['text'].apply(denoise_text)

#   Text distribution is different for the classes: 2500 characters is
    most common in true category,
#   ...while around 5000 characters in common in fake text category.
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 8))
text_len = df[df['category'] == 1]['text'].str.len()
ax1.hist(text_len, color='red')
ax1.set_title('Original text')
text_len = df[df['category'] == 0]['text'].str.len()
ax2.hist(text_len, color='blue')
ax2.set_title('Fake text')
fig.suptitle('Characters in texts')
plt.show()

#   N words in each class.
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 8))
text_len = df[df['category'] == 1]['text'].str.split().map(lambda x: len(x
    ))
ax1.hist(text_len, color='green')
ax1.set_title('Original text')
text_len = df[df['category'] == 0]['text'].str.split().map(lambda x: len(x
    ))
ax2.hist(text_len, color='black')
ax2.set_title('Fake text')
fig.suptitle('Words in texts')
plt.show()

#   Average words for each label.
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
word = df[df['category'] == 1]['text'].str.split().apply(lambda x: [len(i)
     for i in x])
sns.distplot(word.map(lambda x: np.mean(x)), ax=ax1, color='blue')
ax1.set_title('Original')
word = df[df['category'] == 0]['text'].str.split().apply(lambda x: [len(i)
     for i in x])
sns.distplot(word.map(lambda x: np.mean(x)), ax=ax2, color='black')
```

```
ax2.set_title('Fake')
fig.suptitle('Average word length for each text')


def get_corpus(text):
    words = []
    for i in text:
        for j in i.split():
            words.append(j.strip())
    return words


#   Print the first 5 in the dataset.
corpus = get_corpus(df.text)
print(corpus[:5])

#   Find the most common words (upto 10).
counter = Counter(corpus)
mostCommonWords = counter.most_common(10)
mostCommonWords = dict(mostCommonWords)
print(mostCommonWords)


def get_top_text_ngrams(corpus, n, g):
    vec = CountVectorizer(ngram_range=(g, g)).fit(corpus)
    bag_of_words = vec.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in vec.
    vocabulary_.items()]
    words_freq = sorted(words_freq, key=lambda x: x[1], reverse=True)
    return words_freq[:n]


#   We split the data into training and splitting, as presented in the
    report.
#   We found that for this model, the train/test split ratio was: 2:3.
x_train, x_test, y_train, y_test = train_test_split(df.text, df.category,
    random_state=0)

#   Parameters for our upcoming model
    --------------------------------------------------------------------------------

maxFeatures = 10000
maxLength = 300

#   We tokenize the words (representing every word with a vectorized
    number).
tokenizer = text.Tokenizer(num_words=maxFeatures)
tokenizer.fit_on_texts(x_train)
tokenized_train = tokenizer.texts_to_sequences(x_train)
x_train = sequence.pad_sequences(tokenized_train, maxlen=maxLength)
tokenized_test = tokenizer.texts_to_sequences(x_test)
X_test = sequence.pad_sequences(tokenized_test, maxlen=maxLength)
```

```

#   For the embedding, we use GloVe. We believe that using this format
    will produce in better performances.
#   We use a pre-trained file to forward into our model, for better
    performance.

#   Tutorial for glove embeddings: https://nlp.stanford.edu/pubs/glove.pdf
EMBEDDING_FILE = 'glove.twitter.27B.200d.txt'


def get_coefs(word, *arr):
    return word, np.asarray(arr, dtype='float32')


#   Embedding: https://medium.com/analytics-vidhya/basics-of-using-pre-
    trained-glove-vectors-in-python-d38905f356db
embedIndexing = dict(get_coefs(*o.rstrip().rsplit(' ')) for o in open(
    EMBEDDING_FILE, encoding="utf8"))
# embedIndexing = dict(get_coefs(*o.rstrip().rsplit(' ')) for o in open(
    EMBEDDING_FILE, encoding="utf8"))

embeddingsAll = np.stack(embedIndexing.values())
emb_mean, emb_std = embeddingsAll.mean(), embeddingsAll.std()
sizeOfEmbed = embeddingsAll.shape[1]

word_index = tokenizer.word_index
nb_words = min(maxFeatures, len(word_index))
# change below line if computing normal stats is too slow
embedding_matrix = embedding_matrix = np.random.normal(emb_mean, emb_std,
    (nb_words, sizeOfEmbed))
for word, i in word_index.items():
    if i >= maxFeatures: continue
    embedding_vector = embedIndexing.get(word)
    if embedding_vector is not None: embedding_matrix[i] =
    embedding_vector

