Capstone Project
Machine Learning Engineer Nanodegree

Michael Boker
July 9th, 2017

Stock Market Predictions Based on News Headlines

Definition

Project Overview
During the early history of the stock market, stock traders made investment decisions based primarily on qualitative information. This information included reputations of entrepreneurs and employees at companies, products being produced by companies, and current events throughout the world. Since the 1950's, quantitative analysis has played an increasingly important role in these investment decisions[1]. Quantitative analysis involves mathematical models used to find predictive trends in the market, which are used to assist in making wise investment decisions. Computers are much more efficient than humans at processing numerical information, so they have been quite useful for quantitative analysis. Computers have been so dominant in this space, that many of the trader middle-men have been cut out, and trades have been automated. Some estimates say that 70 percent or more of trades are made automatically by computers, with no human involvement[2]. If we can use the qualitative information used by many human traders to augment the quantitative information currently used by the machines, we may be able to create more accurate prediction systems and thus more lucrative automated traders.

In this project, I created a recurrent neural network (RNN) which uses news headlines to predict behavior of the stock market. I used TensorFlow to implement the network. The network was trained on a dataset from kaggle[3]. This dataset pairs the top 25 news headlines for a day with a binary value indicating the behavior of the Dow Jones Industrial Average (DJIA) for that day. The headlines were gathered by scraping the World News subreddit, taking the 25 most upvoted headlines. If the binary value is 1, the value of the DJIA remained the same or rose; if the value is 0, the value of the DJIA fell. This dataset contains points for just under 8 years, spanning from August of 2008 to July of 2016.

Problem Statement
The goal is to create a classifier model, which predicts whether the DJIA will rise or fall on a day, based on the top news headlines for that day. The strategy is to create an RNN for sentiment analysis. Because I am doing a binary classification, I treat a label of 1 (the DJIA rose or stayed the same) as a positive sentiment, and a label of 0 (DJIA fell) as a negative sentiment. Each of the headlines for a day will be concatenated into a single string, so the problem will be similar to predicting the sentiment of a single message.

The expectation for this project was to determine whether a classifier RNN trained on news headlines is a viable and useful tool for a stock prediction system. Many factors play into the behavior of the stock market, and the dataset I had available is relatively small. Therefore, it did not seem likely that a

[1]McWhinney, James E. "A Simple Overview of Quantitative Analysis" *Investopedia*
[2]Salmon, Felix and Stokes, Jon "Algorithms Take Control of Wall Street" *WIRED*
[3]User Aaron7sun "Daily News for Stock Market Prediction" *Kaggle*

classifier with great accuracy would be a result, but I aimed to determine if efforts similar to this are worthwile.

Evaluation Metrics

For evaluating the performance of the model, a combination of prediction accuracy and F score were used. Accuracy indicates the ratio of correct to incorrect predictions, but can be misleading if the data is skewed. F score seemed to be a good metric for ensuring that the model truly learned from the data, and did not just learn a skew in the data. If the values in the training data skew towards one label or the other, and the model learned to be biased towards that label, it would receive a low overall F score. The equations for F score are below.

$$F = \frac{2 * precision * recall}{precision + recall}$$

$$precision = \frac{truepositives}{truepositives + falsepositives}$$

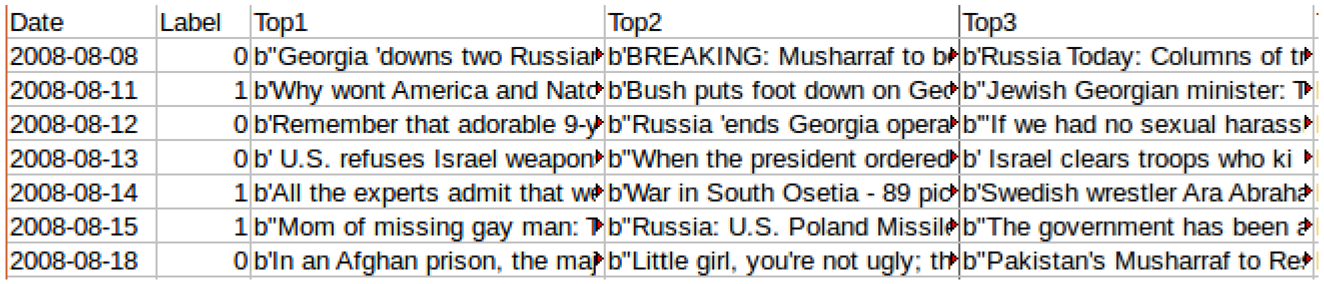
$$recall = \frac{truepositives}{truepositives + falsenegatives}$$

This score was calcuated for each of the two classes, and the average of the two values was used, giving equal weight to each value.

