Detecting the Political Leaning of News Articles using Neural Networks

Author: Avery Vine (100999500)

Professor: Tony White

April 16th, 2020

1 Abstract

In recent decades, the media that we consume as a population has become more and more politically polarized. This increasing divide may be due to a number of factors, but one significant cause is the way that news organizations, social media, and other online platforms tailor the content they serve to an individual based on their previous interests. These "filter bubbles" can make it very difficult to acquire one's news from a balanced and diverse set of news publications.

Neural networks are a fascinating and increasingly important field of study in computer science, and there have been significant advances in the field in the past couple of decades. This paper delves into the process of creating a neural network trained on a collection of 4,000 news articles from eight publications, for the purpose of analyzing and classifying future news articles according to their political biases. This neural network would allow for a web-interface or mobile application that users could use to recognize the biases in the news they are reading, strive to diversify their news sources, and stay well-informed in a politically balanced way. Of the neural network variations explored (a standard LSTM, a model consisting of both a CNN and an LSTM, and a bidirectional LSTM), the bidirectional LSTM was found to achieve the greatest accuracy with the least overfitting.

Contents

2 Introduction

2.1 Problem Statement

In recent decades, the world has experienced extreme political polarization [1]. There exists now a political divide unlike anything we have seen before, and people with differing political views are finding it ever more challenging to work together to achieve common goals. This divide seems to have been amplified by the algorithms that we interact with every day. News organizations, social media, and other online platforms tailor the content they serve to each individual based on what has interested them in the past, causing "filter bubbles" that make it very difficult for people to reconcile competing viewpoints with their own political beliefs [2].

It is not within the scope of this paper to explore why such a political divide exists. Instead, the goal is to alleviate some of that ideological strain by providing a tool that people can use to understand the biases in the news they are reading and break free of that algorithmic bubble, so that they can diversify their news sources and strive to be more well-informed. The method by which I aim to achieve that goal is the development of a classifier that can determine the political leaning of news articles.

In early 2018, a classmate and I attempted to solve this same problem using a Naïve Bayes classifier. As such, achieving a greater accuracy than what was achieved by that classifier (approximately 80%) will be the measure of success for this approach. This paper will explore some of the background behind the original approach, followed by an in-depth explanation of some of the neural network models that could be used in the context of text classification. Finally, it will discuss the model I chose to use, how it evolved over the course of the project, and ultimately the results the model was able to achieve.

2.2 Motivation

There are various ways to classify text, and many such methods can be combined in some way or another to achieve an even greater accuracy. While this paper is not exploring any revolutionary new methods for text classification, it does explore how we can apply the work of machine learning researchers to political journalism, something that is incredibly important in our society. We can leverage the power of machine learning and neural networks to extract meaning at a phrase or even sentence level from news articles, which can lead to greater consistency and accuracy. Having such a model would allow for the creation of a tool (e.g. a mobile application, website, or browser extension) that could be invaluable for any person who strives to get their news from unbiased, well-informed, and diverse sources.

3 Background

Before we examine the various models that were explored for this paper, we should take the time to explore various methods that are able to achieve the task, along with some of the related work that has been done in the field of machine learning text classification.

3.1 Naïve Bayes Classifiers

While there are various approaches one can take to classify text that involve neural networks, the first that should be discussed is the original approach to this project. In 2018, a classmate and I applied a Naïve Bayes classifier in the context of classifying the political leaning of news articles.

"Naïve Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable."[3] Bayes' theorem (Figure 1) can be broken down into four components:

- $P(H|E)$, the probability of the hypothesis H given the sample has evidence E
- $P(E|H)$, the probability that the sample has evidence E given hypothesis H
- $P(H)$, the probability of hypothesis H for any sample
- $P(E)$, the probability of evidence E for any sample

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

Figure 1: Formula for Naïve Bayes

In the context of text classification, the hypothesis H is the class to which the sample belongs (e.g. left-wing or right-wing), and the evidence E is the words in each document (i.e. the actual content of the article). In essence, the Naïve Bayes algorithm works by reading in the training data, splitting it into individual words, and then counting the number of occurrences of each word within each class. It is a technique often used for the purpose of sentiment analysis.

This algorithm, effective as it is (despite its simplicity), has a clear weakness that should be explored in order to understand why it is insufficient for complex and nuanced works such as news articles. The Naïve Bayes assumption is that the presence of a feature in any of the data has no bearing on the presence of other features. While this may be acceptable for simpler classification tasks, this is patently untrue for more complex written works.

When we write, any given word has its meaning affected by the words around it. (Note: I do not mean to impart my own biases on this paper, but for the sake of clarity in my explanations I am adding examples). For example, when we write "climate change", this may be perceived as a slightly left-leaning phrase. "So-called", while certainly negative in connotation, is a politically neutral phrase. However, as soon as we put them together, the phrase "so-called climate change" becomes significantly more of a right-leaning concept. This is something that is completely ignored by Naïve Bayes, yet is so clearly something that should be taken into account.

This significant weakness in Naïve Bayes classification was a significant motivation for revisiting the task of news article classification. Using neural networks can help us avoid some of these assumptions. The next few pages of this report will explore some of the well-known neural network architectures that I considered for this project.

3.2 Neural Networks

3.2.1 Multi-Layer Perceptrons

In his 1949 book, "The Organization of Behavior", Donald Hebb presented a hypothesis that has since become known as "Hebbian Theory". This theory suggests that synapses in our brains that fire simultaneously (or, more accurately, one immediately after the other) strengthen the connection between them ever so slightly, thus reinforcing the association between them [4]. This theory is often considered to be the basis for unsupervised learning in the brain. Hebb's work became the foundation for the concept of the perceptron, which aimed to use a variety of numerical weights to computationally link a series of inputs to a series of outputs. Once it was recognized that stacking several perceptrons together could drastically improve the quality and breadth of the results, the field of machine learning began to receive significantly more research attention, resulting in what we now know of today as the multi-layer perceptron.

A multi-layer perceptron consists of an input layer, a series of hidden layers, and an output layer. Each layer consists of a number of "neurons", which essentially serve the goal of storing a single number. We can think of the input layer as a series of numbers that represent something for which we want to calculate an outcome, the hidden layers as containing those numbers after performing a number of intermediary calculations, and the output layer as some final numerical values representing a prediction. Each layer is linked together with a series of connections (Figure 2. Incidentally, deep learning is a name given to any machine learning that involves multiple hidden layers [5].

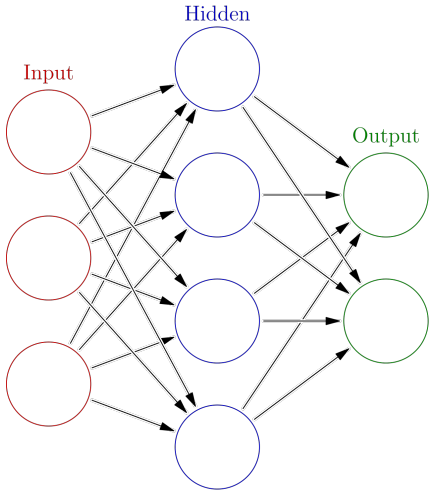

Figure 2: A simple multi-layer perceptron, with one hidden layer [6].

After each calculation, an "activation function" is applied in order to ensure that the transformations we are applying are not linear. There are a variety of activation functions, but the ones that are relevant to this paper are sigmoid, which is bounded and known for its signature S-shaped curve (Figure 3); ReLU, which is unbounded in the positive direction (Figure 3); and tanh, which is a more extreme variation of sigmoid (Figure 4).

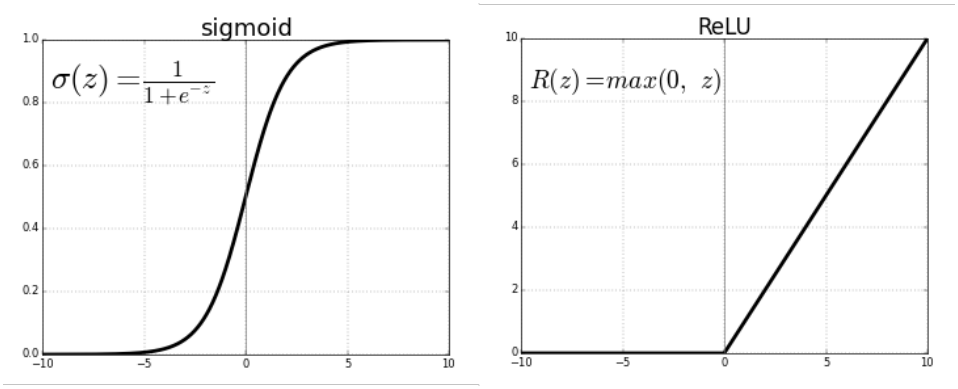

Figure 3: Sigmoid vs. ReLU [7].

