12-2017

Analyzing Political Bias through a User-Friendly Interface

Colin Lightfoot

Analyzing Political Bias through a User-Friendly Interface

Colin Fox Lightfoot

Advisor: James Deverick

The College of William & Mary

Abstract

Many news outlets report stories shown with biased undertones that mislead readers to believe one story over another about the same event. To help people delineate between liberally- and conservatively-biased news articles, we created a website which uses a recurrent neural network with long short-term memory nodes trained to identify bias found in news articles. The network achieved an F1 Score of 0.76, and is used to provide one liberally-biased article and one conservatively-biased article side-by-side for a user to read when the user searches for a specific news story.

Acknowledgement

I would first like to thank the advisor, James Deverick, for his guidance and unwavering support throughout the undergraduate career. Without his support, this honors thesis would have never come to fruition and I would have never had such an amazing experience within the Computer Science Department. I would also like to thank Dr. Timothy Davis, Dr. Robert Lewis, and Dr. Mainak Patel for agreeing to serve on the honors committee. I give further thanks to Dr. Robert Lewis for finding the time to help me ensure the machine accepts input in order to classify the bias analyzed from scraped articles. I am grateful for the Keras API documentation and the Stack Overflow community for providing guidance on how best to approach this project and for helping me fix bugs that arose in the code. Lastly, I thank my friends and family for supporting me throughout my academic endeavors and the emotional turbulence that came with them.

Contents

1 Introduction

Many of today's news outlets report stories are shown with biased undertones that lead readers to believe one perspective of a story over another. When journalists provide a one-sided view of a story, not only are they overlooking (and possibly concealing) facts, but so too may they sway their listeners to believe these partial interpretations of news. Therefore, in order to see many sides of a multi-faceted issue, one must look at multiple news sources and piece their stories together to gain a more comprehensive understanding of the reported event. Unfortunately, most people either lack the ability or the will to spend the time needed for piecing together their own version of the story before deciding which, if any, version of the story to believe.

Many researchers have begun programming machine learning algorithms for key concepts in textual analysis, especially for classifying text using semantic analysis. While there are machine learning algorithms that classify based off few-to-several characteristics, there exist more complex algorithms and structures which are able to classify complex objects (e.g. texts) using higher-dimensional storage and calculations to analyze many characteristics during the classification process.

Neural networks are one of these more complex structures, making them one of the favored machine learning algorithms for semantic analysis. Neural networks are designed to "learn," mimicking our current understanding of human learning. After learning, neural networks are able to classify input data in a manner that follows how they were trained, involving the characteristics stated before. The results of many well-trained neural networks have been surprisingly accurate and have led to some unforeseen breakthroughs in classifying complex subjects. As our work further illustrates, neural networks can detect nuances in texts to determine such complexities as an author's political stance on the topic.

1.1 Motivation for Thesis

Not only is detecting political bias an intriguing concept, but it is an important skill to learn when intending to identify the truth in one's statement. Detecting political bias could also serve an

important purpose in bridging the gap between Americans and how they perceive political events and figures with a click of a button. It would be important to identify if some news outlets are hiding details about an event to put their own bias into the story.

Almost all news outlets have an associated bias to themselves. [1] Unfortunately, most Americans do not have the time, or even the care, to look at multiple news sources to see as many sides of the story as possible. Some people go as far to view only one side of the political spectrum's news, which leads to misunderstandings on a national level and a more polarized country that is already deeply-divided on a fundamental level. [2] Some have even called political parties modern-day tribes due to the fundamental differences between them. The goal of this project is to bring people to a common ground, ensuring everyone is informed on how popular liberal and conservative news outlets view an event so everyone, regardless of political stance, can understand the viewpoints of one another without the impeding reliance on one news outlet over another.

Another reason for starting this project is because Americans should have easy access to publicly available knowledge, and, when analyzing complex topics like political nuances, determine the bias found in information presented or researched. Binary classifications can help make this a reality. Therefore, learning to use machine learning algorithms to detect at least political bias found in texts is a major leap towards achieving the goal.

1.2 Outline

Section 2 discusses related work. Section 3 provides a brief introduction to neural networks, including subsections on neural network sequences, preprocessing said sequences, feedforward neural networks, and backpropagation. Section 4 covers recurrent neural networks, including the types of nodes they use, their known uses, and the pros and cons of using them compared to other algorithms. Section 5 discusses how the final RNN was defined, how to use the project, and how the project works. Section 5 also provides snippets of the code with explanations of how these snippets were used within the project. Section 6 details the techniques used to evaluate the validity of the neural network model, as well as possible reasons for the final model's level of accuracy.

Section 7 discusses the conclusion reached from this project. Section 8 discusses future work that can be built off of this project. Section 9 lists all the references used throughout this project. The Appendix has all the code used to implement the political bias analyzer as well as the experiments performed to tune the final model' hyperparameters.

2 Related Work

Neural networks have been proven able to classify the political bias of texts with astounding results. One study by Stanford researchers Arkajyoti Misra and Sanjib Basak showed that Recurrent Neural Network (RNN) models with Long Short Term Memory (LSTM) nodes can be used convincingly to predict the implicit political bias found in some texts, even if there were no words that indicated any conservative or liberal ideologies. [4] They created two neural networks models based on different datasets: one dataset is extracted from speeches of US congressional floor debates known as the Convote dataset [5], while the other is a collection of sentences on multiple sociopolitical issues by US presidential candidates in recent history known as the Ideological Books Corpus (IBC) dataset. [6] The first model did not perform well because there was little overlap in content which led to severe training challenges and a good portion of the data did not actually show any political bias that could be detected by a human being. The second model performed more accurately than the first shown by a higher F1 Score. Neither models were evaluated by us, as the two researchers did not provide enough explicit details on how their networks were constructed or used to train on and test the data.

Iyyer et. al. performed a study on both datasets as well, sorting the Congressional debates based on party affiliation instead of ideology, yielding a model with higher accuracy (70.2%) than their other model trained with the IBC dataset (69.3%). [8] Training their first model based on party affiliation led to noise between moderates in either Party and views in some debates that would usually not correlate with their respective Parties, such as a moderate Republican agreeing with a liberal position on increased gun control. Iyyer et al. also acknowledged the fact that the sarcasm and idioms found throughout their dataset could potentially add noise that could also lower their model's accuracy. They also trained and tested both datasets with models that make predictions based off the frequencies of each word, known as bag-of-words models. Iyyer et al. found that bag-of-words models were less accurate than RNNs, and larger datasets (or more training data) with shorter sentences are the best datasets to train RNNs. Also, their models had difficulty predicting statements with negations (e.g. "should not" compared to "should") correctly. Negations are still a

common problem for natural language processing and are presumed to cause much of the noise in models' predictions.

The conclusions reached from both groups of researchers are uncongenial when choosing which dataset to use: one found the Convote dataset created a less accurate model, while the other group found the IBC dataset created a less accurate model when the group sorted the Convote dataset a particular way. Considering the fact that the IBC dataset is fairly straightforward to implement, we chose to use the IBC dataset [7] over using the Convote dataset to train the RNN model. We chose to not use the samples labeled as "Neutral" and instead trained and tested our model on the 3726 remaining samples within the IBC dataset. Both groups do highlight how RNNs perform increasingly better than traditional bag-of-words models when there is an increasingly larger dataset from which to train machine learning models.

3 A Brief Introduction to Neural Networks

A neural network (NN), also known as an artificial neural network (ANN), is defined by Dr. Robert Hecht-Nielsen, inventor of one of the first neurocomputers, as “a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs”. [9] Neural networks are mathematical algorithms, or actual hardware, that are modeled off the mammalian cerebral cortex’s neuronal structure but with many fewer processing units compared to the billions of neurons in a mammalian brain. [10]

Neural networks are usually organized into layers that are each made up of a number of nodes that are interconnected with the other layers via weighted connections. A neural network typically contains an input layer, a number of hidden layers, and an output layer. The input layer takes in the data and sends said data to various nodes in one or more of the hidden layers through the system of weighted connections discussed above. The data is transformed through the various hidden layers until it reaches the output layer where the neural network returns an output. [10]

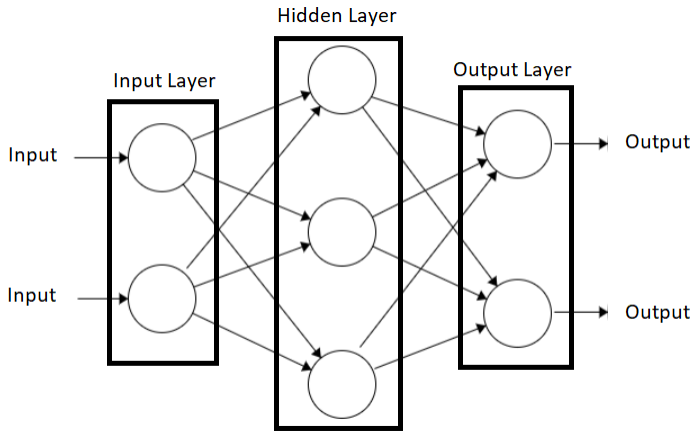

Figure 1: A basic neural network.

Each node contains an activation function in order to produce output. A node’s activation function φ takes in the summation of the node’s inputs and their respective weights as input and produces the node’s output. [12] There are various activation functions that are chosen by the model’s

designer. We used the softmax activation function for the node in the final layer, defined as

$$softmax(n) = \frac{e^n}{\sum e^n}.[11]$$

The softmax activation function outputs a value between 0 and 1 inclusive, and the RNN measures loss via the categorical cross entropy which needs output values between 0 and 1 to classify each sequence. The small range highlights the inaccuracy rate between a model's predicted values on test data and the test data's labelled values. The neural network's nodes undergo training through a learning rule which modifies the weights of the connections between nodes according to the data inputted during the training of the neural network. [10] Basically, a neural network learns to recognize certain attributes by example, similarly to how we learn to classify whether a certain creature is a dog or a cat. The learning rule mainly used, and used for this project, is known as backpropagation and is discussed in Section 3.4.