Preventing a learned bias is important for a task like stock market predictions. If the model learned to favor one prediction over another due to a skew in the data, this could be disastrous when the model is used on new data that may not be skewed the same way.

Analysis

Data Exploration and Visualization



Date	Label	Top1	Top2	Top3
2008-08-08	0	b"Georgia 'downs two Russiar	b'BREAKING: Musharraf to b	b'Russia Today: Columns of tr
2008-08-11	1	b'Why wont America and Natc	b'Bush puts foot down on Gec	b"Jewish Georgian minister: T
2008-08-12	0	b'Remember that adorable 9-y	b"Russia 'ends Georgia opera	b'"If we had no sexual harass
2008-08-13	0	b' U.S. refuses Israel weapon	b"When the president ordered	b' Israel clears troops who ki
2008-08-14	1	b'All the experts admit that we	b'War in South Osetia - 89 pic	b'Swedish wrestler Ara Abraha
2008-08-15	1	b"Mom of missing gay man: T	b"Russia: U.S. Poland Missile	b"The government has been a
2008-08-18	0	b'In an Afghan prison, the maj	b"Little girl, you're not ugly; th	b"Pakistan's Musharraf to Re

Table 1: Snippet of dataset

The dataset consists of 1889 points, each with 27 features. Each point corresponds to a day. The first feature, named “Date,” is the date for that day, and the second feature, named “Label,” is the binary value indicating the performance of the DJIA that day. The remaining 25 features are “Top1” through “Top25,” and they contain the top 25 headlines for the day, in descending order of popularity. Table 1 above contains a snippet of the spreadsheet containing this data, showing 7 points, and the first 5 features, with each of the three included headline features abbreviated to fit on the page.