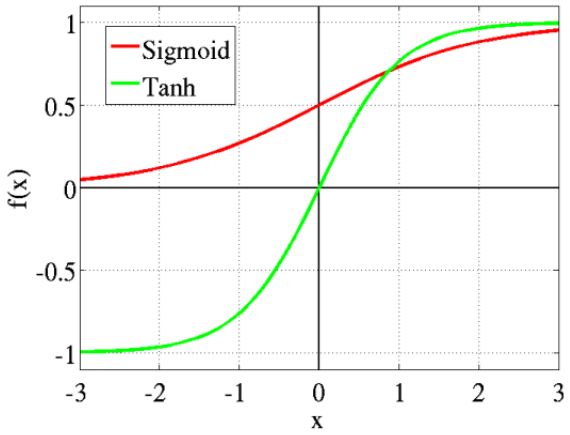

Figure 4: Sigmoid vs. tanh [7].

It is important to note that the input layer does not normally consist of just the data for which we wish to calculate a prediction. A bias term, or an extra neuron, is added to the input layer. As neural networks are essentially complex mathematical functions, a bias term allows for slight adjustments to the positioning of the function, after its shape has been calculated. A very simple example of this is the y-intercept of a linear function ($y = mx + b$).

The concept of dropout should also be introduced here. In order to avoid overfitting the model (i.e. tuning it to fit the training so well that it fails to generalize when presented with data it has not seen before), dropout is a technique that can be applied, where neurons in a neural network are randomly ignored. This means that over the course of several training epochs, neurons are prevented from depending too heavily on other specific neurons in the previous layer, ensuring that the model is not too attuned to the training data [8]. Unfortunately, this also means that the accuracy of a model using dropout can vary between executions, as each execution will ignore a different selection of neurons.

While the multi-layer perceptron has proven to be very successful as a machine learning model for a variety of tasks, its shortcomings for the purpose of this project are similar to that of Naïve Bayes, in that it makes calculations for each piece of data independently of the other data. It does not take into account that the words surrounding a particular word can influence its meaning. There are several approaches that address this problem. This paper will outline the two that are relevant to the decisions made over the course of the project (see **Methodology** below). The first strategy for dealing with the problem of context is the use of convolutional neural networks.

3.2.2 Convolutional Neural Networks

Convolutional neural networks, or CNNs, were developed primarily as a tool for applying deep learning and the concept of neural networks in general to images in a more efficient way. Consider an RGB image with a resolution of 256×256. It is a fairly low resolution image, but 196607 numbers are used to represent it (256 width $\times$ 256 height $\times$ 3 colour layers, plus 1 bias term). If this were to be fed into a traditional fully-connected multi-layer perceptron as a vector, the number of parameters quickly exceeds several billion after just one hidden layer of the same dimension, which makes the model extraordinarily slow to train. In addition, if the image has a very high resolution, pixels that are spatially relevant to each other become spread out in the vector representation that is fed into the input layer, which can lead to lower accuracy.

CNNs were designed to solve these problems. A convolutional layer of a CNN consists of a filter, often 2-dimensional, that sweeps across the image, applies mathematical operations to the pixels beneath it, and provides a final value for each set of pixels it sweeps over (Figure 5). Once the filter has done this for one set of pixels, it shifts x number of pixels (where x is called the stride) and begins again, meaning that pixels are accounted by the filter multiple times. This technique allows us to amplify features that exist within the image (e.g. edges of objects), and even simplify the image (assuming that there is no zero-padding added to the image before the filter is applied). In the case of an RGB image, one filter is applied for each layer, resulting in three numbers per calculation instead of one.

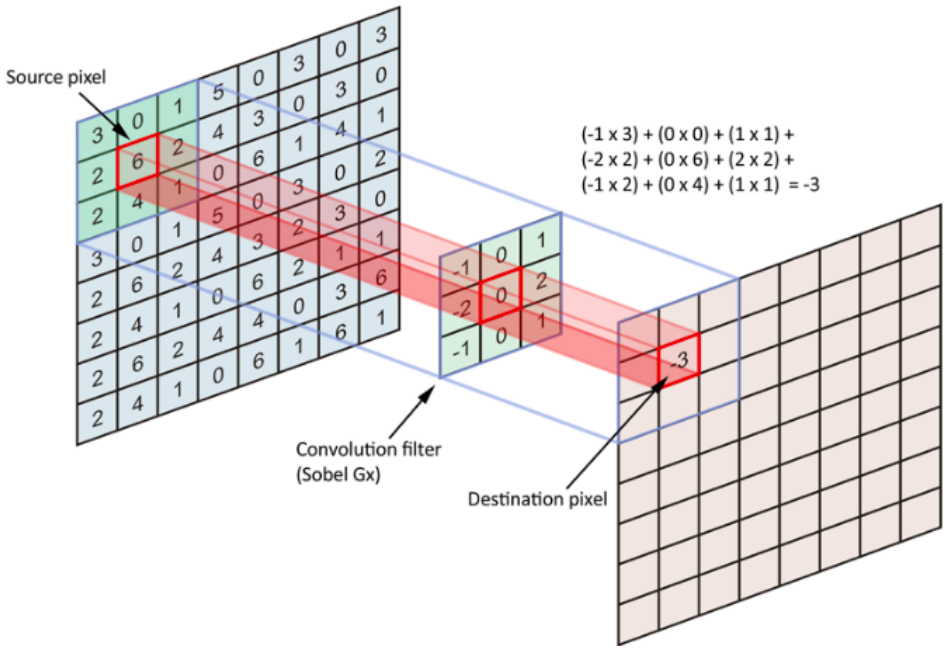

Figure 5: The first step of a 3×3 filter sweeping over a 9×9 image [9].

Another technique used in CNNs is pooling. Most commonly implemented as max pooling (as opposed to average), another 2-dimensional filter sweeps across the image, this time going over each pixel exactly once instead of multiple times. The filter takes in pixels and returns the maximum (or average) value amongst those pixels, providing us once again with the two-fold benefit of shrinking the image and amplifying features. Complex CNNs will often alternate between convolutional layers and max pooling, with different parameters at each step, before finally culminating in a much smaller fully-connected multi-layer perceptron for classification purposes (Figure 6).

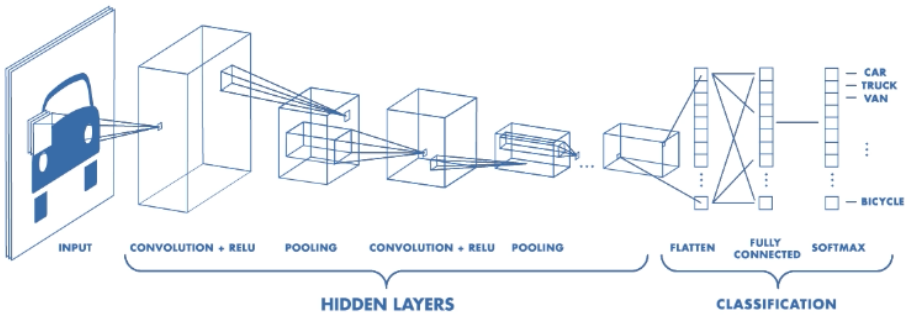

Figure 6: An example of a CNN for classifying various vehicle types [9].

Using a CNN has massive performance benefits over a traditional multi-layer perceptron, thanks to the difference in the number of parameters the two models have. The number of parameters in a CNN is based on the number of filters and their sizes, and entirely independent of the input size. So, considering our 256×256 RGB image example from before: if we apply a single 3×3 filter to each of the three colour layers (plus 1 bias term), the resulting number of parameters is 28, which is far simpler than our original image. Even with with a larger example of 32 5×5 filters, the number of parameters remains significantly smaller at 2401.