#   Model parameters that were fine-tuned.
batch_size = 256
epochs = 1
sizeOfEmbed = 200
log_dir = "logs\\model\\"
#   A custom callbacks function, which initially included tensorboard.
mycallbacks = [
    tf.keras.callbacks.ReduceLROnPlateau(monitor='val_accuracy', patience
    =2, verbose=1, factor=0.5, min_lr=0.00001),
    tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=2,
    restore_best_weights=True),  # Restoring the best
    #   ...weights will help keep the optimal weights.
    #   tf.keras.callbacks.TensorBoard(log_dir="./logs"),  # NEWLY ADDED -
     CHECK.
    tf.keras.callbacks.TensorBoard(log_dir=log_dir.format(time())),  #
    NEWLY ADDED - CHECK.
    #   tensorboard --logdir logs --> to check tensorboard feedback.
]

#   We define the neural network
    --------------------------------------------------------------------------------

model = Sequential()
#   Embedding layer (which is un-trainable).
model.add(
    Embedding(maxFeatures, output_dim=sizeOfEmbed, weights=[
    embedding_matrix], input_length=maxLength, trainable=False))
model.add(LSTM(units=128, return_sequences=True, recurrent_dropout=0.25,
    dropout=0.25))
model.add(LSTM(units=64, recurrent_dropout=0.1, dropout=0.1))
model.add(Dense(units=32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer=tf.keras.optimizers.Adam(lr=0.01), loss='
    binary_crossentropy', metrics=['accuracy'])

model.summary()

print("Model weights:")
print(model.weights)

#   Track training history, for visualisations.
history = model.fit(x_train, y_train, batch_size=batch_size,
    validation_data=(X_test, y_test), epochs=epochs,
                    callbacks=mycallbacks)

#   We evaluate our model by predicting a few instances from our test data
     (the first 5)--------------------------------
print("Stats:")
print("Accuracy of train data:", model.evaluate(x_train, y_train)[1] *
    100, "%")
print("Accuracy of test data:", model.evaluate(X_test, y_test)[1] * 100, "
    %")

#   Produce a figure, for every epoch, and show performance metrics.
epochs = [i for i in range(1)]
fig, ax = plt.subplots(1, 2)
train_acc = history.history['accuracy']
train_loss = history.history['loss']
val_acc = history.history['val_accuracy']
val_loss = history.history['val_loss']
fig.set_size_inches(20, 10)

ax[0].plot(epochs, train_acc, 'go-', label='Training Accuracy')
ax[0].plot(epochs, val_acc, 'ro-', label='Testing Accuracy')
ax[0].set_title('Training & Testing Accuracy')
ax[0].legend()
ax[0].set_xlabel("Epochs")
ax[0].set_ylabel("Accuracy")

```

```
ax[1].plot(epochs, train_loss, 'go-', label='Training Loss')
ax[1].plot(epochs, val_loss, 'ro-', label='Testing Loss')
ax[1].set_title('Training & Testing Loss')
ax[1].legend()
ax[1].set_xlabel("Epochs")
ax[1].set_ylabel("Loss")
plt.show()

#   We predict a few instances (up to 5). For all instances with
    probability 0.5 and over, it is fiction; else factual.
print("Predictions:")
pred = model.predict_classes(X_test)
print(pred[:5])

#   We print the performance metrics.
print("Report:")
print(classification_report(y_test, pred, target_names=['Fact', 'Fiction'
    ]))