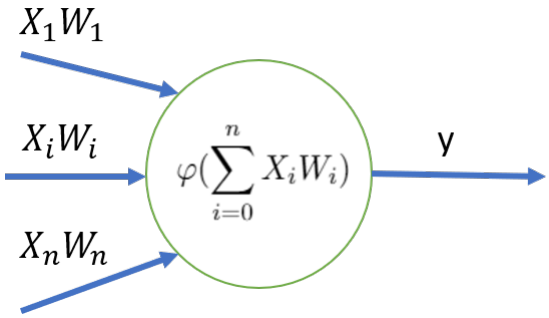

Figure 2: A basic node.

Training with this learning rule is repeatedly cyclically through cycles known as time steps or epochs (each time a network is given more input). With this learning rule, the neural network makes guesses on which of the two classifications an input is and adjusts its weighted connections accordingly to become more accurate. The weighted connections are adjusted by the process known as backpropagation. Backpropagation uses gradient descent to steadily adjust a neural network's weights after each epoch: this process is discussed further in Section 3.4. The more epochs the more accurate the neural network model is on classifying the given dataset (note that to

ensure the model is the most accurate when given input of the same type from a different dataset, verification performed by a technique known as cross validation - described in Section 6.1 - is tested against the model). [10] Once the model is trained within the researcher's accepted level of accuracy, the neural network can be used to take in data and output its defined classifications without readjusting the weights. The model can also be retrained to gradually refine its accuracy with further backpropagation.

3.1 Sequences

The input to an RNN is known as a sequence. An input sequence is represented as $(x^1, x^2, ..., x^S)$ and a target sequence is represented as $(y^1, y^2, ..., y^T)$, where each x^t, y^t are real-valued scalars or vectors. The variables S, T represent the maximum allowed length of any input sequence where $S, T \geq 1$. Therefore the input and output may be single data points or high-dimensional vectors and their respective dimensions do not have to match.

A training set is a set of examples where each example composed is usually a (input sequence, target sequence) pairing. This project's dataset's examples are composed in a (target sequence, input sequence) pairing. The data points neural networks predict are denoted as $\hat{y}^t$. Sequences' x^t data points, except for some Recurrent Neural Network (RNN) models, denote data across a span of time. [13]

RNNs have recently been shown to accurately use non-temporal sequence data for analyses, such as applications to genetic data, [14] as long as each sequence has a defined order. RNNs can also take in words and apply them as a sequence of data using natural language processing. [13] For instance and assuming no preprocessing (see below) is done, the word sequence, "I like Ike", would map as: $x^1 =$ "I", $x^2 =$ "like", $x^3 =$ "Ike". This process of breaking down word sequences is known as tokenizing. Word sequencing is useful when one is trying to analyze the sentiment of text or categorizing text into one of two categories, such as what this project accomplishes. The strings of text are then hashed into unique numbers so the strings can be used in the model's mathematical algorithm.

3.2 Preprocessing Sequences

Preprocessing is useful to perform before word sequencing in order to train and test on similar data. For instance, without preprocessing the three sentences “I like Ike”, “I like Ike.” and “i like ike” would yield three different data points for “Ike” (“Ike”, “Ike.”, and “ike”), whereas if each sentence’s uppercase characters were transformed into lowercase characters and all punctuation was striped then there would be only one data point for “Ike” (“ike”). Having a lower variety of data when training a neural network on thousands or more words lowers the accuracy of neural network’s predictions significantly. The lower accuracy is due to a lack of storage in a neural network to accommodate for all the different versions of words. Neural networks also need to pad the input in order to train or test the model or make a prediction. This is because neural networks require each input to be of the same size in order to accept the input.

RNNs rely on padding and cutting, especially for textual data when analyzing word sequences. Padding is adding a certain character a bunch of times to either the beginning (prepending) or the end (appending) of statements in a dataset so all statements in the dataset have the same amount of characters and can therefore be accepted as input by the RNN. Cutting is analyzing only a certain number of characters in a statement in order to ensure all statements have the same number of characters. [16] In this project, a large amount of padding was used to ensure large text articles could be run in the RNN model in order to make a prediction on said articles’ political biases. This assurance on text size allows this thesis’ user interface to input almost every article it finds when scraping the web into the RNN to be analyzed.

3.3 Feedforward Neural Networks

Feedforward networks are neural networks whose structure as a directed graph has no cycles. The absence of cycles allows the layered formation seen in neural networks. The formation of layers allows output to be directed from one layer to the next, organizing the network’s structure. The input x to a feedforward network initially sets the values of the lowest layer, shown in Figure 3 as the “Input Layer”. Each higher layer is then successfully computed until output is generated at

the topmost layer $\hat{y}$, shown in Figure 3 as the "Output Layer". These networks are mostly used for supervised learning tasks, such as classification. RNNs are the opposite of feedforward networks, as they can rely on their internal memories to process sequences of input. This distinction is important to note, as cycles give RNNs their ability to accurately analysis texts and will be further discussed in Section 4. These types of neural networks "learn" by iteratively updating each of the weights to minimize a loss function, $L(\hat{y}, y)$, that focuses on the distance between the output $\hat{y}$ and the target y. The most popular algorithm for teaching these networks is known as the process called "backpropagation". [18]

3.4 Backpropagation

Backpropagation, short for "backward propagation of errors" [18] is the most successful algorithm to date for training neural networks. This algorithm uses the chain rule to calculate the derivative of the loss function L with respect to each parameter in a network in order to perform the adjustment of weights within said network using gradient descent. [13] This is done by calculating the gradient of the network's error function with respect to the neural network's weights.

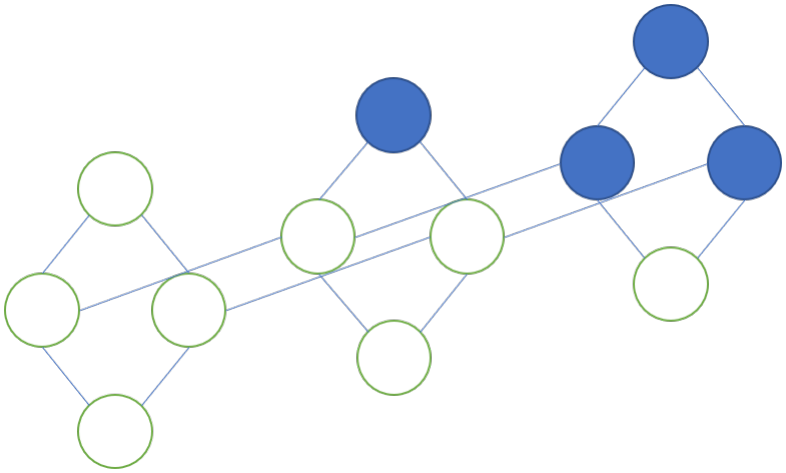

Figure 3: White nodes are the initial incorrect predictions, and the gradients are blue/black. The gradients are calculated starting from the final (rightmost) layer and move to the initial (leftmost) layer under backpropagation.

The “backwards” part of the term highlights the fact that gradient calculation and weight adjustment are first performed on the network's final layer of weights with the gradient calculation and weight adjustment of the network's first layer of weights being performed last. This allows the algorithm to reuse partial computations of the gradient from one later into the computation of the gradient in the previous layer, allowing for a backwards flow of the error information throughout the network. This backwards flow of error leads to efficient gradient computations at each layer rather than calculating the gradients of each layer separately. [18]

To calculate the gradients in a feedforward neural network, first an example is propagated forward as described in Section 3.3 to produce a value v_j at each node and outputs $\hat{y}$ at the topmost layer. The the loss function value $L(\hat{y}, y)$ is computed at each output node k. Then for each output node k, we calculate

$$\delta_k = \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}k} \cdot \varphi'_k(a_k),$$

where a_k is the activation function results for the output node. Given the value δ_k for each output node, for each node in the layer before we can calculate

$$\delta_j = \varphi'(a_j) \sum_k \delta_k \cdot w_{kj}$$

where w_{kj} represents the weight for input node j for incoming node k, $\varphi(x)$ is the activation function, and a_j is the activation function results for input node j. [13] [18] This calculation is performed iteratively through each earlier layer to yield δ_j for every node j given the δ_k values for each node in the later layers connected to j. Each value δ_j represents the derivative of the total loss function with respect to the node's incoming activation:

$$\delta_j = \frac{\partial L}{\partial a_j}.$$

And, given the values v_j calculated during the forward pass and the values δ_j for each node connected to node j that was calculated during the backward pass, the derivative of the loss L with

respect to a given weight is

$$\frac{\partial L}{\partial w_{kj}} = \delta_j v_k.$$

The loss is important to study, as the resulting scalar value is the amount weight w_{kj} is adjusted in the current backward pass. [13]

4 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are neural networks designed to make use of sequential information. The dependence that inputs and outputs have on one another in neural networks is useful for predicting what information might come next. RNNs are *recurrent* because they perform the same computations for every element in a sequence while the output is dependent on the model's outputs from previous epochs. The output can be dependent on previous epochs because the RNN's nodes can store previous data that can be used in future epochs' calculations. [15]

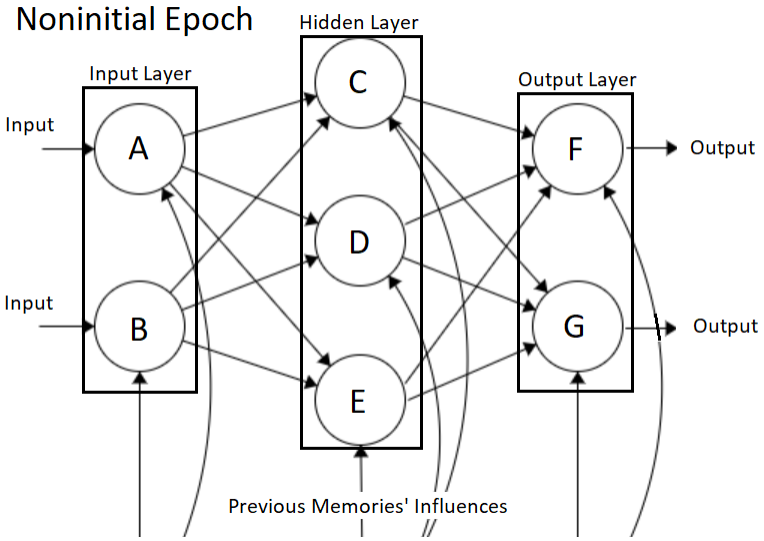

Figure 4: A noninitial RNN epoch.