Interestingly, despite being developed with images in mind, CNNs can be applied to other contexts. In fact, they have been used as a form of text preprocessing for other neural network models, with the goal of extracting phrases and abstract concepts from groups of words (see **Related Works**). A secondary benefit of using CNNs for this purpose is a reduction in the number of features, which can allow for faster execution in other layers of the model. However, despite their success in extracting phrases, CNNs are rarely chosen to conduct the main task of text classification. This is often left to something like a recurrent neural network.

3.2.3 Recurrent Neural Networks

Recurrent neural networks, or RNNs, share a lot of similarities with multi-layer perceptrons, but they have a "memory" that they can carry forward to each new set of calculations. This makes them extraordinarily useful for any time-series data, or data that takes place over some period of time. Data from text classification problems can be thought of as being a form of time-series data, as the words earlier in a sentence can influence the meaning of later ones.

An RNN begins as a simple neural network (represented as A in Figure 7). A single piece of data x_t is fed into the network, and a result h_t is predicted, just like in a multi-layer perceptron. However, alongside that result, some portion of the calculations that were made during the processing of that data are carried forward, to influence the calculations for the next piece of data. Those numbers that are carried forward are the "memory". The next piece of data x_{t+1} is fed into the exact same neural network as the previous data, with the exact same weights, but the numbers carried forward from the previous data influence the prediction h_{t+1}. This process continues to loop until there is a prediction for every single piece of data [10].

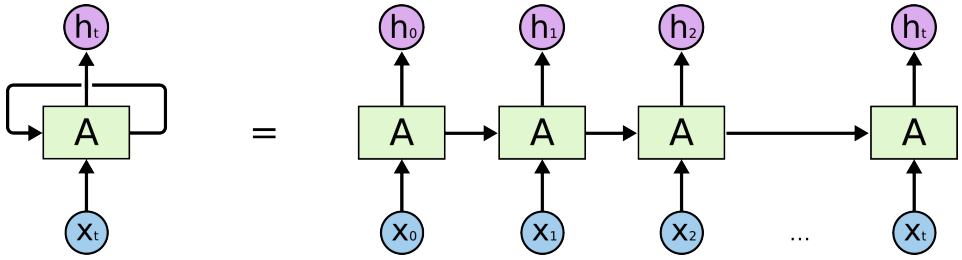

Figure 7: An RNN can be represented with a single neural network layer A, or "unrolled" as repetitions of that same layer A. Portions of each calculation are carried forward [10].

Unfortunately, one of the problems to this approach is that the values carried forward from any given calculation can only have an impact for a few layers. Consider some set of weights W for our RNN, where $\forall\ w \in W, w < 1$. Since the same neural network is used in every iteration of the loop, this means that W is also used at every iteration. Thus, our "memory" from any given layer is multiplied by weights of less than 1 at every step, and that memory's contribution to the calculations quickly becomes negligible. This means that RNNs have a very limited memory, potentially of only a few steps, and can only remember a few words in the context of text classification. The idea that the gradient calculated during backpropagation disappears due to this is known as the vanishing gradient problem [11].

3.2.4 Long-Short Term Memory Neural Networks

Long-short term memory neural networks, or LSTMs, were devised to counteract this problem. In Figure 9, one can see that the upper path between each layer is straightforward, only interacting with other pieces in a few places. This is a path that is set aside exclusively for calculations to be transferred between steps. Any interaction with this data is gated by a few key functions: a "forget gate" to determine how much information from previous calculations should be removed, an "update gate" to determine how much information from the current calculation should be transferred to memory, and an "output gate" to determine how much of the memory should be used for the current prediction (Figure 8) [10].

$$\Gamma_u = \sigma(W_u[h^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u$$

$$\Gamma_f = \sigma(W_f[h^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f$$

$$\Gamma_o = \sigma(W_o[h^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o$$

Figure 8: The formulas for forget gates, update gates, and output gates, respectively.

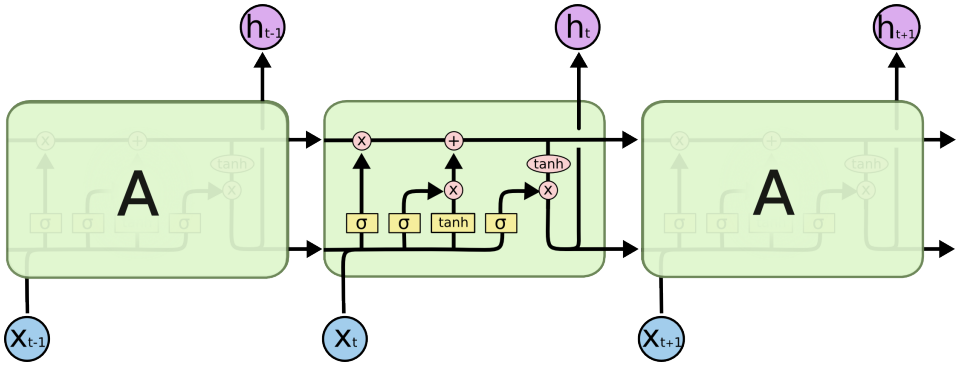

Figure 9: The arrows between the steps in the upper half indicate the "memory highway" that allows LSTMs to remember information for much longer than normal RNNs. It is modified twice (forget gate & update gate) and drawn from once (output gate) at each layer repetition [10].

In the context of text classification, LSTMs allow words to have an impact on the calculations for the words that follow them for a much greater number of steps. This is why they are very commonly used for tasks of this kind, as I will discuss in the next section.

3.3 Related Works

Many works have been published that employ RNNs, and especially LSTMs, for text classification. However, these LSTMs are often accompanied by some other machine learning component in an attempt to improve accuracy.

Zhou et al. (2015) decided to combine the power of both kinds of neural networks mentioned above, in an attempt gain a higher level of accuracy for text classification tasks [12]. They designed a model that they called a C-LSTM. CNNs are able to extract meaning and context from localized data, but do not associate those meanings to each other long-term, as discussed above. Zhou et al. realized that they could feed the results of such extractions directly into an RNN - or, more specifically, an LSTM - in order to combine the power of spatial correlation extraction with long-term correlation ability. It is important to note that they did not employ any form of max-over-time pooling or dynamic k-max pooling, as those can break the organization of sequences requires for the input to an LSTM.

Zhou et al. tested their model in the context of both sentiment classification (which is essentially analogous to political leaning classification) and question type classification (e.g. classifying "what is the highest waterfall in the United States?" as a "location" question). In the sentiment classification case, they found that their results were on part with other state-of-the-art methods at the time, with an accuracy of 87.7%. When it came to question type classification, the model came second only to an SVM model with highly-engineered features, which was impressive for a much more generalized model like C-LSTM. These encouraging results directly influenced my later decision to experiment with the use of CNNs as a layer in the model (see **Results**).

Zhou et al. (2016) (confusingly, not the same Zhou) took a different approach, employing a bidirectional LSTM to begin the text classification process before feeding the results into a two-dimensional convolutional layer and a two-dimensional max pooling layer [13]. If an LSTM is bidirectional (abbreviated as BLSTM), it means that both past and future pieces of data influence calculations and predictions, instead of exclusively the former. This can be helpful in the context of text classification, because the meaning of a phrase is sometimes clarified or altered by what comes after it. The theory proposed in this paper was that the results of an LSTM can be thought of as an $n \times m$ matrix, where each of the n predictions produced by the LSTM has m features. As this is a two-dimensional matrix, it can be fed into a convolutional layer and then pooled, just like in a traditional CNN.

Zhou et al. tested four variations of their model (BLSTM-CNN, BLSTM-2DCNN, etc.) on six datasets, two of which were the same as those tested in the C-LSTM paper. In the sentiment classification case, their BLSTM-2DCNN model performed even better than the C-LSTM model discussed above, with an accuracy of 89.5%. When it came to the question type classification, their BLSTM-CNN model also beat all baselines with an accuracy of 96.1%. The idea of a bidirectional relationship between pieces of data, combined with Zhou et al.'s exceptional results, guided some of my experimentation by influencing me to explore the idea of a BLSTM (see **Results**). Unfortunately, I did not get the chance to experiment with convolution after the LSTM layer.

Finally, during my research I came across a paper where Rao and Spasojevic (2016) used an LSTM to conduct text classification on tweets in two different contexts: actionable/non-actionable tweets, or tweets that the support teams for service providers may or may not be able to act on; and political leaning with respect to American politics (i.e. Democratic/Republican instead of generically left-leaning/right-leaning) [14]. While the former is certainly interesting, the latter is a direct parallel to the project for this paper, and as such will be the focus on this section. Their report also explained that the latter task was more nuanced. This is because the actionability of tweets is largely based on the general sentiment of the tweet (i.e. action should be taken on tweets that can hurt the company's brand), whereas political tweets can have a wide variety of sentiments associated with them.