#   We print the confusion matrix.
print("C Matrix:")
cm = confusion_matrix(y_test, pred)
print(cm)

print("Ibrahim Alhas")

cmm = pd.DataFrame(cm, index=['Fake', 'Original'], columns=['Fake', '
    Original'])
plt.figure(figsize=(10, 10))
sns.heatmap(cm, cmap="Blues", linecolor='black', linewidth=1, annot=True,
    fmt='', xticklabels=['Fake', 'Original'],
            yticklabels=['Fake', 'Original'])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

#   End---------------------------------------------------
# print('Accuracy on testing set:', accuracy_score(binary_predictions,
    y_test))
# print('Precision on testing set:', precision_score(binary_predictions,
    y_test))
# print('Recall on testing set:', recall_score(binary_predictions, y_test)
    )
# print('F1 on testing set:', f1_score(binary_predictions, y_val))

#   End---------------------------------------------------
```

**Listing C.2** Solution 2 Code

```
#   Author: Ibrahim Alhas - ID: 1533204.

#   MODEL 3:    CNN with built-in tensorflow tokenizer.
#   This is the final version of the model (not the base).

#   Packages and libraries used for this model.
#   ** Install these if not installed already **.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime
from time import time
import re
from sklearn.metrics import accuracy_score, confusion_matrix,
    precision_score, recall_score, f1_score, roc_curve, \
     classification_report
from tensorflow import keras
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras import layers
from keras.models import Sequential
from sklearn.model_selection import train_test_split, cross_validate
import tensorflow as tf
import seaborn as sns
import warnings
import keras
from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Flatten, Activation,
    BatchNormalization
from keras.layers.noise import GaussianNoise
from keras.layers import Conv2D, MaxPooling2D

warnings.filterwarnings('ignore')
# plt.style.use('ggplot')

#   Basic data visualisation and analysis
    ---------------------------------------------------------------------------------

#   We see that the title column is from news articles, and the text
    column forms the twitter tweet extracts.
true = pd.read_csv('True.csv')
false = pd.read_csv('Fake.csv')

#   We drop the columns we do not need. See chapter 3, model CNN for more
    details.
true = true.drop('title', axis=1)
true = true.drop('subject', axis=1)
true = true.drop('date', axis=1)
false = false.drop('title', axis=1)
false = false.drop('subject', axis=1)
false = false.drop('date', axis=1)

#   We set the labels for each data instance, where factual = 1, otherwise
     0.
false['label'] = 0
true['label'] = 1

```

```python
#   We merge the two divided datasets (true and fake) into a singular dataset.
data = pd.concat([true, false], ignore_index=True)
texts = data['text']
labels = data['label']
x = texts
y = labels

#   We incorporate the publishers feature from title and text instances, and place it into the dataset manually.
#   First Creating list of index that do not have publication part. We can use this as a new feature.
unknown_publishers = []
for index, row in enumerate(true.text.values):
    try:
        record = row.split(" -", maxsplit=1)
        # if no text part is present, following will give error
        print(record[1])
        # if len of piblication part is greater than 260
        # following will give error, ensuring no text having "-" in between is counted
        assert (len(record[0]) < 260)
    except:
        unknown_publishers.append(index)

#   We print the instances where publication information is absent or different.
print(true.iloc[unknown_publishers].text)

#   We want to use the publication information as a new feature.
publisher = []
tmp_text = []
for index, row in enumerate(true.text.values):
    if index in unknown_publishers:
        #   Append unknown publisher:
        tmp_text.append(row)
        publisher.append("Unknown")
        continue
    record = row.split(" -", maxsplit=1)
    publisher.append(record[0])
    tmp_text.append(record[1])

#   Replace text column with new text + add a new feature column called publisher/source.
true["publisher"] = publisher
true["text"] = tmp_text
del publisher, tmp_text, record, unknown_publishers