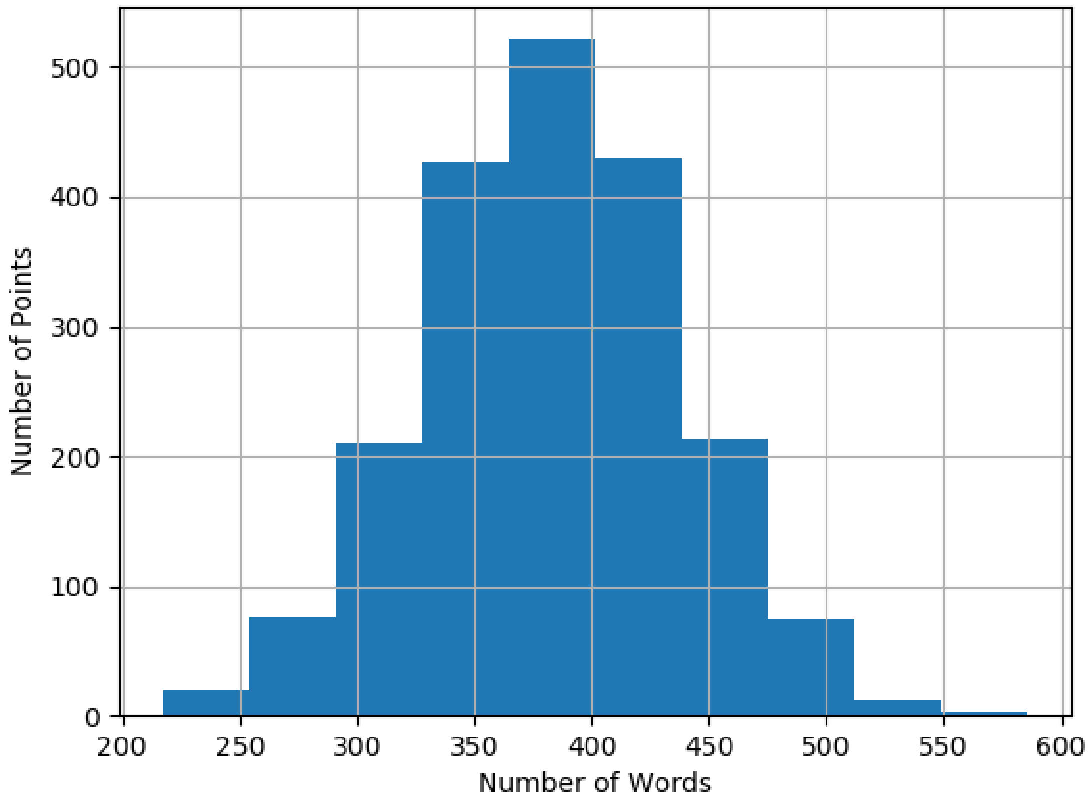


Figure 1: Histogram of word counts

We can see from the histogram in Figure 1 above, that the combined lengths of headlines are normally distributed. Knowing this, I determined the characteristics of this normal distribution and found that the mean was 382.6 words, the standard deviation was 54.6 words, the 25th percentile value was 347.0 words, and the 75th percentile was 418.0 words.

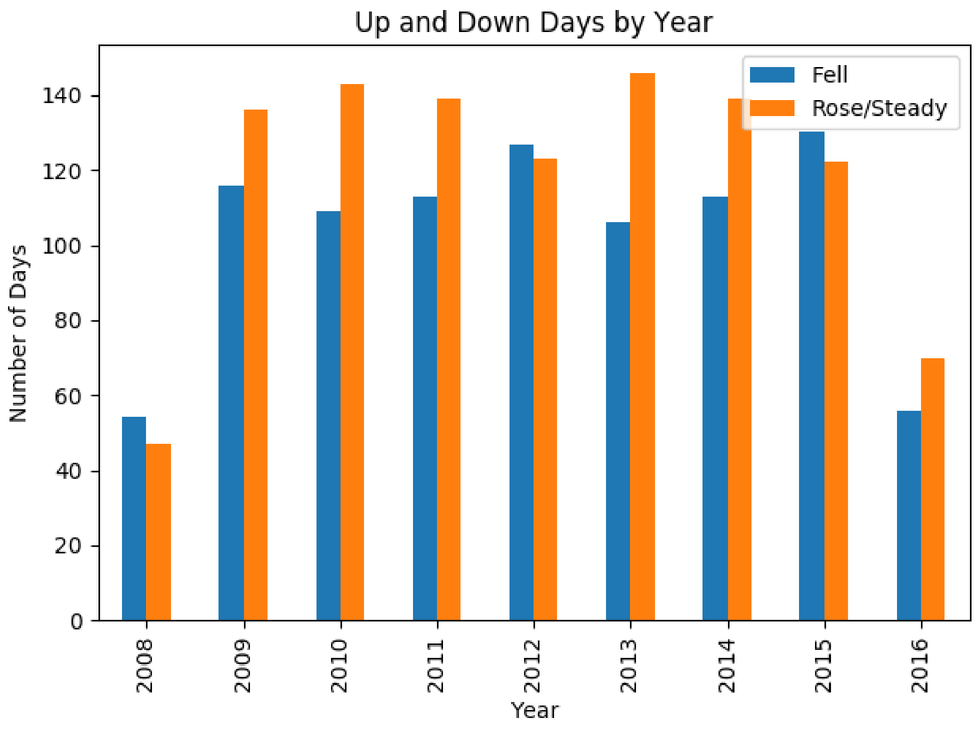


Figure 2: Days of DJIA rise compared to days of DJIA fall, by year

Out of all of the days in the dataset, 56.4% (1065 out of 1889) had labels of 1. This is pretty close to an even split between classes. A quick visual scan of the dataset indicates that the two labels are each evenly distributed throughout the data, without any large contiguous blocks of one label. The bar chart in Figure 2 above shows that the points of each class are indeed evenly distributed among each year in the data. The ratio of number of days for which the DJIA remained steady or rose compared to the number of days for which the DJIA fell is not the exact same for each year. However, there is not any skew large enough to present a major concern.