Rao and Spasojevic retrieved their political leaning data by using manually curated Twitter Lists to pick out users with well-known political leanings, then aggregating all of their tweets over a three-month period. This yielded 336,000 tweets for the training data and 84,000 tweets for the test data, labelled 0 and 1 for Democratic and Republican respectively. This data was then fed into an embedding layer with 128 units, before reaching the most important layer of their model, an LSTM with 64 output units. They then used a dropout layer where half of the connections were dropped in order to prevent overfitting, before feeding their results into a fully-connected layer for a final prediction.

During their experimentation, Rao and Spasojevic kept many of the values of their model static, in order to test specifically the number of embedding units (features), the number of LSTM output units, and the optimizer at the end. Interestingly, they found that in general, the model's accuracy slowly trended upwards as both the number of embedding units and number of LSTM units was increased, as long as these two values were similar in size. They believe that this suggests optimal values could be chosen for these two hyperparameters, given a fixed input size. Despite this, they also found that the model achieved peak accuracy when the number of LSTM output units was 32 and the number of embedding units was 128, and as such those are the choices they made when constructing the finalized model.

One significant difference between Rao and Spasojevic's project and this one is the data itself. Tweets are purposefully designed to be very short, fulfilling their purpose in 140 characters or less (at the time of their project - this limit has since been raised to 280). Conversely, news articles are often hundreds or even thousands of words long, and therefore have the capability of conveying much more complex ideas. This significant difference in the representation of data means that while their model was able to achieve an impressive accuracy of 87.57% on their test set, their relatively simple model may not apply itself perfectly to this project's use case, and it is worth exploring more complex models. Rao and Spasojevic's paper still held a significant influence on some of the decisions I made concerning the creation of the initial model, but I moved towards integrating additional layers soon after achieving some level of success (see **Building the Model**).

4 Data and Aggregation

Political leaning is a concept that we as a species have invented, and as such the model must be trained using a supervised learning method (learning in which the model is provided the correct classification for all of the training data). Ideally, the political labels applied to the training data would accurately and objectively represent each article. However, individual biases may end up being present in the training data, which leads to those same biases being reflected in the model after training. One of the ways in which I aimed to improve this project in comparison to its previous iteration from 2018 was to diminish the effect of my own personal biases on the model by diminishing it in the training data. To do this on a large scale, however, proved to be very difficult. When my initial data aggregation method failed to yield the quality and quantity of data I was hoping to achieve, I switched to an alternative data aggregation method. I will go into detail about the two methods below.

4.1 Aggregation Methods

I used two data aggregation methods over the course of this project, as outlined in the sections below.

4.1.1 Method One - Reddit Scraping

Reddit is a social media site in which users gather in communities called "subreddits". Each subreddit has a name prefixed with "/r/" and contains posts pertaining to a particular topic. For example, /r/science is a subreddit dedicated to the discussion of various studies that are being published in scientific journals, while /r/ottawa is a place where users can post about things related to the nation's capital. When people post content to Reddit that other people enjoy or agree with, they give it an "upvote", or a virtual point. It follows that posts with many upvotes should represent the interests of a large number of people. As such, my initial approach was to scrape popular articles from several political subreddits, starting with /r/liberal and /r/conservative, and apply labels to the data based on which subreddit it came from. This method does not guarantee that the general biases of the global population of Reddit users would be accounted for, as the data comes entirely from that population; however, in theory it should be relatively effective. In order to collect this data, I used the PRAW python module to scrape the top n posts from each subreddit [15], then another module called newspaper3k to extract the article contents [16].

Unfortunately, there are various complications with this method of data aggregation. Primarily, there are simply not enough highly upvoted news articles on either subreddit to achieve the quantity of data a neural network needs. As soon as one scrapes more than a couple hundred articles, the number of upvotes is not high enough for anyone to confidently say it represents the general views of an entire political leaning (Figure 10). It should be noted that at the time of writing, /r/conservative has significantly more members than /r/liberal, yet the problem remains. Secondly, not all of the highly upvoted posts on each subreddit are actually news publications. This problem was especially prevalent with the

/r/conservative subreddit, where every fifth scraped post was from a website that was not a news publication (Figure 11).

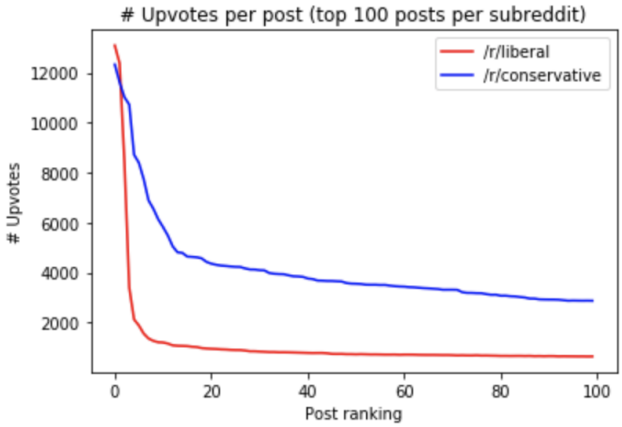

Figure 10: The number of upvotes a post has dropped off significantly after the first few posts on both subreddits.

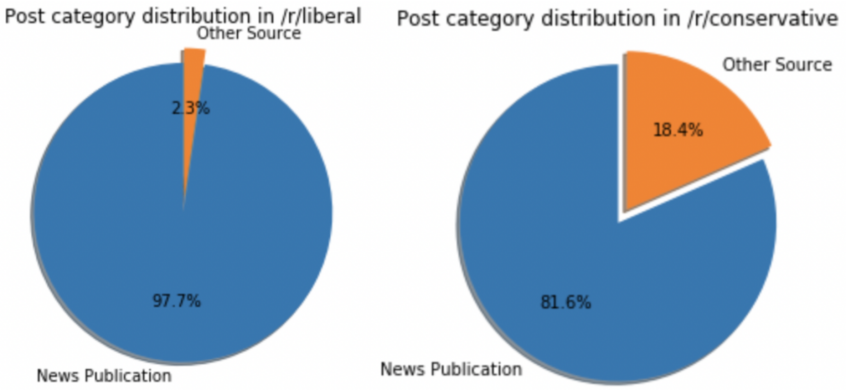

Figure 11: A significant number of posts from /r/conservative were not from news publications.

Despite the low quality and quantity of data that this aggregation method produced, I opted to temporarily use the data while I worked on producing a model that functioned (see **Results**). However, I knew that if I wanted to improve the scale and accuracy of the model, I would need significantly more high-quality data.

4.1.2 Method Two - News Publication Scraping

An alternative method of data collection that is capable of yielding far more data is to use a web crawler to collect news articles directly from the publications themselves, and apply my own labels. The command line tool is "news-please" is an open-source web crawler that designed specifically for use on news publications [17]. It leverages several well-known Python modules, including scrapy and newspaper3k (which was instrumental to the previous aggregation method), to recursively follow internal hyperlinks and aggregate news articles, before parsing out their titles, contents, authors, and more.

In order to apply labels to the news articles collected from news-please, I opted to use a general guide published on a website called AllSides, which is dedicated to providing articles from multiple political leanings that cover the same news story. They use a combination of blind bias surveys, third party analysis, editorial review, community feedback, and independent research to calculate their Media Bias Chart (Figure 12) [18]. While it may not be perfect, their methodology does mean that one particular bias calculation method will not have an overly dominant effect on the chart, and it allowed me to classify the articles I was crawling without imparting my personal biases on the model.

Figure 12: Political biases of various news publications, according to AllSides [19].

One potential issue with this method is that the Media Bias Chart only provides averages for each publication. It is quite possible that some articles from each news source do not align themselves politically with the rest of the publication, which would have a negative influence on the accuracy of the model during training. It should also be noted that while the Media Bias Chart has 5 categories of political bias, this project was conceived as a binary classifier. As such, I did not use any sources in the "Center" category of the chart, and made sure to use the same number of publications from each of the remaining four categories ("Left", "Lean Left", "Lean Right", and "Right").