Figure 5 shows a RNN being unfolded, showing that the number of layers in the network is equivalent to the number of words in the input sequence. In the figure below, x_t is the input at time step t, s_t is the hidden state at epoch t, and o_t is the output at time step t. The data is stored in the node's state during a particular epoch s_t of a RNN and is calculated based on the previous hidden state and the input at the current time step:

$$s_t = f(Ux_t + Ws_{t-1}),$$

where U is the current sequence's input and W is the same node in the previous epoch's data which is also being used as input. The output o_t of a RNN (e.g. the prediction of the next word in a sentence) would be a vector of probabilities across the remembered words of previous time steps stored in the model: $o_t = \text{softmax}(Vs_t)$, where V is the non-regularized output of s_t.

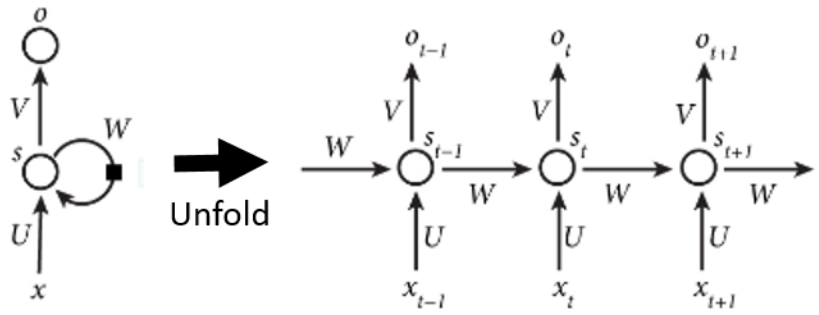

Figure 5: An unfolded recurrent neural network.

Our model stores a large number of words and tries to predict whether the sequence of text is largely liberally or conservatively biased based on the inputted words and outputs a scalar value where [0, 0.5) means the text is conservatively biased, 0.5 means the text has no noticeable bias, and (0.5, 1] means the text is liberally biased. [15]

4.1 Long Short Term Memory Nodes

Traditional RNNs experience gradient calculation problems during the gradient backpropagation phase where the gradient is multiplied too many times (meaning the network has too many time steps) by the weight matrix used with the connections between the nodes of the recurrent hidden layer. In other words, the size of the weights can impact an RNN's training. For instance, if the values in the weight matrix are less than 1, then the multiplied gradients can approach zero quickly making the training pace incredibly slow or stop learning altogether. This predicament is known as the vanishing gradient problem. On the other hand, if the weighted matrices are much larger than 1, or there are many time steps, then the calculated gradients can become so large that the model's learning diverges and leads to an inaccurate model. This phenomenon is known as the exploding

gradients problem. Both of these issues led to the creation of the Long Short Term Memory (LSTM) node that is composed of a memory cell. [20]

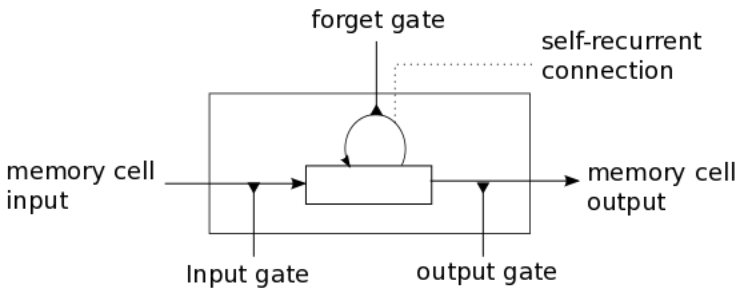

Figure 6: A simple LSTM node. [20]

A memory cell is composed of at least four attributes: an input gate, a self-recurrent connection (a pointer to itself), a forget gate, and an output gate. The self-recurrent connection has a weight of 1, preventing any parameters and input from causing either the vanishing or exploding problem while allowing a memory cell's state to remain the same from one time step to another. Each gate controls the interactions between the memory cell and other nodes. The input gate, for instance, controls whether or not an incoming signal can alter the state of the memory cell or block it while the output gate controls whether or not the state of the memory cell can affect other nodes. The forget gate controls whether to run the self-recurrent connection and remember the previous state or not run the connection and forget its state.

The following equations detail how a memory layer of nodes is updated every time step t. The variable x_t is the input to a memory layer at time t. Variables $W_i, W_f, W_c, W_o, U_i, U_f, U_c, U_o$, and V_o are the weight matrices that influence the nodes (as seen in Figure 5), $\varphi(x)$ represents the activation function as discussed in Section 3, and b_i, b_f, b_c, and b_o are the bias vectors received from the gates. The value for the input gate i_t at time t is computed like so:

$$i_t = \varphi(W_i x_t + U_i y_{t-1} + b_i).$$

The value for the forget gate activation function f_t at time t is computed as:

$$f_t = \varphi(W_f x_t + U_f y_{t-1} + b_f).$$

The forget gate activation function f_t at time t is necessary to calculate the new state of each cell C_t. The value of the output gates at time t are computed with the following function:

$$o_t = \varphi(W_o x_t + U_o y_{t-1} + V_o C_t + b_o).$$

The value of the output gates at time t are necessary to generate a nodes output y_t at itme t. Candidate, or possible, values $\widetilde{C}$ are also calculated to readjust the state C_t at time t:

$$\widetilde{C} = tanh(W_c x_t + U_f y_{t-1} + b_f).$$

The new state of each cell C_t at time t is then readjusted with the above-calculated variables:

$$C_t = i_t \times \widetilde{C}_t + f_t \times C_{t-1}.$$

This new state allows us to calculate the output y_t at time t with the following equation:

$$y_t = o_t \times tanh(C_t).$$

Therefore, from an input sequence $x_1, x_2, ..., x_n$ the LSTM nodes will produce an output $y_1, y_2, ..., y_n$ that is then averaged over all the time steps to create the output y for each node. [20]

4.2 Known Uses

RNNs have shown great success in many natural language processing tasks, from predicting stock prices [22] to studying an artist's lyrics and writing lyrics with the artist's lyrical style. [23] The most commonly used RNNs are ones that use LSTM nodes, like the ones this project uses. This is

because LSTM-based RNNs can capture long-term dependencies, such as remembering the most frequently used words, that prove beneficial to predicting data based off of specific periods in time (i.e. stock price predictions). And, in the way humans learn a language through observing a set of words multiple times, neural networks can "learn" to generate lines of text by using words or sets of words most frequently used in a dataset.

The neural Turing machine (NTM) combines RNNs with addressable memory in order to perform complex algorithmic tasks, such as sorting. The RNN part of the NTM executes the task while the Turing machine part of the NTM uses write and read heads to store and read memory from a memory buffer, such as a tape. [13]

RNNs have even been implemented to generate captions for pictures given a training set of images x and target captions y. [24] Sutskever et al. were even able to design an RNN that translated text from English to French accurately. [25] With a large enough training set that provides an input x and a target y, RNNs can be trained to output fairly accurate predictions on what the model designer wants to predict. Of course RNNs are not suitable for all tasks, such as spam filters or image recognition. But, for this project, RNNs ability to hold previous epochs' data, their ability to classify sequential information, and their proven abilitis for textual analysis make them well-suited for this project.

4.3 RNNs Compared to Other Models

Standard vector machines (SVMs) and Naïve Bayes models are less useful for most textual analyses topics when compared to the accuracies RNNs have produced. But these models are good for such textual analyses topics as constructing spam filters. [27] This is because neither learn any structure and instead learn bag-of-words representations. Therefore, with an SVM, the only information usually received (based on the feature selection chosen) is knowing the words in the input and how frequently they are used. But because RNNs learn structures, you get this information as well as the context in which the most frequent words (denoted as keywords below) are used. Iyyer et al. tried comparing bag-of-words models to RNNs and found RNNs to be clearly

more accurate models for textual analysis. [8] Compared to other neural networks, RNNs are more aptly designed for textual analysis.

While RNNs learn to recognize the long-term time-dependencies (contexts) in which keywords are used, convolutional neural networks (CNNs) learn the exact situation in which the keywords are used. This is because RNNs are time-dependent whereas CNNs are structure-dependent. For example, if a CNN is learning to classify texts as either liberally or conservatively biased, the model looks for parts of sentences with learned keywords that it knows and pieces them together in order to base its conclusions. RNNs instead look to see the order in which the words are processed (e.g. if a keyword is preceded by a certain other keyword learned to change the initial keywords meaning), placing more importance on the latest usage of a keyword, to determine a text's bias. [26] Since CNNs must base their classifications on known word sequences rather than approximating their classifications based on the context in which certain keywords were remembered to have been labelled, they are not the best for textual analysis. Compared to other time-dependent machine learning algorithms, RNNs are still favored for textual analysis problems.

Markov chains and hidden Markov models (HMMs) are also time-dependent algorithms used for classifying input into distinct states. But their approaches to sequence classifications are limited because they inefficiently scale in time the with the number states S are used at an asymptotic rate of $O(|S|^2)$. This is because more states are needed as the amount of input sequences increases since each hidden state in an HMM is dependent on previous states. And the table keeping track of the probability of moving between states is of size $|S|^2$. Therefore, when a lot of input is placed in a Markov model, computation becomes impractical, preventing a model from learning a large set of contexts per keyword. [13] The lack of contexts then would lead any HMM to be less accurate compared to an RNN model since an RNN model can capture many more contexts per keyword.