#   Validate that the publisher/source column has been added to the dataset.
print(true.head())

#   Check for missing values, then drop them for both datasets.
```

```
print([index for index, text in enumerate(true.text.values) if str(text).
    strip() == ''])
true = true.drop(8970, axis=0)
fakeEmptyIndex = [index for index, text in enumerate(false.text.values) if
     str(text).strip() == '']
print(f"No of empty rows: {len(fakeEmptyIndex)}")
false.iloc[fakeEmptyIndex].tail()
# -
#   For CNNs, we have to vectorize the text into 2d integers (tensors).
MAX_SEQUENCE_LENGTH = 5000
MAX_NUM_WORDS = 25000
EMBEDDING_DIM = 300
TEST_SPLIT = 0.2
epochs = 1

#   We tokenize the text, just like all other models
    --------------------------------------------------------------------
tokenizer = Tokenizer(num_words=MAX_NUM_WORDS)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
word_index = tokenizer.word_index
num_words = min(MAX_NUM_WORDS, len(word_index)) + 1
data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH, padding='pre',
     truncating='pre')

#   Print the total number of tokens:
print('Found %s tokens.' % len(word_index))

#   We partition our dataset into train/test.
x_train, x_val, y_train, y_val = train_test_split(data, labels.apply(
    lambda x: 0 if x == 0 else 1),
                                                  test_size=TEST_SPLIT)
log_dir = "logs\\model\\"
#   A custom callbacks function, which initially included tensorboard.
mycallbacks = [
    tf.keras.callbacks.ReduceLROnPlateau(monitor='val_accuracy', patience
    =2, verbose=1, factor=0.5, min_lr=0.00001),
    tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=2,
    restore_best_weights=True),  # Restoring the best
    #   ...weights will help keep the optimal weights.
    #   tf.keras.callbacks.TensorBoard(log_dir="./logs"),  # NEWLY ADDED -
     CHECK.
    #   tf.keras.callbacks.TensorBoard(log_dir=log_dir.format(time())),  #
     NEWLY ADDED - CHECK.
    #   tensorboard --logdir logs --> to check tensorboard feedback.
]

#   Parameters for our model. We experimented with some combinations and
    settled on this configuration------------------
model = Sequential(
    [
        #   Word/sequence processing:
        layers.Embedding(num_words, EMBEDDING_DIM, input_length=
    MAX_SEQUENCE_LENGTH, trainable=True),
        #   The layers:
        layers.Conv1D(128, 5, activation='relu'),
        layers.GlobalMaxPooling1D(),
        #   We classify our model here:
        layers.Dense(128, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])

#   We compile our model and run, with the loss function crossentropy, and
    optimizer rmsprop (we experimented with adam,
#   ...but rmsprop produced better results).
model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['
    accuracy'])

model.summary()

print("Model weights:")
print(model.weights)

# tensorboard_callback = keras.callbacks.TensorBoard(log_dir="./logs")
history = model.fit(x_train, y_train, batch_size=256, epochs=epochs,
    validation_data=(x_val, y_val),
                    callbacks=mycallbacks)

#   Produce a figure, for every epoch, and show performance metrics.
epochs = [i for i in range(1)]
fig, ax = plt.subplots(1, 2)
train_acc = history.history['accuracy']
train_loss = history.history['loss']
val_acc = history.history['val_accuracy']
val_loss = history.history['val_loss']
fig.set_size_inches(20, 10)

ax[0].plot(epochs, train_acc, 'go-', label='Training Accuracy')
ax[0].plot(epochs, val_acc, 'ro-', label='Testing Accuracy')
ax[0].set_title('Training & Testing Accuracy')
ax[0].legend()
ax[0].set_xlabel("Epochs")
ax[0].set_ylabel("Accuracy")

ax[1].plot(epochs, train_loss, 'go-', label='Training Loss')
ax[1].plot(epochs, val_loss, 'ro-', label='Testing Loss')
ax[1].set_title('Training & Testing Loss')
ax[1].legend()
ax[1].set_xlabel("Epochs")
ax[1].set_ylabel("Loss")
plt.show()

'''
history_dict = history.history
acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
```