4.2 Analysis of Data

I used news-please to scrape articles from eight different publications, labelling article with a political bias as they were scraped, in accordance with the Media Bias Chart. Unfortunately, while the tool does let the user specify a minimum publication date (e.g. January 1st, 2015), it actually scrapes articles older than that before discarding them. As such, I was forced to scrape for many hours before I was able to collect enough data from every publication. I ended up with many thousands of articles from certain publications, and yet only hundreds from others. Once every publication had reached a minimum count of 500 articles scraped, I randomly selected 500 articles from each one. This formed a collection of 2,0000 high-quality left-leaning articles and 2,000 high-quality right-leaning news articles, all from 25/01/2015 or later, for use in my model (Table 1).

News Publication	**Bias (AllSides Chart)**	**Bias (Dataset)**	**# Articles**
New York Times	Lean Left	Left	500
Washington Post	Lean Left	Left	500
Huffington Post	Left	Left	500
Vox	Left	Left	500
Wall Street Journal	Lean Right	Right	500
Fox News	Lean Right	Right	500
Breitbart	Right	Right	500
New York Post	Right	Right	500
Total Number of Articles: 4,000			

Table 1: Summary of the data collected for this project.

Something that only became apparent after I had progressed much further into the project was that the articles retrieved from the New York Times were on average extremely short compared to other publications. However, other publications (especially Vox) had much higher average word counts, making the distribution between the two political leanings more even than may have been originally apparent (Figure 13). One way of dealing with such issues is specifying a maximum length for each input. This can be achieved by setting a maximum sequence length when using Keras' tokenizer (see **Methodology**). Nonetheless, this aspect of the project would need more attention to if it were to be repeated it in the future.

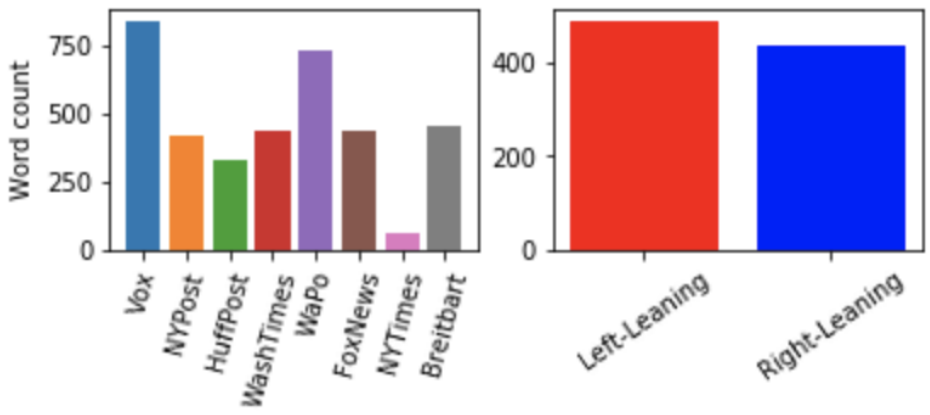

Figure 13: Average word count per publication vs per political leaning.

5 Methodology

The following sections will describe the decisions and steps that went into the construction of the initial model.

5.1 Data Preprocessing

One of the most important things one must do when creating a machine learning model is prepare the data that is going to be fed to that model. Since the data for this project consists purely of raw text from news articles, there is a lot to be done. The following steps are executed each time the model is run, in part because sequence length is a parameter I modified.

First of all, not all of the data is to be used for training the model. Each time the model is executed, the data is randomly split into three components using Scikit-Learn's train_test_split method [20]. 20% of the data became the testing data, and then I split a further 20% of the training data off for validation. Having this separation of the data provides the ability to measure the performance of the model both during and after training.

Besides regular words, the raw article data contains a mix of punctuation, varying capitalization, numbers, links, and more. As such, the data must be preprocessed before use, albeit the level of processing isn't very complex. Every article was converted to contain only lowercase letters, and then special characters and numbers were removed. One particular operation that is often used when cleaning data for use in machine learning or other algorithms is some form of stemming or lemmatization. Stemming is the act of removing portions of words in order to attempt to reduce them to their base form (e.g. caresses →

caress). However, stemming can sometimes produce crude results that do not exist in the English language (e.g. ponies → poni). Lemmatization is a similar technique that aims to use the morphology of a word as a guide for reducing it to its base form (e.g. ponies → pony) [21]. It is much more time-consuming, but can yield higher quality results. One of these two techniques is often applied in order to group together words that are related, such that there can be more instances of a smaller number of words, rather than a broad number of variations of the base word with few examples. Despite the prevalence of these operations, neither one was used in the data cleaning process, because I was using a pre-calculated set of word embeddings from Gensim (explained in further detail below).

Of course, a computer cannot conduct mathematical operations on a word, and our data still only consists of raw words. As mathematical operations are very much required by neural networks, words must somehow be represented as numbers. A very simple place to start is to build up a "vocabulary" of all of the potential words the model might encounter during training. For example, if we have three pieces of data ["my name is", "name is avery", "my name avery"], our vocabulary consists of four words. These words can be numbered [my: 1, name: 2, is: 3, Avery: 4], which allows us to represent our original data as a sequence of numbers, like so: [[1, 2, 3], [2, 3, 4], [1, 2, 4]]. Keras offers a "tokenizer" that is able to do this efficiently for large amounts of data (more about Keras in **Methodology**). The tokenizer constructs a vocabulary from the raw training data, and transforms all of that data into a series of sequences. However, there is still more that must be done before we can feed the data to a machine learning model.

The relationship between two words can seem obvious to humans (e.g. "king" and "queen"), yet that relationship is very abstract. Thus, in order for a computer to derive meaning from a word, it must be numerically represented somehow in relation to other words. One common way to do this is to use a d-dimensional vector representation for each word. The words "king" and "queen" would have similar vector representations to "man" and "woman", allowing the computer to interpret the relationships between those words. Thus, the measure of similarity between two vectors can be computed using cosine similarity (Figure 14). The closer the cosine similarity is to 1, the most similar in context two words are.

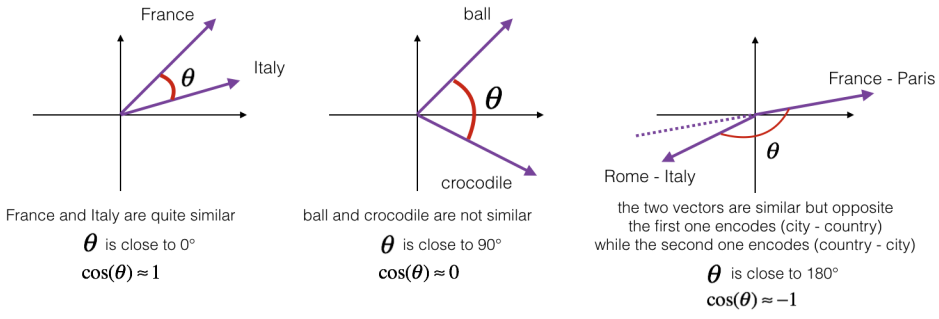

Figure 14: Various examples of cosine similarity in a simplistic 2-dimensional space [22].

Keras allows us to train our own word embeddings as a part of the model training process, but doing so proved to cause the model to overfit massively (see discussion in **Building the Model**). An alternative to this is to use word embeddings that have been pre-trained on another corpus. Gensim provides a number of different options in terms of word embeddings, and I was able to experiment with various embedding sets of different dimensions that had been trained on a corpus of several billion Twitter posts [23]. I took the vocabulary that was constructed using Keras' tokenizer, found the Gensim word embedding for each word, and fed that into the model instead. This is also the reason why applying stemming or lemmatization does not work: if the words are modified in any way, they cannot be matched to their Gensim word embeddings.

5.2 Building the Model

A simple and intuitive way to get started with neural networks is using Keras, which is a Python module geared towards creating machine learning models. Keras is flexible in that it is just as easy to make a model that consists of two or three components as it is to create one with 10 or more, and can be used with a variety of backends (Tensorflow, Theano, etc). My initial research into machine learning and creating neural networks came from reading sections of Aurélien Géron's book, "Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow" [24]. Thus, using Keras with a Tensorflow backend seemed like a logical choice.

As the goal of this project was to utilize a neural network as an alternative to Naïve Bayes for text classification, the early stages of the project consisted of some research into related works (see **Related Works**). There were two approaches that stood out: using a Convolutional Neural Network (CNN) and using a Recurrent Neural Network (RNN). While CNNs, or specifically 1D convolutional layers, can be useful in the context of text classification, they are primarily used for image classification, whereas RNNs seem to have found a larger degree of success in this problem space. As such, I opted to use an RNN - specifically, the LSTM variation described earlier in the paper.