4.4 LSTM Compared to GRU Nodes

A Gated Recurrent Unit (GRU) takes in the current input and its previous state to outputs a new state similar to how LSTM nodes work. The main computational difference between the two is

that a GRU lacks a memory unit and instead exposes its data without any control, lacking second non-linearity function unlike LSTM nodes have to compute their outputs. GRUs are also more computationally efficient than LSTM nodes due to the lack of a second non-linearity function and the fewer amount of parameters used to train them. GRU tradeoff has not been fully explored, but studies show LSTM nodes and GRUs are comparable in performance. Therefore, tuning other hyperparameters is more crucial than deciding to use either LSTM nodes or GRUs in an RNN. [17]

5 Implementation of Project

5.1 Implementing the RNN

To train the LSTM-based RNN for this project, we first tuned the hyper-parameters to specific scalar values and then run LSTM-RNN.py. We repeated the process until we achieved the highest accuracy possible with similar percentages for both the `evaluate()` and `validate()` methods (this is elaborated in Section 6.1). The hyperparameters final values were discovered through repeated trial and error (the experimental results can be found in Appendix D). We used a multiple-layered RNN for the classification task. The layer after the input layer was an embedded layer which vectored each word in a sequence for processing. The next layer was comprised of 300 LSTM nodes to allow the neural network to mimic long-term memory. A recurrent dropout layer was then added with the dropout rate of 0.20 to allow for some memory loss at the end of each epoch. Finally, a softmax activation layer was added to provide output known to closely approach its correct classes' attributed scalar value. The binary cross entropy loss was then minimized by the 'Adam' optimizer. The RNN model was trained on 67% of the dataset while the accuracies and F1 Score were calculated on the remaining 33% of the dataset. To determine the correct number of epochs to train our models on, we used the keras API's early stopping callback feature.

Early stopping is the ability for a model to stop training once a specific trait (we monitored the average loss at the end of each epoch, `val_loss`) of the model has stopped improving to prevent overfitting or underfitting which allows us to disregard determining a lower-bound and an upper-bound for the number of epochs necessary to train the model to its highest possible accuracy. Overfitting is when an RNN is trained on the training dataset too many times and cannot accurately predict many other inputs besides the samples within the training dataset, rendering the model increasingly innaccurate the longer it is trained. Underfitting is when an RNN is not trained enough, making it unable to find and classify patterns found within new data input, making the model inaccurate unless it is further trained. The model continues to train until the specified trait stops improving above a chosen minimal improvement rate regardless of whether the number of epochs desired for training is a larger number than the epoch in which the training stops. We can therefore

enter a large number for the amount of epochs desired to train the model while the model prevents itself from running more epochs and over-training. This prevents the need to determine a lower-bound and an upper-bound for the number of epochs necessary to train the model. Early stopping therefore increases the model's understanding of the patterns learned from the training set while ensuring the model is not overfitting the training data by using only the number of epochs necessary to remain above the specified trait's chosen minimal improvement rate until the model falls below this improvement threshold.

5.2 Finding the Most Accurate Hyperparameters

All variables except the number of epochs were chosen based off testing each variable until a definite local, and possibly also global, maximum was reached. We started with an RNN architecture that had hyperparameters which were tested sequentially, where each hyperparameter was tested until the most accurate model was reached before testing the next hyperparameter. This allowed us to run an optimal amount of model trainings to yield us the most accurate results with the shortest amount of experiments possible. We first trained the model by only changing the batch size until we found 32 to yield the most accurate model. Next was hidden layer size with 300 nodes yielding the most accurate model. Then the amount of words to remember (noted as the `TOP_WORDS` variable within Appendix A) that yielding the most accurate model was found to be 5000 words. We then ran a few experiments changing the embedded layer's embedded vector length and found that a embedded vector length of 112 yielded the most accurate model. Finally we experimented with the dropout layer's dropout rate and found that a dropout rate of 0.20 yielded the most accurate model. We tried adding another dropout layer, but found that an extra dropout layer lowered the model's accuracy. From our research, we understand that one hidden layer is sufficient for the large majority of problems and therefore did not attempt to create another LSTM node hidden layer. [29] Tables showing the results of our experiments can be seen in Appendix D. We would like to highlight the fact that each of our hyperparameter's values in our most accurate model may have only be their respective local maximums rather than their global maximums. This means there might be

values for each hyperparameter in our final RNN that could yield more accurate models than our final model.

5.3 Implementing the Web Page

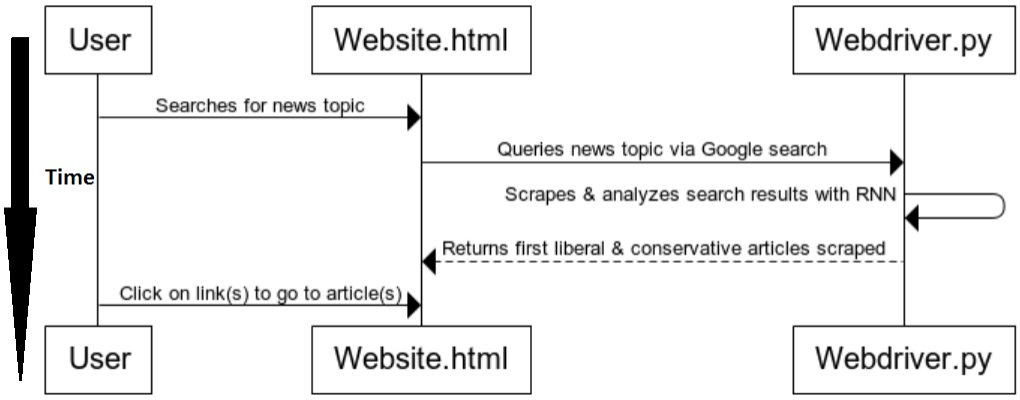

Figure 7: How the multiple files interact.

The code will automatically stores the latest algorithm as a file to be read by `Webdriver.py`. Following Figure 7 above, to run the project, the user runs `Webdriver.py` and a Python Flask-powered local host will begin to run. Next, the user uses a web browser to reach the local host to arrive on `Website.html` (shown in Figure 8), the website interface for this project.

Figure 8: The Website.html webpage.

Now the user simply types in a news topic to begin running the background tasks of finding the most popular conservative and the most popular liberal news coverage of said topic and putting the links to both side-by-side onto the website. For instance, if one were to search the term “tax reform”, then the project would find the first result on Google’s search results based on the queried URLs’ relevancies to the user’s query. The first article classified as conservatively-biased by the RNN’s link is pasted in the website’s table’s “Most Popular Conservative View” column. The process is the same for liberally-biased articles. `Website.html` then shows the links to the most relevant article deemed conservative and the most relevant article deemed liberal as shown in Figure 9.

Figure 9: The end result of a search.

`Webdriver.py` is responsible for requesting and returning the website’s user’s search query as a link to politically-biased articles as described in the paragraph above. To complete these tasks, `Webdriver.py` opens up an invisible browser window (example screenshot is shown in Figure 10) and copies the user’s query onto the invisible browser’s Google search engine.

```
inputString = request.form['inputString']
driver = webdriver.PhantomJS() # Creates an invisible browser.
driver.get('https://google.com/') # Navigates to Google.com.
searchBarInput = driver.find_element_by_name('q') # Assigns query to Google Search bar.
```

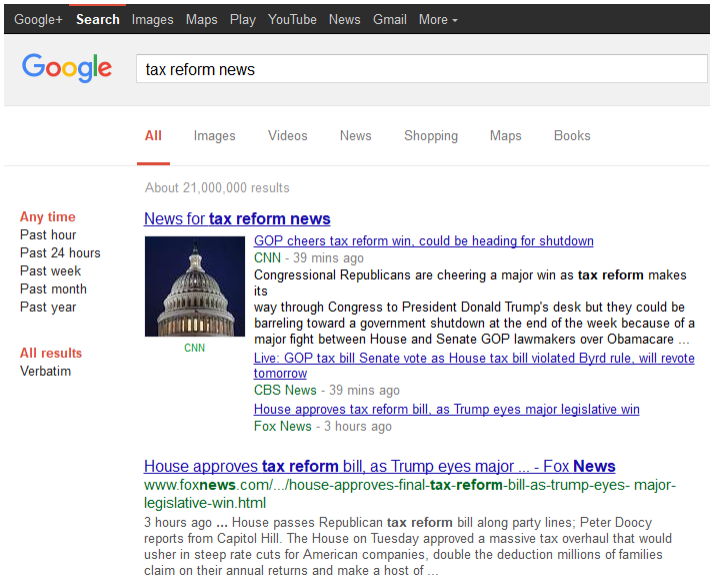

Figure 10: The invisible browser window that is scraped in the background.

`Webdriver.py` then waits for Google's results page to load the "most relevant" URLs and begins reading and opening the first accepted URL, runs the articles' text through the RNN model created. You can see the code below that would execute if a CNN news article's text is scraped and ran through the RNN.

```
    searchBarInput.send_keys(inputString + " news") # what you are searching for
    searchBarInput.send_keys(Keys.RETURN) # Hit <RETURN> so Google begins searching
    time.sleep(1) # sleep for a bit so the results webpage will be rendered
    ...
    urls = driver.find_elements_by_css_selector('h3.r a')
    for url in urls:
        ...
        if "cnn.com" in url:
            linkName = "CNN Article"
            lookAtPage = requests.get(url, headers=headers)
            soup = BeautifulSoup(lookAtPage.text, "html.parser")
```

```
            paragraphs = soup.find_all('div', {"class":"zn-body__paragraph"})
            text = ''
            for paragraph in paragraphs:
                text = text + paragraph.text
            checkURL(url, text, linkName)
            ...
```