```
loss = history_dict['loss']
val_loss = history_dict['val_loss']
epochs = history.epoch

plt.figure(figsize=(12, 9))
plt.plot(epochs, loss, 'r', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss', size=20)
plt.xlabel('Epochs', size=20)
plt.ylabel('Loss', size=20)
plt.legend(prop={'size': 20})
plt.show()

plt.figure(figsize=(12, 9))
plt.plot(epochs, acc, 'g', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy', size=20)
plt.xlabel('Epochs', size=20)
plt.ylabel('Accuracy', size=20)
plt.legend(prop={'size': 20})
plt.ylim((0.5, 1))
plt.show()
'''
#   We evaluate our model by predicting a few instances from our test data
    (the first 5)-------------------------------
print("Evaluation:")
print(model.evaluate(x_val, y_val))

#   We predict a few instances (up to 5).
pred = model.predict(x_val)
print(pred[:5])

binary_predictions = []
for i in pred:
    if i >= 0.5:
        binary_predictions.append(1)
    else:
        binary_predictions.append(0)

#   We print performance metrics:
print('Accuracy on test set:', accuracy_score(binary_predictions, y_val))
print('Precision on test set:', precision_score(binary_predictions, y_val)
    )
print('Recall on test set:', recall_score(binary_predictions, y_val))
print('F1 on test set:', f1_score(binary_predictions, y_val))

#   We print the classification report (as an extra):
print(classification_report(y_val, pred.round(), target_names=['Fact', '
    Fiction']))

#   We print the confusion matrix.
cmm = confusion_matrix(y_val, pred.round())
print(cmm)
```

```python

print("Ibrahim Alhas")

cmm = pd.DataFrame(cmm, index=['Fake', 'Original'], columns=['Fake', '
    Original'])
plt.figure(figsize=(10, 10))
sns.heatmap(cmm, cmap="Blues", linecolor='black', linewidth=1, annot=True,
     fmt='', xticklabels=['Fake', 'Original'],
            yticklabels=['Fake', 'Original'])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()