5.2.1 Initial Architecture

The initial architecture was relatively simple. The first layer was an embedding layer that fed the model the vector representations for the training data vocabulary. I initially made use of Keras' embedding layer to calculate custom word embeddings, but that proved to cause a massive amount of overfitting (Table 2, Figure 15).

Using Gensim's 25-dimensional Twitter embeddings meant that the vector representations for the data were well-trained in relation to each other, but not specifically for this context, meaning that the bulk of the calculations in the model could be left to the neural network itself. I chose to begin with 25 feature dimensions, as it allowed me to iterate more quickly due to the faster execution of the model. The maximum sequence length was specified as 1000, which limited the impact that extremely long articles had over shorter ones.

The LSTM layer I initially added had an output size of 128, a dropout value of 0.3 (chosen to counter an anticipated moderate amount overfitting), and a ReLU activation function. Those 128 outputs were fed into a dense layer that amalgamated everything into a single result, with sigmoid applied in order to regularize the result and achieve a final prediction. Surprisingly, the initial run was already significantly more successful than the Naïve Bayes approach had been. With 30 epochs of training, it produced a result that did not overfit and exceeded 90% accuracy (Table 3, Figure 16).

	Parameter	Value
Embedding	Source	Training Data
	Feature Dims.	25
	Sequence Length	1000
LSTM	Output Units	128
	Dropout	0.3
	Activation	ReLU
Dense	Output Units	1
	Activation	ReLU
Compile	Loss Function	Binary Cross-Entropy
	Optimizer	Adam
Fit	Batch Size	64
	Epochs	30

	Train	Valid	Test
Accuracy	99.9%	91.68%	91.55%

Table 2: The initial model, when used with word embeddings created directly from training data.

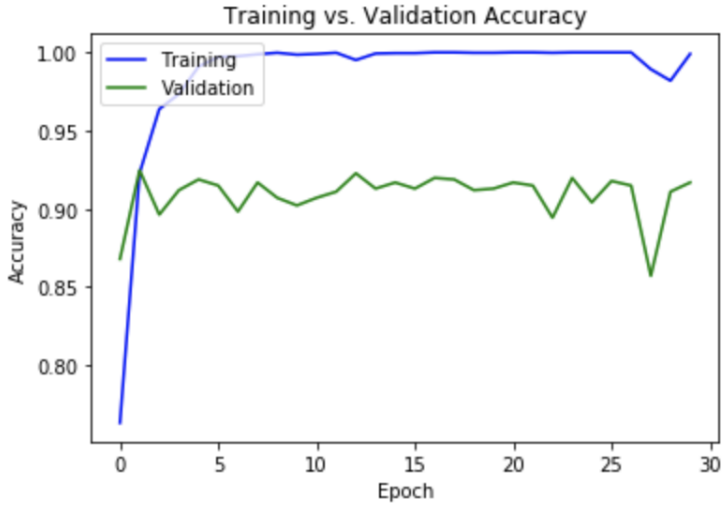

Figure 15: The accuracy graph for the initial model, when used with word embeddings created directly from training data. Lots of overfitting! (See Figure 16)

	Parameter	Value
Embedding	Source	Gensim (Twitter)
	Feature Dims.	25
	Sequence Length	1000
LSTM	Output Units	128
	Dropout	0.3
	Activation	ReLU
Dense	Output Units	1
	Activation	ReLU
Compile	Loss Function	Binary Cross-Entropy
	Optimizer	Adam
Fit	Batch Size	64
	Epochs	30

	Train	Valid	Test
Accuracy	91.68%	91.78%	90.32%

Table 3: The initial model, when used with word embeddings from Gensim.

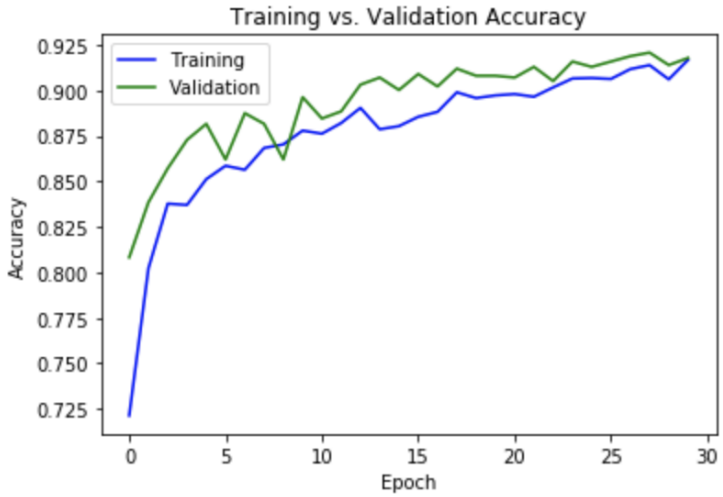

Figure 16: The accuracy graph for the initial model, when used with word embeddings from Gensim.

While the test accuracy was still not quite as high as the training and validation accuracy, this model did achieve the goal of the project (surpassing the results of the initial implementation). However, there are many more avenues to explore with the model, which I will outline below, along with their respective results.

6 Results

6.1 Experimenting with Standard LSTM

Besides adding layers, there are a number of modifications that can be made to even a model as simple as this one, and as such I began my experimentation with some of those modifications. Using a greater number of epochs did not prove to help during training; the validation accuracy appeared to peak consistently at the same value as above over the course of 50 epochs. Incrementing batch size helped speed up training, but decreased the overall accuracy by about 1%. The model details can be seen in Table 4.

	Parameter	**V1**	**V2**	**V3**
Embedding	Source	Gensim	Gensim	Gensim
	Feature Dims.	25	25	25
	Seq. Length	1000	1000	1000
LSTM	Output Units	128	128	128
	Dropout	0.3	0.3	0.3
	Activation	ReLU	ReLU	ReLU
Dense	Output Units	1	1	1
	Activation	1	ReLU	ReLU
Compile	Loss Function	Bin. X-Entropy	Bin. X-Entropy	Bin. X-Entropy
	Optimizer	Adam	Adam	Adam
Fit	Batch Size	64	64	**128**
	Epochs	30	**50**	30
Accuracy	Train	91.68	92.39%	90.21%
	Valid	91.78	91.14%	90.73%

Table 4: Comparing the results of tweaks to the number of epochs and batch size. V1 is the initial model.

Although more time could have been spent tweaking various hyperparameters for just these initial layers, my interest in the works I found during my research pushed me to experiment almost immediately with adding new layers.

6.2 Experimenting with CNN + LSTM

The first major modification made to the model was the addition of some CNN-related layers. The modified model now contained a 1-dimensional convolutional layer meant to extract higher-level meanings from words, followed by a 1-dimensional max-pooling layer to pick out the most impactful features. The resulting data was then fed into the LSTM, as before, but this time with a dropouts of 0.4 and 0.5 (in an attempt to preemptively stop overfitting). For the sake of rapid experimentation, I lowered the number of epochs to 15. The model details can be seen in Table 5 and Figure 17.

	Parameter	V1	V4	V5	V6	V7
Embedding	Source	Gensim	Gensim	Gensim	Gensim	Gensim
	Feature Dims.	25	25	25	25	25
	Seq. Length	1000	1000	1000	**250**	1000
1-D CNN	Output Units	—	128	**64**	128	128
	Filter Size	—	3	3	3	3
	Activation	—	ReLU	ReLU	ReLU	ReLU
Max Pool	Pool Size	—	4	4	**2**	**2**
LSTM	Output Units	128	128	**64**	128	128
	Dropout	0.3	**0.4**	**0.4**	**0.4**	**0.5**
	Activation	ReLU	ReLU	ReLU	ReLU	ReLU
Dense	Output Units	1	1	1	1	1
	Activation	ReLU	ReLU	ReLU	ReLU	ReLU
Compile	Loss Function	X-Entr.	X-Entr.	X-Entr.	X-Entr.	X-Entr.
	Optimizer	Adam	Adam	Adam	Adam	Adam
Fit	Batch Size	64	**32**	64	64	64
	Epochs	30	**15**	**15**	**15**	**20**
Accuracy	Train	91.68%	97.48%	93.76%	95.42%	93.25%
	Valid	91.78%	91.67%	90.49%	91.26%	90.85%

Table 5: Various models that include 1-D convolutional / max pooling layers. Not included: 5+ other experiments yielding similar overfitting.