Relevant URLs are determined based off Google's "Relevancy" search results. Only the listed news outlets' URLs in `Webdriver.py` are accepted and can therefore be analyzed by our RNN; the list is based on [21]'s list on the top-ranked news outlets. To run the RNN, it must first be opened from the file it was saved to, the article that is going to be analyzed's text needs to be preprocessed, and then the loaded model makes a prediction as seen in the code below.

```
loadedModel     = load_model('finalizedModel.h5')
...
def checkURL(url, text, linkName):
    ...
    # Preprocess article to predict its political bias
    tokenizer = Tokenizer(num_words=TOP_WORDS, split=' ')
    tokenizer.fit_on_texts([text])
    X = tokenizer.texts_to_sequences([text])
    X = pad_sequences(X, maxlen=1000)

    # Predict if the article is politically biased, and, if so, which way is it biased.
    prediction = loadedModel.predict(X)

    # Fill proper URLs based on prediction.
    if conservativeURL == ' ' and prediction[0][0] > NEUTRAL:
            conservativeURL = url
            cLinkName = linkName
```

```
            cNumber = prediction[0][0]

    if liberalURL == ' ' and prediction[0][0] < NEUTRAL:
            liberalURL = url
            lLinkName = linkName
            lNumber = prediction[0][0]
```

Basing our list of acceptable outlets on a list of top-ranked news should prevent our data scraping from being affected as much from the "Popularity" factor of Google's search engine and thus ensuring only the top-ranked news outlets are viewed. If the model calculates that the article is biased towards one side of the political spectrum than the other and the URL for the biased side has not been filled by another URL, then the spot is now filled with a URL link to the article. The RNN classifies the text by first turning each word in the article into a vector that is then run through the network and outputs a scalar value as seen in Figure 9. As seen in Figure 11, if the output value is

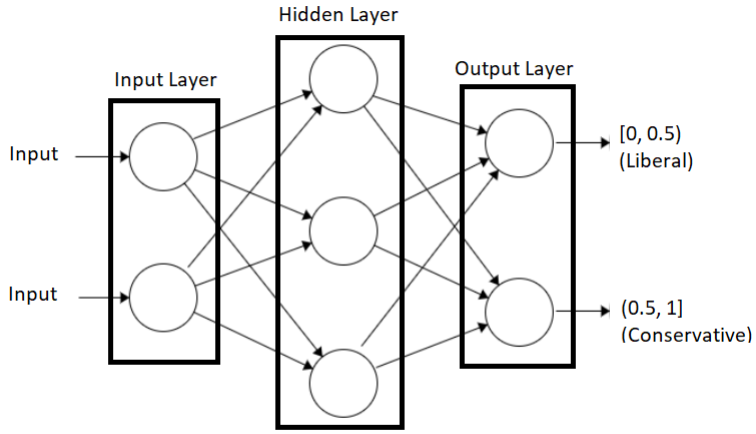

Figure 11: How the RNN determines bias.

within the range [0, 0.5) then the article is classified as having a liberal bias by our model. And if the output value is within the range (0.5, 1], then the article is classified as having a conservative bias

by our model. This process is then reiterated for each article whose URL is listed in the results and continues down the page, and sometimes even onto other results pages, until both the conservative and liberal URL link slots are filled on the website or there are no more results to analyze.

The Appendix contains copies of the actual code used to implement this project, with Appendix A containing the code to train the most recent RNN model used in the project, Appendix B containing the code that provides the functionality for opening and running the saved RNN as well as querying and collecting data to be analyzed by the RNN, and Appendix C containing the code for the website displayed in Figure 7 of which the user interacts with to run a search query on the project. All files pertaining to this thesis are stored at this URL address: `https://github.com/hydure/HonorsProject`, including the final stored RNN model used by `Webdriver.py`.

6 Evaluation of Project

In order to tune the final model to the highest accuracy achievable given a certain size of test data, we measured how our models fared under certain hyper-parameters by evaluating them against reserved samples from the test dataset. The remaining samples in the test dataset were then used to validate each model to ensure that no model was overtrained or undertrained by the training dataset. The model's accuracies for both the evaluating and validating test datasets were compared to see if their accuracy values were close to one another. This was to see if each model was a good classifier for random samples of input with unknown classifications: if a model is a good classifier for random samples, then the model can be considered to accurately classify samples. The metrics we used were each model's F1 Scores, overall accuracy, the percentage of correct conservative classifications, and the percentage of correct liberal classifications.

6.1 Measurements

A model's F1 Score is a measure of accuracy which considers both the model's precision and recall. Precision is measured as

$$\frac{\text{true positives}}{\text{false positives + true positives}}.$$

Precision demonstrates the model's ability to accurately differentiate one class from all others. For example, when at a car dealership a car dealer might say that all of the cars on the left lot are trucks when they are not all trucks. Figuring out how many of the cars that were classified as trucks are actually trucks is the amount of true positives found while the total of all the cars both correctly and incorrectly classified as trucks are is the sum of the amount of true positives and false positives. If the car dealer has a high precision, then he would almost never identify a non-truck car as a truck. Recall is measured as

$$\frac{\text{true positives}}{\text{false negatives + true positives}}.$$

Recall demonstrates the ability to recognize all objects of one class from all of the objects in the dataset. For example, a car dealer might only sell trucks but classifies only a few cars of his as trucks base off the characteristics he remembers trucks having. The incorrect classifications of trucks not being trucks are the false negatives while the correct classifications of the trucks as trucks are true positives. If the car dealer has a high recall then he would realize most of his cars are trucks.

Measuring the model's precision and recall helps show the overall strength of the model in regards to not predicting Type I and II errors respectfully. A Type I error is labeling an object as a false positive and is checked by precision, while a Type II error is labeling an object as a false negative and is checked by recall. Another way to look at this is the number of Type I errors divided the sample size plus the model's precision equals one, and the number of Type II errors divided the sample size plus the model's recall equals one as well. The formula for calculating the F1 Score is:

$$\text{F1 Score } = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision + recall}}.[28]$$

The keras machine learning API documentation we used for recording a model's precision and recall is limited and was unable to be used, whereas there was a function, `evaluate()`, listed which output the model's F1 Score for the RNN model. This function was used to calculate the final model's F1 Score, 0.74. The model's F1 Score turned out to be higher than Misra's and Basak's F1 Score when ran on the IBC dataset, 0.718, where they used a larger percentage of the IBC dataset as a training set (80% compared to the size of 67% of the dataset). [4] An F1 Score varies between 0 and 1 with the goal of getting a model to achieve a score of 1. Therefore, based on the testing dataset, the precision and recall were fairly high values, showing the model is fairly robust in classifying political bias.

A cross-validation analysis partitions a dataset into a training set and a testing set to demonstrate the accuracy of the trained model. In our project, we first evaluated our model with the testing set and then used part of the testing set against the model after evaluating the model to validate the model. The accuracy and F1 Score of each model is calculated to discover which model is the

most robust against test data and to see if the model over-fits or under-fits the training. Overfitting is when the neural network is over-trained on the training data, meaning the model classifies the trained samples well but cannot classify samples from the test dataset as accurately as it could have if it was not over-trained. Underfitting is when the model cannot grasp any patterns from the training set and therefore cannot make accurate predictions on the test dataset. Both overfitting and underfitting are undesired outcomes of either too much or too little training with data that has high levels of bias. We implemented the early stopping callback provided by the keras API when training our models to prevent these issues from occurring.

But after running a cross-validation analysis on the model to ensure we did not over-fit our models, the final model's accuracy against the testing dataset partition was 68.89%. This model is therefore more accurate than the 50% accuracy assumed to occur from randomly guessing biases in articles. The 60.89% accuracy resulted from the larger liberal test dataset yielding an accuracy of 65.04% while the smaller conservative test dataset achieved an accuracy of 57.14% during the model's cross-validation. Therefore, given a subset of the IBC dataset was the model's only training data, the model can fairly predict whether or not a sentence hold an objectively conservative bias while having a somewhat hard time predicting whether a sentence has an objectively liberal bias while still being more accurate than random guessing in regards to bias prediction. Changing the training subset would alter the accuracy of the model, but each model received the same seed for each training phase to ensure the accuracies between models could be compared.

6.2 Possible Noise

As Iyyer et al. noted, [8] the IBC's lack of data meant the RNN model did not have a lot of data to train on compared to the amount of liberally- and conservatively-biased text available to the public. And, because the IBC dataset's samples were single sentences, there were not as many keywords and patterns to record as samples with larger amounts of texts would have. The sentences were also long, allowing most sentences to likely lose their meaning as contexts and keywords were recurrently forgotten when the RNN model was learning the latest input's keywords

and context. But, after only being tested against samples containing similar singular long sentences and classifying them fairly accurately, there is a high probability that the articles ran through the model when the server is running will have more than enough keywords and contexts for the RNN model to accurately classify the texts.

The model's conservative classification accuracy being lower than its liberal classification accuracy could be attributed to the fact that there were less conservative samples. Having a larger training set usually yields more accurate models, as there would be a larger sample size representing the entire population of data compared to a smaller sample size that, on average, does not represent the population of data as accurately. Since the model can only remember so many contexts and keywords, some noise could be due to the model possibly being tested against a larger liberally-biased test set than it was a conservatively-biased test set. This is because the greater amount of liberal samples most likely lead to more liberal contexts and keywords being remembered compared to conservative contexts and keywords. We must stress that our models' trainings are limited to our training dataset's samples' data and size. We know the IBC dataset had only a few thousand short, only sentence-long samples. With such small datasets, machine learning algorithms tend to be fairly inaccurate, as you can see with our model's respective accuracies. We could not fix the small dataset issue by creating and adding our own labelled data to the dataset, as any political bias classification we would create for a sentence would be biased because we have no way to determine if our sentence classifications are accurate. Instead, we must rely on domain experts for training data. While we believe our training data is state-of-the are, there is always additional and more accurate training data that can be used to train our model.

7 Conclusion

In this work we have shown how an RNN using LSTM nodes in one of its layers can accomplish the ambitious task of determining political bias in news articles with a decent cross-validation accuracy of 58.26% and a high cross-validation F1 Score of 0.76. While there is already research on this topic, we took the idea one step further and decided to make the model easily accessible and usable to the public through a simple user interface shown in Section 5. This was accomplished by creating a website with a search bar that, when a news topic is typed in the search bar and the "Search" button is clicked, runs a query through Google's search engine in an invisible window and looks at each URL rendered on Google's results page matching one of the trusted websites and then scrapes through the entire article. Next, the scraped text is then run through an RNN model trained using the IBC dataset. The model outputs a value between 0 and 1, and if the output is less than 0.5, then the model is conservatively-biased, and if the output is greater than 0.5 the model is liberally-biased. If the article fulfills a bias that is not already filled, then its URL is shown on the web-page. This process continues until the most relevant liberally-biased article and conservatively-biased articles are found, providing the user with a fair and balanced opportunity to see both political views (if either exists) about the news topic in question.