#   End---------------------------------------------------
```

**Listing C.3** Solution 3 Code

# References

Agarap, Abien Fred (2018). “Deep learning using rectified linear units (relu)”. In: *arXiv preprint arXiv:1803.08375*.

Agostinelli, Forest et al. (2014). “Learning activation functions to improve deep neural networks”. In: *arXiv preprint arXiv:1412.6830*.

Alpaydin, Ethem (2020). *Introduction to machine learning*. MIT press.

Amari, Shun-ichi (1993a). “Backpropagation and stochastic gradient descent method”. In: *Neurocomputing* 5.4, pp. 185–196. ISSN: 0925-2312. DOI: https://doi.org/10.1016/0925-2312(93)90006-O. URL: http://www.sciencedirect.com/science/article/pii/092523129390006O.

— (1993b). “Backpropagation and stochastic gradient descent method”. In: *Neurocomputing* 5.4-5, pp. 185–196.

Anthony, Martin and Peter L Bartlett (2009a). *Neural network learning: Theoretical foundations*. cambridge university press.

— (2009b). *Neural network learning: Theoretical foundations*. cambridge university press.

Bengio, Y., P. Simard, and P. Frasconi (1994). “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE Transactions on Neural Networks* 5.2, pp. 157–166.

Bosc, Tom, Elena Cabrio, and Serena Villata (May 2016). “DART: a Dataset of Arguments and their Relations on Twitter”. In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC’16)*. Portorož, Slovenia: European Language Resources Association (ELRA), pp. 1258–1263. URL: https://www.aclweb.org/anthology/L16-1200.

BrainMaps.org (n.d.). URL: http://brainmaps.org/SiteImages/smi32-pic.jpg.

Britz, Denny (2015). “Understanding convolutional neural networks for NLP”. In: *URL: http://www. wildml. com/2015/11/understanding-convolutional-neuralnetworks-for-nlp/(visited on 11/07/2015)*.

Carlini, Nicholas and David Wagner (2017). “Towards evaluating the robustness of neural networks”. In: *2017 ieee symposium on security and privacy (sp)*. IEEE, pp. 39–57.

Carrillo-de-Albornoz, Jorge et al. (2019). “Beyond opinion classification: Extracting facts, opinions and experiences from health forums”. In: *PloS one* 14.1, e0209961.

Conduct, BCS Codes of and Practice (n.d.). URL: https://www.cs.uct.ac.za/mit_notes/ethics/htmls/ch04s04.html.

Dumitru, Ciobanu and Vasilescu Maria (2013). “Advantages and Disadvantages of Using Neural Networks for Predictions.” In: *Ovidius University Annals, Series Economic Sciences* 13.1.

Ertam, Fatih and Galip Aydın (2017). “Data classification with deep learning using Tensorflow”. In: *2017 international conference on computer science and engineering (UBMK)*. IEEE, pp. 755–758.

Finn, Aidan, Nicholas Kushmerick, and Barry Smyth (2001). “Fact or Fiction: Content Classification for Digital Libraries.” In: *DELOS*.

Grosse, Roger (2017). “Lecture 15: Exploding and vanishing gradients”. In: *University of Toronto Computer Science*.

H.K.Lam (2020). *Lecture notes in 7CCSMPNN Pattern Recognition, Neural Networks and Deep Learning, King’s College London*.

Halasz, Peter (n.d.). *Twitter pre-trained word vectors*. URL: https://zenodo.org/record/3237458#.X1txBoDQiUl.

Hayman, Samantha (1999). “The mcculloch-pitts model”. In: *IJCNN’99. International Joint Conference on Neural Networks. Proceedings (Cat. No. 99CH36339)*. Vol. 6. IEEE, pp. 4438–4439.

Hinkelmann, Knut, Sajjad Ahmed, and Flavio Corradini (2019). “Combining Machine Learning with Knowledge Engineering to detect Fake News in Social Networks - A Survey”. In: *AAAI Spring Symposium: Combining Machine Learning with Knowledge Engineering*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “LSTM can solve hard long time lag problems”. In: *Advances in neural information processing systems*, pp. 473–479.

Jain, Anil K, Jianchang Mao, and K Moidin Mohiuddin (1996a). “Artificial neural networks: A tutorial”. In: *Computer* 29.3, pp. 31–44.

— (1996b). “Artificial neural networks: A tutorial”. In: *Computer* 29.3, pp. 31–44.

Keogh, Eamonn and Abdullah Mueen (2017). “Curse of Dimensionality”. In: *Encyclopedia of Machine Learning and Data Mining*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, pp. 314–315. ISBN: 978-1-4899-7687-1. DOI: 10.1007/978-1-4899-7687-1_192. URL: https://doi.org/10.1007/978-1-4899-7687-1_192.

Kim, Yoon (2014). “Convolutional Neural Networks for Sentence Classification”. In: *CoRR* abs/1408.5882. arXiv: 1408.5882. URL: http://arxiv.org/abs/1408.5882.

Kingma, Diederik P. and Jimmy Ba (2014a). *Adam: A Method for Stochastic Optimization.* arXiv: 1412.6980 `[cs.LG]`.

— (2014b). “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980*.