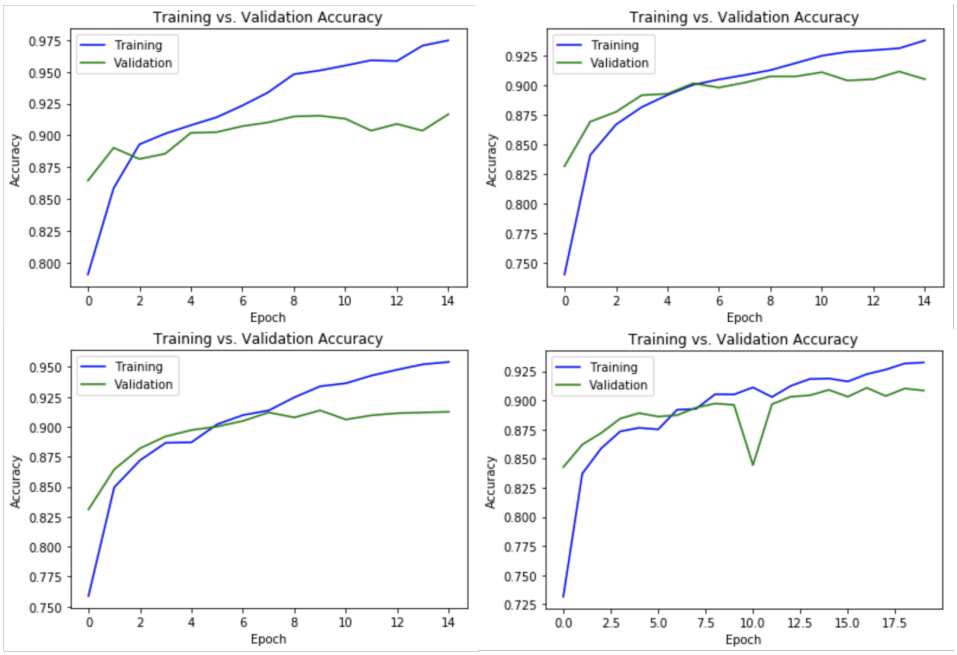

Figure 17: Training vs. validation accuracy for V4 (top left), V5 (top right), V6 (bottom left), and V7 (bottom right). Every CNN experiment resulted in overfitting.

While the accuracy of the model did increase during training, the addition of these layers resulted in overfitting for several hyperparameter variations that I tried, despite the significantly lower number of epochs. Higher dropout rates (0.4, 0.5), lower numbers of output units for the CNN and LSTM layers (64), and smaller pooling layer sizes (2) all seemed to reduce overfitting somewhat. However, even with those modifications, the validation accuracy remained several percentage points behind the training accuracy. It seems reasonable to suggest that the quantity of data collected for this project (4,000 articles) was insufficient, as with more data the model would have to spend longer attempting to fit the model. Alternatively, the use of data augmentation techniques could have provided more data to work with without using the web crawler to scrape more articles (see **Future Work**). After this experimentation, I reverted to a model without a CNN layer that seemed to be capable of achieving a result less prone to overfitting, which also allowed for the use of the 50-dimensional Gensim Twitter dataset.

6.3 Experimenting with BLSTM

Although the use of a CNN layer did not provide did not yield acceptable results, the modification of the LSTM layer to be bidirectional (i.e. a BLSTM, as described above) proved to be more successful. The idea of the context of word being affected by what is on either side of it seems to be validated, as the simple addition of a bidirectional component increased the accuracy of the model without affecting the amount overfitting.

	Parameter	**V1**	**V8**	**V9**	**V10**	**V11**
Embedding	Source	Gensim	Gensim	Gensim	Gensim	Gensim
	Feature Dims.	25	50	50	50	50
	Seq. Length	1000	500	500	500	500
LSTM	Bidirectional	No	No	**Yes**	**Yes**	**Yes**
	Output Units	128	128	128	**64**	**50**
	Dropout	0.3	0.5	0.5	**0.4**	0.5
	Activation	ReLU	ReLU	ReLU	ReLU	ReLU
Dense	Output Units	1	1	1	1	1
	Activation	ReLU	ReLU	ReLU	ReLU	ReLU
Compile	Loss Function	X-Entr.	X-Entr.	X-Entr.	X-Entr.	X-Entr.
	Optimizer	Adam	Adam	Adam	Adam	Adam
Fit	Batch Size	**64**	32	32	32	32
	Epochs	30	**50**	**50**	**30**	**35**
Accuracy	Train	91.68%	94.21%	95.81%	94.50%	92.50%
	Valid	91.78%	92.66%	93.03%	93.44%	92.19%
	Test	90.32%	—	—	92.88%	92.96%

Table 6: Comparing the initial model (standard LSTM) and a later experiment (standard LSTM) with various bidirectional LSTMs. V11 achieved the greatest test accuracy of all the models tested.

Unlike the CNN experiments, the addition of a bidirectional component did not vastly change the model architecture, but still gave it more capability to relate words to each other, which manifested as a general improvement in accuracy.

6.4 Final Model

In the end, the most successful model proved to be V11 (detailed in Table 7 and Figure 18 below). This model was able to achieve a significantly higher accuracy than the Naïve Bayes classifier approach was able to provide, which was the primary goal of this project. It was also able to achieve a greater accuracy than the standard LSTM model that was initially constructed. Moreover, it was able to achieve this in a way that avoided the overfitting problems presented by the use of convolutional layers, leading to a distribution of training, validation, and test accuracy within a single percentage point difference of each other. The fact that the most successful model contains 50 LSTM units in combination with 50-dimensional embeddings from Gensim also seems to corroborate Rao and Spasojevic's suspicion that linking these hyperparameters can lead to better model accuracy (as discussed in **Related Works**).

	Parameter	**V1**	**V11**
Embedding	Source	Gensim	Gensim
	Feature Dims.	25	50
	Seq. Length	1000	500
LSTM	Bidirectional	No	Yes
	Output Units	128	50
	Dropout	0.3	0.5
	Activation	ReLU	ReLU
Dense	Output Units	1	1
	Activation	1	ReLU
Compile	Loss Function	Bin. X-Entropy	Bin. X-Entropy
	Optimizer	Adam	Adam
Fit	Batch Size	64	32
	Epochs	30	35
Accuracy	Train	91.68	**92.50%**
	Valid	91.78	**92.19%**
	Test	90.32	**92.96%**

Table 7: Comparing the initial model (V1) to the final model (V11).

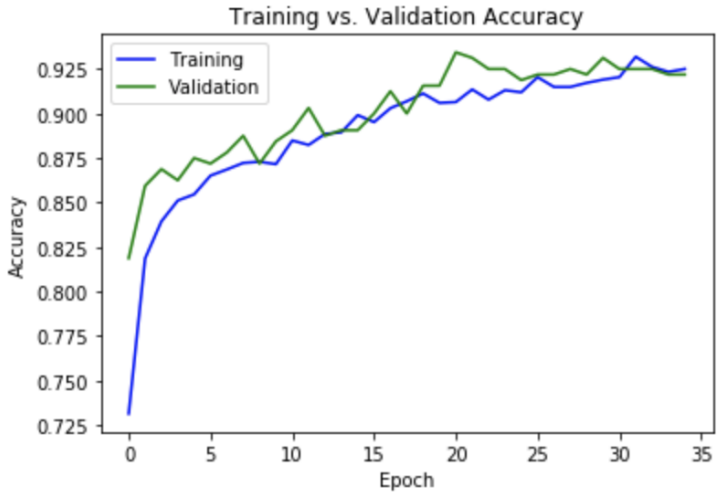

Figure 18: Training vs. validation accuracy for the final model (V11).

7 Challenges

I faced a number of significant challenges over the course of the project. The first one, described in **Data Aggregation**, was the inadequate quality and quantity of the data collected from Reddit. While I believe that this data aggregation method, in principle, achieved the goal of eliminating a significant amount of political bias from my training data by relying on the aggregate opinion of political communities on Reddit, there were simply not enough news articles that had a high enough number of upvotes for me to consider them representative of the community in which they were posted. As such, I was forced to pivot to an entirely different aggregation method, in which the decisions I personally made relating to the bias of one particular news source over another would have ramifications on the model, once it had been trained. If I were to repeat this project in the future, I would want to use another process, similar to the Reddit aggregation method, to collect large quantities of news articles without having to personally make decisions about which political leaning they belong to.