Using the web-page, anyone can find a liberal and a conservative stance for any existing news topic being researched, ensuring users of the web-page access to a listing of relevant articles about a reported event that is statistically-proven to be fair and balanced. Exactly how a neural network is calculated mathematically and the reasoning for its design has also been discussed; the characteristics, along with explanation, of neural networks and what these algorithms actually output have been provided in this way as well. Furthermore, RNNs should be chosen over other models for textual analysis, especially in regards to sentiment analysis (e.g. political bias analyses). The importance of using LSTM nodes to mimic long-term memory has led to the RNN becoming one of the best, if not the best, machine learning algorithms to use for long-term contextual analysis. The design can be replicated in another experiment by following the setup previously described. We have created a platform for someone to search the news in a fair and balanced way, and was

accomplished by using a fairly accurate machine learning algorithm to classify articles as being liberally- or conservatively-biased on a web-page designed to perform these tasks.

8 Future Work

As they become available, we plan to use larger datasets of speeches, transcripts, and statements by people identified as having mostly liberal or conservative values due to achieve higher accuracy. Our dataset left us with many limitations that can be overcome by using datasets with larger samples, such as samples with multiple sentences, to train and test our RNN models could improve our models' accuracies rather than how we are currently training on samples composed of a single sentence to create an accurate bias prediction for paragraph-long articles. Limitations can also be further reduced by using datasets with larger numbers of samples to train our models. These datasets would make our models more accurate because training off a larger sample set normally yields a more accurate model. Limitations can be further reduced by using symmetric datasets with equal, or almost equal, amounts of liberal and conservative samples, These datasets would help lead to closer accuracy percentages between the conservative and liberal accuracies, as our future models would be learning from an equal amount of bias from each end of the political spectrum.

We may also want to base which news outlets we scrape from based on their overall trustworthiness, as described in [1], rather than scraping from Feedspot's Top USA News Websites list rankings. Already listed URLs might not need to be removed; instead, we could just add more URLs to scrape from to the current list of URLs acceptable to scrape from.

We would also like to try and classify political bias on other machine learning algorithms, such as RNNS that use a GRU node layer, CNN, ANN, SVM, or k-nearest neighbors, Markov-based and Naive-Bayes algorithms to compare the pros and cons between each major type of machine learning algorithm so we can more deeply understand and appreciate the different machine learning algorithms and their uses for textual analyses.

We would also like to further train our RNN model to be more accurate by testing new values for our model's hyperparameters and seeing the results of adding more layers to the model. We would also like to give users the ability to send updates to our model, such as when the model incorrectly predicts an article's bias, so that we may update and train the model even further on a regular basis.

This could lead to our model becoming increasingly inaccurate, as we have no guarantee the users are political bias experts nor do we know if they are being malicious or unknowingly mislabelling the articles as being biased to one side of the spectrum when it could actually be neutral or on the other end of the political spectrum.

In their research, Iyyer et al. found that certain “stop” words, such as the negations “no” and “not”, seriously impacted the accuracy of their machine learning algorithms and we are confident these negations affected our models’ accuracies as well. Therefore, when we further training the model we could attempt to add a natural language processing pre-phase to try and remove samples with these “stop” words to try to prevent our models becoming as inaccurate as the current model being used.

Designing and implementing a more approachable user interface is another possible future work. More article URL links could be displayed, or the URL link’s articles could be displayed upon search. Another possibility is clicking on a URL link and having the link’s article appear on the website. We could also create more table columns for “Neutral” articles or two more columns that displays articles which have been determined to be “More Conservative” and “More Liberal” than most articles. To add any of these three columns to the website’s table, we would define a range that would most be (0.4, 0.6) for the “Neutral” column, [0, 0.2) for the “Most Conservative” column and (0.8, 1] for the “Most Liberal” column. The user interface is also relatively slow and figuring out a way to parallel process articles through the RNN at the same time while also scraping for the next article would help decrease the search time.

9 References

[1] Jeff Desjardins. *The Least and Most Trusted News Sources in America* http://www.visualcapitalist.com/least-most-trusted-news-sources/.

[2] Carroll Doherty. *7 things to know about polarization in America* http://www.pewresearch.org/fact-tank/2014/06/12/7-things-to-know-about-polarization-in-america/.

[3] Michael M. Grynbaum. *Fox News Drops 'Fair and Balanced' Motto* https://www.nytimes.com/2017/06/14/business/media/fox-news-fair-and-balanced.html.

[4] Arkajyoti Misra and Sanjib Basak. *Political Bias Analysis* https://cs224d.stanford.edu/reports/MisraBasak.pdf

[5] Lillian Lee. *U.S. Congressional Speech Data* http://www.cs.cornell.edu/home/llee/data/convote.html

[6] Mohit Iyyer, Peter Enns, Jordan Boyd-Graber, and Philip Resnik. *The Ideological Books Corpus* https://www.cs.umd.edu/ miyyer/ibc/index.html

[7] Yanchuan Sim, Brice Acree, Justin Gross, and Noah Smith. *Measuring Ideological Proportions in Political Speeches.* Empirical Methods in Natural Language Processing, 2013.

[8] Mohit Iyyer, Peter Enns, Jordan Boyd-Graber, Philip Resnik. *Political Ideology Detection Using Recurrent Neural Networks* http://www.aclweb.org/anthology/P/P14/P14-1105.pdf

[9] Maureen Caudill. *Neural Network Primer: Part I* Feb. 1989.

[10] University of Wisconsin-Madison. *A Basic Introduction to Neural Networks* http://pages.cs.wisc.edu/ bolo/shipyard/neural/local.html

[11] Matlab. *Softmax Documentation* https://www.mathworks.com/help/nnet/ref/softmax.html

[12] Tony. *Neural Networks: The Node* http://www.neuraldump.com/2016/05/neural-networks-the-node/

http://www.neuraldump.com/2016/05/neural-networks-the-node/

[13] Zachary C. Lipton, Jon Berkowitz, and Charles Elkan. *A Critical Review of Recurrent Neural Networks for Sequence Learning*

https://arxiv.org/pdf/1506.00019.pdf.

[14] Pierre Baldi and Gianluca Pollastri. *The principled design of large-scale recurrent neural network architectures-DAG-RNNs and the protein structure prediction problem.* The Journal of Machine Learning Research, 4:575–602, 2003.

[15] Denny Britz. *Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs* http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.

[16] Denny Britz. *Recurrent Neural Networks Tutorial, Part 2 – Implementing a RNN with Python, Numpy and Theano*

http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-2-implementing-a-language-model-rnn-with-python-numpy-and-theano/.

[17] Denny Britz. *Recurrent Neural Network Tutorial, Part 4 – Implementing a GRU/LSTM RNN with Python and Theano*

http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/.

[18] John McGonagle, George Shaikouski, Andrew Hsu, Jimin Khim, Christopher Williams. *Backpropagation*

https://brilliant.org/wiki/backpropagation/.

[19] Trask. *Anyone Can Learn To Code an LSTM-RNN in Python (Part 1: RNN)* https://iamtrask.github.io/2015/11/15/anyone-can-code-lstm/.

[20] DeepLearning.net. *LSTM Networks for Sentiment Analysis* http://deeplearning.net/tutorial/lstm.html#code.

[21] Feedspot.com. *Top 100 USA News Websites* http://blog.feedspot.com/usa_news_websites/.

[22] Siraj Raval. *How to Predict Stock Prices Easily - Intro to Deep Learning #7* https://www.youtube.com/watch?v=ftMq5ps503w&t=527s.

[23] Siraj Raval. *Generate Rap Lyrics - Fresh Machine Learning #4* https://www.youtube.com/watch?v=yE0dcDNRZjw.

[24] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. *Show and tell: A neural image caption generator.* In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 3156–3164, 2015.

[25] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to sequence learning with neural networks* In Advances in Neural Information Processing Systems, pages 3104–3112, 2014.

[26] J O'Brien Antognini. *RNN vs CNN at a high level* https://datascience.stackexchange.com/questions/11619/rnn-vs-cnn-at-a-high-level.

[27] Roman Trusov. *Why would any one use Recursive Neural Nets for text classification as against SVM or Naive Bayes or any traditional statistical models?* https://www.quora.com/Why-would-any-one-use-Recursive-Neural-Nets-for-text-classification-as-against-SVM-or-Naive-Bayes-or-any-traditional-statistical-models.

[28] Adam Yedidia. *Against the F-score* https://adamyedidia.files.wordpress.com/2014/11/f_score.pdf.