Kuo, C-C Jay (2016). “Understanding convolutional neural networks with a mathematical model”. In: *Journal of Visual Communication and Image Representation* 41, pp. 406–413.

Levine, Timothy R (2014). “Truth-default theory (TDT) a theory of human deception and deception detection”. In: *Journal of Language and Social Psychology* 33.4, pp. 378–392.

Lillicrap, Timothy P and Adam Santoro (2019). “Backpropagation through time and the brain”. In: *Current opinion in neurobiology* 55, pp. 82–89.

Lu, Lu et al. (2019). “Dying relu and initialization: Theory and numerical examples”. In: *arXiv preprint arXiv:1903.06733*.

Lu, Yingjing (2019). “The Level Weighted Structural Similarity Loss: A Step Away from the MSE”. In: *AAAI*.

Lydia, Agnes and Sagayaraj Francis (2019). “Adagrad—An optimizer for stochastic gradient descent”. In: *Int. J. Inf. Comput. Sci.* 6.5.

Ma, Long and Yanqing Zhang (2015). “Using Word2Vec to process big text data”. In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, pp. 2895–2897.

Marlin, Todd (n.d.). *How to identify fact from fiction during the COVID-19 pandemic and beyond.* URL: https://www.ey.com/en_gl/forensic-integrity-services/how-to-identify-fact-from-fiction-during-the-covid-19-pandemic-and-beyond.

McCulloch, Warren S and Walter Pitts (1943). “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Medsker, Larry R and LC Jain (2001). “Recurrent neural networks”. In: *Design and Applications* 5.

Mian, Areeb and Shujhat Khan (Mar. 2020a). “Coronavirus: the spread of misinformation”. In: *BMC Medicine.* DOI: 10.1186/s12916-020-01556-3.

— (2020b). “Coronavirus: the spread of misinformation”. In: *BMC medicine* 18.1, pp. 1–2.

Moolayil, Jojo, Jojo Moolayil, and Suresh John (2019). *Learn Keras for Deep Neural Networks.* Springer.

Mühlenbein, Heinz (1990). “Limitations of multi-layer perceptron networks-steps towards genetic neural networks”. In: *Parallel Computing* 14.3, pp. 249–260.

Murphy, Charlie, Patrick Gray, and Gordon Stewart (2017). "Verified perceptron convergence theorem". In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*.

Nadkarni, Prakash M, Lucila Ohno-Machado, and Wendy W Chapman (2011). "Natural language processing: an introduction". In: *Journal of the American Medical Informatics Association* 18.5, pp. 544–551.

Nasr, G. E., E. Badr, and C. Joun (2002). "Cross Entropy Error Function in Neural Networks: Forecasting Gasoline Demand". In: *FLAIRS Conference*.

O'Shea, Keiron and Ryan Nash (2015). "An introduction to convolutional neural networks". In: *arXiv preprint arXiv:1511.08458*.

Pedregosa, Fabian et al. (2011). "Scikit-learn: Machine learning in Python". In: *the Journal of machine Learning research* 12, pp. 2825–2830.

Peltarion (n.d.). *Binary CrossEntropy*. URL: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/binary-crossentropy.

Pennington, Jeffrey, Richard Socher, and Christopher D Manning (2014a). "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543.

— (2014b). "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543.

Priddy, Kevin L and Paul E Keller (2005). *Artificial neural networks: an introduction*. Vol. 68. SPIE press.

Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain". In: *Psychological Review*, pp. 65–386.

Ruder, Sebastian (2016). "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747*.

Selvaperumal, P. and A. Suruliandi (2014). "A short message classification algorithm for tweet classification". In: *2014 International Conference on Recent Trends in Information Technology*, pp. 1–3.

Shanker, Murali, Michael Y Hu, and Ming S Hung (1996). "Effect of data standardization on neural network training". In: *Omega* 24.4, pp. 385–397.

Torralba, Antonio and Alexei A Efros (2011). "Unbiased look at dataset bias". In: *CVPR 2011*. IEEE, pp. 1521–1528.

U.S.FDA (n.d.). *FDA cautions against use of hydroxychloroquine or chloroquine for COVID-19 outside of the hospital setting or a clinical trial due to risk of heart rhythm problems*. URL: https://www.fda.gov/drugs/drug-safety-and-availability/fda-cautions-against-use-hydroxychloroquine-or-chloroquine-covid-19-outside-hospital-setting-or.

Van Rossum, Guido et al. (2007). “Python programming language.” In: *USENIX annual technical conference*. Vol. 41, p. 36.

Vidal, Rene et al. (2017). “Mathematics of deep learning”. In: *arXiv preprint arXiv:1712.04741*.

Zhang, Yin, Rong Jin, and Zhi-Hua Zhou (2010). “Understanding bag-of-words model: a statistical framework”. In: *International Journal of Machine Learning and Cybernetics* 1.1-4, pp. 43–52.