Another significant challenge I faced was the time it took simply to train my model. LSTMs by definition requires some component of each calculation to carry over to the next one (as described in **Long-Short Term Memory Neural Networks** above). This not only meant that training could take a significant amount of time, but also that the processing power of the GPU was being underutilized (averaging around 15% utilization) as the CPU was simply not able to deliver training data to the GPU quickly enough and acted as a bottleneck. GPU utilization can often be increased by using a higher batch size during training, but increasing the batch size comes with its own problems. With higher batch sizes, the model was simply not able to allocate enough memory during training, leading to the model crashing. This is because LSTMs are relatively complex in their own right,

and they require lots of memory to function. This combination of slow speeds and memory constraints forced me to spend far longer than I would have liked trying to get the model to work in the first place. I was finally able to overcome that challenge by adjusting the length of the training data sequences, which reduced amount of data being fed into the model at any one time.

The third, and very significant, challenge that I faced was the overfitting of the model. This was a persistent theme throughout the project, especially when convolutional layers were incorporated into the model. Despite several experiments with various hyperparameter variations, overfitting was present every time a model involving convolutional layers was trained. A potential solution to this challenge could have been the acquisition of more data, which ties this directly to the first challenge. Another could have been data augmentation, which would have increased the amount of data available for the model to train on, without requiring more web scraping (see **Future Work**).

8 Future Work

The data aggregation process has already been thoroughly discussed in this report, but it is worth mentioning one more time in the context of potential next steps for this project. Future work would almost certainly entail dedicating a significant amount of time to reducing the bias in the training data. Removing bias is especially important in the context of this model, because it deals with politics. Many people are somewhat sensitive to things that may affront their political views and, as such, bias is something that should be considered carefully at every stage, right from the data aggregation all the way to the hyperparameter tuning for the final product.

Another fascinating angle to the data aspect of the project is the idea of data augmentation. It is somewhat common in machine learning to apply a variety of transformations to the training data before use, as this can help the model prepare for inputs that are not explicitly outlined in the training data. This can be achieved with a couple of relatively simple techniques, as outlined in a paper by Wei and Zou (2019). They found that the application of synonym replacement, random insertion, random swap, and random deletion could increase model accuracy by an average of 0.8% when applied to full datasets, and by an average of 3.0% on small datasets (e.g. $N_{train} = 500$) [25]. Applying this early in the project could have also helped to reduce the significant amount of overfitting caused by the application of convolutional layers.

This work on data aggregation and augmentation could also be applied to other future work: for example, the inclusion of other sorts of layers in the model. The addition of bidirectionality to the LSTM layer already proved to create a model with superior accuracy to a standard LSTM, while avoiding overfitting. It would be very interesting to see what results a BLSTM could achieve given access to more data than was provided for this project. Future work on data could also be useful when it comes to the addition of a convolutional layer to the model. Despite my experimentation with adding such a layer, I feel that there is so much I left unexplored, thanks to the large amount of overfitting they caused. As such,

it would be an excellent opportunity for future work on this project.

Finally, it should be noted that this model is not very useful if no one can use it! A step that I would love to take, especially as someone with a significant interest in mobile application development, would be the creation of a user interface to interact with the model. A mobile application, website, or browser extension would allow this model to become a tool that could serve millions of people as they strive to consume news that is as unbiased as possible.

9 Conclusions

The goal of this project was to use modern techniques to create a machine learning model that was capable of classifying news articles by their political leaning. Despite the many challenges along the way, I feel that this goal was reached. The final model consisted of an 50-dimensional pre-trained Gensim embedding layer; a bidirectional LSTM layer with 50 output units, a dropout of 0.5, and a ReLU activation function; and a dense layer to condense the LSTM output into a single number, with a sigmoid activation function for the final prediction. While a single BLSTM layer is certainly not the definition of a complex model, it easily surpasses the results achieved by the Naïve Bayes classifier in the original implementation of the project.

It is the very fact that the final model was relatively simple that leads me to wonder what could be achieved with more layers, if only the sudden increase in overfitting that I found to be associated with complex models could be avoided. With the correct values for its many hyperparameters, it is plausible that a complex model with multiple sophisticated and well-tuned layers could achieve an accuracy of 95% or higher. However, this project's implementation is by no means inadequate, and would already sufficiently serve as the backend to a mobile application, website, or browser extension. With a proper user interface, this model could serve as an invaluable tool for any person who strives to get their news from unbiased, well-informed, and diverse sources.

10 Citations

[1] Political polarization. (2020, April 4). Retrieved April 6, 2020, from https://en.wikipedia.org/wiki/Political_polarization

[2] What is a Filter Bubble? - Definition from Techopedia. (n.d.). Retrieved April 6, 2020, from https://www.techopedia.com/definition/28556/filter-bubble

[3] 1.9. Naive Bayes. (n.d.). Retrieved April 5, 2020, from https://scikit-learn.org/stable/modules/naive_bayes.html

[4] Hebb, D. O. (1949). The Organization of Behavior. New York: John Wiley and Sons.

[5] Brownlee, J. (2019, December 19). What is Deep Learning? Retrieved April 9, 2020, from https://machinelearningmastery.com/what-is-deep-learning/

[6] Artificial neural network. (2020, April 4). Retrieved April 5, 2020, from https://en.wikipedia.org/wiki/Artificial_neural_network

[7] Sharma, S. (2019, February 14). Activation Functions in Neural Networks. Retrieved April 5, 2020, from https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[8] Nelson, D. (2018, January 21). Neural Net Dropout: Dealing with Overfitting. Retrieved April 9, 2020, from https://www.datascience.us/neural-net-dropout-dealing-overfitting/

[9] freeCodeCamp.org. (2018, February 26). An intuitive guide to Convolutional Neural Networks. Retrieved April 9, 2020, from https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/

[10] Understanding LSTM Networks. (n.d.). Retrieved April 10, 2020, from https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[11] Vanishing gradient problem. (2020, March 31). Retrieved April 10, 2020, from https://en.wikipedia.org/wiki/Vanishing_gradient_problem

[12] Zhou, C., Sun, C., Liu, Z., & Lau, F. (2015). A C-LSTM Neural network for Text Classification. doi: 1511.08630

[13] Zhou, P., Qi, Z., Zheng, S., Xu, J., Bao, H., & Xu, B. (2016). Text Classification Improved by Integrating Bidirectional LSTM with Two-dimensional Max Pooling. doi: 1611.06639

[14] Rao, A., & Spasojevic, N. (2016). Actionable and Political Text Classification using Word Embeddings and LSTM. doi: 1607.02501

[15] PRAW - The Python Reddit API Wrapper. (n.d.). Retrieved February 6, 2020, from https://praw.readthedocs.io/en/latest/

[16] Newspaper3k: Article scraping & curation. (n.d.). Retrieved February 6, 2020, from https://newspaper.readthedocs.io/en/latest/

[17] Fhamborg. (2016, September 17). fhamborg/news-please. Retrieved February 27, 2020, from https://github.com/fhamborg/news-please

[18] How AllSides Rates Media Bias. (2019, October 3). Retrieved April 14, 2020, from https://www.allsides.com/media-bias/media-bias-rating-methods

[19] Media Bias Chart. (2019, November 19). Retrieved February 27, 2020, from https://www.allsides.com/media-bias/media-bias-chart

[20] sklearn.model_selection.train_test_split. (n.d.). Retrieved April 11, 2020, from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

[21] Stemming and lemmatization. (n.d.). Retrieved April 6, 2020, from https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

[22] Fisseha Berhane, PhD. (n.d.). Retrieved April 7, 2020, from https://datascience-enthusiast.com/DL/Operations_on_word_vectors.html

[23] RaRe-Technologies. (2018, March 16). RaRe-Technologies/gensim-data. Retrieved March 1, 2020, from https://github.com/RaRe-Technologies/gensim-data

[24] Géron Aurélien. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: concepts, tools, and techniques to build intelligent systems. Sebastopol, CA: OReilly Media, Inc.

[25] Wei, J., & Zou, K. (2019). EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), 2–2. doi: 10.18653/v1/d19-1670

11 Appendix

Visit https://github.com/AveryVine/Honours-Project to view or download the source code for this project.