[29] Doug Y'barbo *How to choose the number of hidden layers and nodes in a feedforward neural network?*

https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw

10 Appendices

10.1 Appendix A

LSTM-RNN.py:

```
import numpy as np    # Linear Algebra
import pandas as pd   # Data Processing, CSV file I/O (e.g. pd.read_csv)
import re             # Regular Expression changing (CSV file cleanup)

from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.callbacks import EarlyStopping
from keras.layers import Dense, Embedding, LSTM, SpatialDropout1D, Dropout
from sklearn.model_selection import train_test_split
from keras.utils.np_utils import to_categorical

########################### HYPERPARAMETERS ##############################

SEED = 7                      # Fixes random seed for reproducibility.
URL = 'ibcData.csv'           # Specified dataset to gather data from.
SEPERATOR = ','               # Seperator the dataset uses to divide data.
PADDING_LENGTH = 1000         # The amount of words allowed per piece of
                              # text.
HIDDEN_LAYER_SIZE = 300       # Details the amount of nodes in a hidden
                              # layer.
TOP_WORDS = 5000              # Most-used words in the dataset.
MAX_REVIEW_LENGTH = 500       # Char length of each text being sent in
                              # (necessary).
EMBEDDING_VECTOR_LEN = 112    # The specific Embedded later will have
                              # 112-length vectors to represent each word.
BATCH_SIZE = 32               # Takes 64 sentences at a time and
                              # continually retrains RNN.
NUMBER_OF_EPOCHS = 100        # Fits RNN to more accurately guess the
                              # data's political bias.
VERBOSE = 1                   # Gives a lot of information when
                              # predicting/evaluating model.
NONVERBOSE = 0                # Gives only results when
                              # predicting/evaluating model.
VALIDATION_SIZE = 1000        # The size that you want your validation
                              # sets to be.
DROPOUT = 0.2                 # Helps slow down overfitting of data
```

```
                              # (slower convergence rate)
FILE_NAME = 'finalizedModel.h5' # File LSTM RNN is saved to so it can be
                              # used for website

############################## FUNCTIONS ###################################

# Function to see what your CSV file looks after it is cleaned up
def debugAfterCleanUp(data):
    print(data)
    print(data[ data['bias'] == 'Conservative'].size)
    print(data[ data['bias'] == 'Liberal'].size)

# Checks the shape of the below four datasets
def checkShapes(X_train, X_test, Y_train, Y_test):
    print(X_train.shape,Y_train.shape)
    print(X_test.shape,Y_test.shape)

# Prints a summary of the model
def printModelSummary(model):
    print(model.summary())

# Evaluates the model
def evaluate(model, X_test, Y_test):
    score, accuracy = model.evaluate(X_test, Y_test, verbose = VERBOSE)
    print("Evaluation:")
    print(" F1 Score: %.2f" % (score))
    print(" Accuracy: %.2f%%\n" % (accuracy * 100))

# Validates the model by extracting a validation set and
# measuring the correct number of guesses
def validate(model, X_test, Y_test):

    X_validate = X_test[-VALIDATION_SIZE:]
    Y_validate = Y_test[-VALIDATION_SIZE:]
    X_test = X_test[:-VALIDATION_SIZE]
    Y_test = Y_test[:-VALIDATION_SIZE]
    score, accuracy = model.evaluate(X_test, Y_test, verbose = VERBOSE, \
                          batch_size = BATCH_SIZE)
    print("Validation:")
    print(" F1 Score: %.2f" % (score))
    print(" Accuracy: %.2f%%\n" % (accuracy*100))
    print("Getting percentage of correct guesses per political leaning...\n")

    conCount, libCount, conCorrect, libCorrect = 0, 0, 0, 0

    for x in range(len(X_validate)):

        result = model.predict(X_validate[x].reshape(1, X_test.shape[1]), \
                              batch_size = 1, verbose = VERBOSE)[0]

```

```
88          if np.argmax(result) == np.argmax(Y_validate[x]):
89              if np.argmax(Y_validate[x]) == 0:
90                  libCorrect += 1
91              else:
92                  conCorrect += 1
93
94          if np.argmax(Y_validate[x]) == 0:
95              libCount += 1
96          else:
97              conCount += 1
98
99      print("Conservative Accuracy:", conCorrect / conCount * 100, "%")
100     print("    Liberal Accuracy:", libCorrect / libCount * 100, "%\n")
101
102 def save(model):
103     model.save(FILE_NAME)           # Creates a HDF5 file to save the whole model
104     print("Model saved.\n")         # (e.g. its architecture, weights, and optimizer rate)
105
106 ############################## PREPARE DATA ##############################
107
108 # Read the data from the CSV file by column
109 data = pd.read_csv(URL, header = None, names = ['bias', 'text'], sep = SEPERATOR)
110
111 # Make all characters lowercase if they are not already
112 data['text'] = data['text'].apply(lambda x: x.lower())
113
114 # Take out all superfluous ASCII characters
115 data['text'] = data['text'].apply((lambda x: re.sub('[^a-zA-z0-9\s]', '', x)))
116
117 # Eliminate duplicate whitespaces
118 data['text'] = data['text'].apply((lambda x: re.sub(r'\s+', ' ', x)))
119
120 #debugAfterCleanUp(data);
121
122 # Preprocess texts
123 tokenizer = Tokenizer(num_words=TOP_WORDS, split=' ')
124 tokenizer.fit_on_texts(data['text'].values)
125 X = tokenizer.texts_to_sequences(data['text'].values)
126 X = pad_sequences(X, maxlen=PADDING_LENGTH)
127
128 # Declare the train and test datasets
129 Y = pd.get_dummies(data['bias']).values
130 X_train, X_test, Y_train, Y_test = \
131           train_test_split(X, Y, test_size = 0.33, random_state = SEED)
132
133 #checkShapes(X_train, X_test, Y_train, Y_test)
134
135 ############################## TRAIN MODEL ##############################
136
137 # Define the model
```

```
model = Sequential()
model.add(Embedding(TOP_WORDS, EMBEDDING_VECTOR_LEN, input_length=X.shape[1]))
model.add(LSTM(HIDDEN_LAYER_SIZE))
model.add(Dropout(DROPOUT))
model.add(Dense(2, activation='softmax'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', \
              metrics=['accuracy'])

#printModelSummary(model)

# Stops fitting the model when the improvement is negligible to
# help prevent over-fitting
earlyStopping = EarlyStopping(monitor='val_loss', min_delta=0, patience=0,
    verbose=NONVERBOSE, mode='auto')

# Fit the model
model.fit(X_train, Y_train, validation_data=(X_test, Y_test), \
          epochs=NUMBER_OF_EPOCHS, batch_size=BATCH_SIZE, \
          callbacks=[earlyStopping])

print("*" * 75)

# Evaluate the model
evaluate(model, X_test, Y_test)

# Validate the module
validate(model, X_test, Y_test)

print("*" * 75 + '\n')

# Save the model
save(model)

################################################################################
```

10.2 Appendix B

Webdriver.py

```
from flask import Flask, render_template, request, redirect, url_for
from wtforms import Form, TextAreaField, validators
from keras.models import load_model
from keras.preprocessing.sequence import pad_sequences
from keras.preprocessing.text import Tokenizer
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from string import Template
```

```
import time, requests, numpy
from bs4 import BeautifulSoup

############################ GLOBAL VARIABLES ############################

# Made these global so both @app.route functions and checkURL can access them.
conservativeURL = ' '
liberalURL      = ' '
cLinkName       = ' '
lLinkName       = ' '
errMessage      = ' '
lNumber         = ' '
cNumber         = ' '
loadedModel     = load_model('finalizedModel.h5')
MAX_REVIEW_LENGTH = 500
TOP_WORDS = 5000             # Most-used words in the article.
NEUTRAL = 0.5                # Article is predicted to not be politically biased.

############################ FUNCTIONS ###################################

# Runs article through algorithm and determines if it is politically biased and how.
# If politically biased and its biased URL isn't filled, fill its bias' URL slot.
def checkURL(url, text, linkName):
    global conservativeURL
    global liberalURL
    global cLinkName
    global lLinkName
    global lNumber
    global cNumber

    # Preprocess article to predict its political bias
    tokenizer = Tokenizer(num_words=TOP_WORDS, split=' ')
    tokenizer.fit_on_texts([text])
    X = tokenizer.texts_to_sequences([text])
    X = pad_sequences(X, maxlen=1000)
    print("Article has been preprocessed and a prediction is being made...")

    # Predict if the article is politically biased, and, if so, which way is it biased.
    prediction = loadedModel.predict(X)

    # Fill proper URLs based on prediction.
    if conservativeURL == ' ' and prediction[0][0] > NEUTRAL:
            conservativeURL = url
            cLinkName = linkName
            cNumber = prediction[0][0]

    if liberalURL == ' ' and prediction[0][0] < NEUTRAL:
            liberalURL = url
            lLinkName = linkName
            lNumber = prediction[0][0]
```

```
59
60  ############################## FLASK APP ###################################
61
62  app = Flask(__name__)
63
64  class SearchForm(Form):
65      inputString = TextAreaField('', [validators.DataRequired()])
66
67  @app.route('/')
68  def index():
69      return render_template('Website.html', cLinkName=cLinkName, lLinkName=lLinkName, \
70                             conservativeURL=conservativeURL, liberalURL=liberalURL, \
71                             errMessage=errMessage, lNumber=lNumber, cNumber=cNumber)
72
73  @app.route('/results', methods=['GET', 'POST'])
74  def results():
75      global conservativeURL
76      global liberalURL
77      global cLinkName
78      global lLinkName
79      global errMessage
80      global tokenizer
81      global cNumber
82      global lNumber
83
84      # Need to clear these fields to run another query
85      conservativeURL = ' '
86      liberalURL    = ' '
87      cLinkName     = ' '
88      lLinkName     = ' '
89      errMessage    = ' '
90      cNumber       = 0
91      lNumber       = 0
92
93      form = SearchForm(request.form)
94      if request.method == 'POST' and form.validate():
95
96          inputString = request.form['inputString']
97          driver = webdriver.PhantomJS() # Creates an invisible browser.
98          driver.get('https://google.com/') # Navigates to Google.com.
99          searchBarInput = driver.find_element_by_name('q') # Assigns query to Google Search
                bar.
100
101         if inputString != '':
102
103             searchBarInput.send_keys(inputString + " news") # what you are searching for
104             searchBarInput.send_keys(Keys.RETURN) # Hit <RETURN> so Google begins searching
105             time.sleep(1) # sleep for a bit so the results webpage will be rendered
106
107             # Scrape liberal and conservative websites that are in the top 10 news websites
```

```
        # (top 10 according to http://blog.feedspot.com/usa_news_websites/ 's metrics).
        urls = driver.find_elements_by_css_selector('h3.r a')

        # Continue mining until conservative- and liberalURL are found
        while conservativeURL == ' ' or liberalURL == ' ':

            urls = driver.find_elements_by_css_selector('h3.r a')
            for url in urls:

                if conservativeURL != ' ' and liberalURL != ' ':
                    break

                # Need to remove end of links that make some webpages
                # impossible to create a usuable link for my webpage.
                stoppingPoint = url.get_attribute('href').index('&')
                url = url.get_attribute('href')[29 : stoppingPoint]
                # print(url)

                # The queried search page is a url and needs to be skipped.
                if '?q=' in url:
                    continue

                if "cnn.com" in url: # 1
                    linkName = "CNN Article"

                    lookAtPage = requests.get(url)
                    soup = BeautifulSoup(lookAtPage.text, "html.parser")
                    paragraphs = soup.find_all('div', {"class":"zn-body__paragraph"})
                    text = ''
                    for paragraph in paragraphs:
                        text = text + paragraph.text
                    #print(text)
                    checkURL(url, text, linkName)

                if "nytimes.com" in url: # 2
                    linkName = "NY Times Article"
                    lookAtPage = requests.get(url)
                    soup = BeautifulSoup(lookAtPage.text, "html.parser")
                    paragraphs = soup.find_all('p', {"class":"story-body-text
                        story-content"})
                    text = ''
                    for paragraph in paragraphs:
                        text = text + paragraph.text
                    checkURL(url, text, linkName)
                    #print(text)

                if "huffingtonpost.com" in url: # 3
                    linkName = "Huffington Post Article"
                    lookAtPage = requests.get(url)
                    soup = BeautifulSoup(lookAtPage.text, "html.parser")
```

```
            paragraphs = soup.find_all('p', {"class":"p1"})
            text = ''
            for paragraph in paragraphs:
                text = text + paragraph.text
            #print(text)
            checkURL(url, text, linkName)

        if "foxnews.com" in url: # 4
            linkName = "Fox News Article"
            lookAtPage = requests.get(url)
            soup = BeautifulSoup(lookAtPage.text, "html.parser")
            paragraphs = soup.find_all('p')
            text = ''
            for paragraph in paragraphs:
                text = text + paragraph.text
            text = text[161:-162]
            #print(text)
            checkURL(url, text, linkName)

        if "usatoday.com" in url: # 5
            linkName = "USA Today Article"
            lookAtPage = requests.get(url)
            soup = BeautifulSoup(lookAtPage.text, "html.parser")
            paragraphs = soup.find_all('p', {"class":"p-text"})
            text = ''
            for paragraph in paragraphs:
                text = text + paragraph.text
            #print(text)
            checkURL(url, text, linkName)

        if "reuters.com" in url: # 6
            linkName = "Reuters Article"
            lookAtPage = requests.get(url)
            soup = BeautifulSoup(lookAtPage.text, "html.parser")
            paragraphs = soup.find_all('p')
            text = ''
            for paragraph in paragraphs:
                text = text + paragraph.text
            text = text[:-36]
            #print(text)
            checkURL(url, text, linkName)

        if "politico.com" in url: # 7
            linkName = "Politico Article"
            lookAtPage = requests.get(url)
            soup = BeautifulSoup(lookAtPage.text, "html.parser")
            paragraphs = soup.find_all('p')
            text = ''
            for paragraph in paragraphs:
                text = text + paragraph.text
```

```
                    #print(text)
                    checkURL(url, text, linkName)

                if "yahoo.com/news" in url and "tagged" not in url: # 8
                    linkName = "Yahoo! News Article"
                    lookAtPage = requests.get(url)
                    soup = BeautifulSoup(lookAtPage.text, "html.parser")
                    paragraphs = soup.find_all('p', {"class":"canvas-atom"})
                    text = ''
                    for paragraph in paragraphs:
                        text = text + paragraph.text
                    #print(text)
                    checkURL(url, text, linkName)

                if "npr.org" in url: # 9
                    linkName = "NPR Article"
                    lookAtPage = requests.get(url)
                    soup = BeautifulSoup(lookAtPage.text, "html.parser")
                    paragraphs = soup.find_all('p')
                    text = ''
                    for paragraph in paragraphs:
                        text = text + paragraph.text
                    text = text[:-44]
                    #print(text)
                    checkURL(url, text, linkName)

                if "latimes.com" in url: # 10
                    linkName = "LA Times Article"
                    lookAtPage = requests.get(url)
                    soup = BeautifulSoup(lookAtPage.text, "html.parser")
                    paragraphs = soup.find_all('p')
                    text = ''
                    for paragraph in paragraphs:
                        text = text + paragraph.text
                    #print(text)
                    checkURL(url, text, linkName)

                if "washingtonpost.com" in url: # Requested by Hayden Le.
                    linkName = "Washington Post Article"
                    lookAtPage = requests.get(url)
                    soup = BeautifulSoup(lookAtPage.text, "html.parser")
                    paragraphs = soup.find_all('p')
                    text = ''
                    for paragraph in paragraphs:
                        text = text + paragraph.text
                    #print(text)
                    checkURL(url, text, linkName)

            if conservativeURL != ' ' and liberalURL != ' ':
                errMessage = "Search for \"" + inputString + "\" has completed
```

```
                    successfully!"
                break

            # Go to the next page, if possible, to continue the process.
            try:
                nextPage = driver.find_element_by_link_text("Next").click()

            except:
                errMessage = "Could not find enough sources on topic."
                break

        driver.save_screenshot('screen.png') # Save a screenshot to see operation.

    driver.quit()
    return redirect(url_for('index'))

#############################################################################

if __name__ == '__main__':
    app.run(debug=True, use_reloader=True)

#############################################################################
```

10.3 Appendix C

Website.html

```
<!doctype html>
<html>
    <head>
        <title>Fair & Balanced News</title>
        <meta charset=utf-8>
    </head>
    <body>
        <h1 style='text-align: center;'>Fair & Balanced News</h1>
        <div style="width:500px; margin: 0 auto">
            <h4 style='text-align: center;'>For what news do you want to search?</h4>
            <form method=post action="/results" style="text-align: center;">
                <input type="text" id="inputString" name="inputString"/>
                <input type=submit value='Search' name='search_btn'>
            </form>
            <h3 style='text-align: center;'>{{ errMessage }}</h3>
            <table align="center" border="1">
                <tr>
                    <th>Most Popular Liberal View</th>
                    <th>Most Popular Conservative View</th>
                </tr>
                <tr>
                    <td style='text-align: center;'>
```

```
                    <a href="{{ conservativeURL }}">{{ cLinkName }} {{cNumber}}</a>
                </td>
                <td style='text-align: center;'>
                    <a href="{{ liberalURL }}">{{ lLinkName }} {{lNumber}} </a>
                </td>
            </tr>
        </table>
    </div>
</body>
</html>
```

10.4 Appendix D

The following RNN hyperparameters were used when testing for the **batch size**:
HIDDEN_LAYER_SIZE = 300, TOP_WORDS = 5000, EMBEDDING_VECTOR = 128, DROPOUT = 0.2.

Batch Size	Evaluation's F1 Score	Evaluation's Accuracy (%)	Validation's F1 Score	Validation's Accuracy (%)
16	0.74	58.62	0.77	55.65
32	0.71	69.49	0.75	56.09
39	0.68	58.86	0.70	55.65
48	0.70	58.21	0.75	55.22
52	0.70	57.07	0.73	53.04
64	0.70	54.47	0.70	54.78

The **batch size** of 32 yielded the most accurate model.

The following RNN hyperparameters were used when testing for the **hidden layer size**:
BATCH_SIZE = 32, TOP_WORDS = 5000, EMBEDDING_VECTOR = 128, DROPOUT = 0.2.

Hidden Layer Size	Evaluation's F1 Score	Evaluation's Accuracy (%)	Validation's F1 Score	Validation's Accuracy (%)
250	0.72	57.32	0.77	54.78
275	0.73	57.48	0.77	53.48
295	0.72	57.48	0.75	56.09
300	0.71	60.49	0.75	56.09
305	0.72	56.91	0.75	53.04
315	0.69	54.31	0.69	50.43
325	0.71	58.70	0.76	53.48
350	0.86	55.53	0.90	53.04

The **hidden layer size** of 300 yielded the most accurate model.

The following RNN hyperparameters were used when testing for **top word**:
HIDDEN_LAYER_SIZE = 300, BATCH_SIZE = 32, EMBEDDING_VECTOR = 128, DROPOUT = 0.2.

Top Word	Evaluation's F1 Score	Evaluation's Accuracy (%)	Validation's F1 Score	Validation's Accuracy (%)
4750	0.70	56.26	0.73	53.91
5000	0.71	60.49	0.75	56.09
5250	0.71	59.35	0.75	56.96
5500	0.73	58.46	0.76	56.52

The **top word** of size 5000 yielded the most accurate model.

The following RNN hyperparameters were used when testing for the **embedding vector length**:
TOP_WORD = 5000, HIDDEN_LAYER_SIZE = 300, BATCH_SIZE = 32, DROPOUT = 0.2.

Embedding Vector Length	Evaluation's F1 Score	Evaluation's Accuracy (%)	Validation's F1 Score	Validation's Accuracy (%)
32	0.72	60.24	0.74	60.87
64	0.69	60.73	0.71	56.52
96	0.72	60.24	0.74	60.87
108	0.75	58.05	0.79	53.91
112	0.70	60.98	0.73	60.43
116	0.72	57.56	0.74	53.91
120	0.71	56.34	0.73	55.65

The **embedding vector length** of 112 yielded the most accurate model.

The following RNN hyperparameters were used when testing for the **dropout rate**:
TOP_WORD = 5000, HIDDEN_LAYER_SIZE = 300, BATCH_SIZE = 32, EMBEDDING_VECTOR = 112.

Dropout Rate	Evaluation's F1 Score	Evaluation's Accuracy (%)	Validation's F1 Score	Validation's Accuracy (%)
0.15	0.71	57.97	0.74	54.35
0.19	0.72	59.43	0.75	59.13
0.20	0.70	60.98	0.73	60.43
0.21	0.71	57.89	0.77	56.09
0.25	0.71	54.15	0.74	49.13

The **dropout rate** of 0.20 yielded the most accurate model.

Trying to add **another dropout layer** yielded:

Dropout Rate	Evaluation's F1 Score	Evaluation's Accuracy (%)	Validation's F1 Score	Validation's Accuracy (%)
0.20	0.7-0	59.11	0.74	58.70

Therefore, **another dropout layer** should not be included to yield the most accurate model.