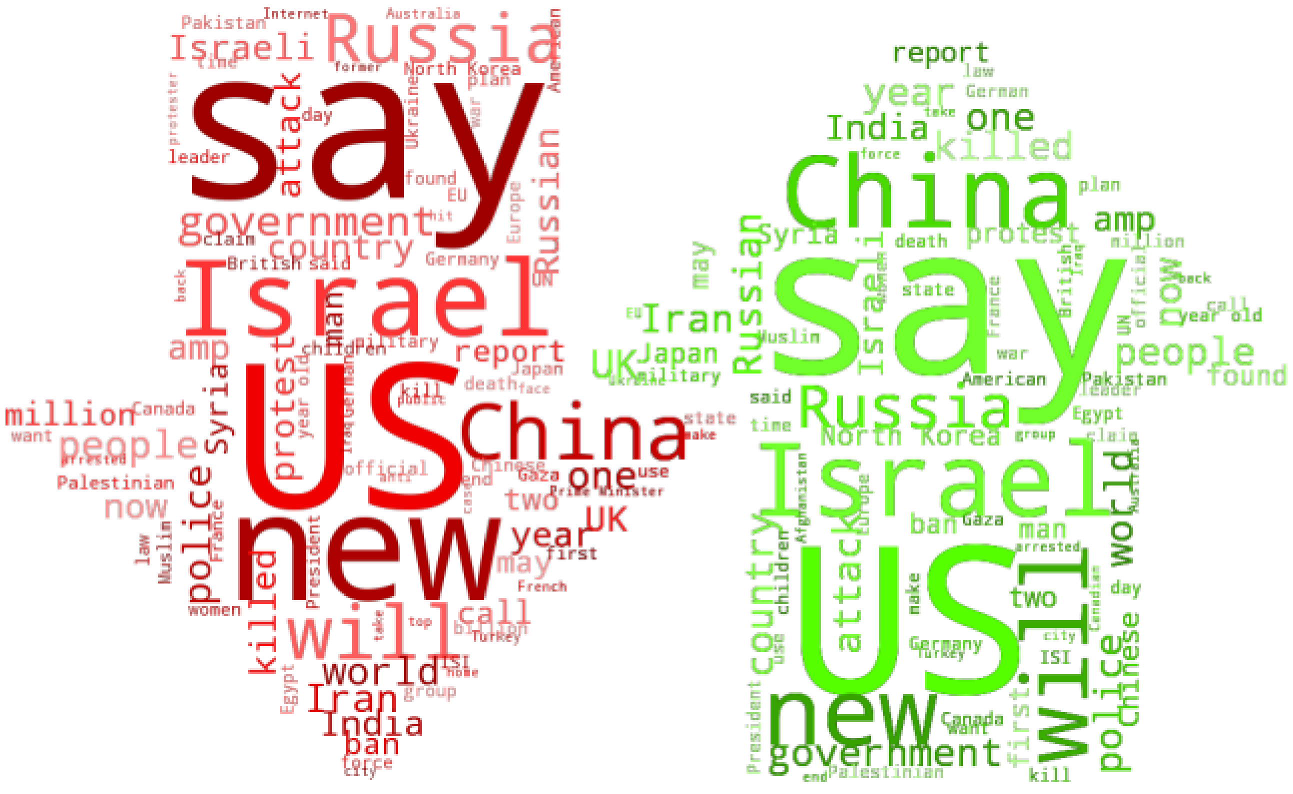


Figure 3: Word clouds for down days (Left) and up days (Right)

To determine whether there is any clear relationship between the DJIA behavior and the most common words used in headlines, the word clouds in Figure 3 above were created. Examining these clouds, we see many words are shared between them. Country names such as "US," "China, and "Israel" are in both clouds. Other words such as "people," "government," "say," and "country" are also found in both clouds. Most importantly, the most common words, and thus the words which are biggest in the clouds, are very similar between the two clouds. This is concerning, because it increases the challenge of predicting the behavior of the market based on these headlines. However, it makes sense, because these words occur very frequently in headlines, and many stories make headlines for multiple days. So, one story about China may have related headlines in the news for two consecutive weeks, during which there could be 6 days of the DJIA rising, and 4 days of the DJIA falling. The word "US," an abbreviation for the United States, is likely present in more than one of the top 25 news headlines on any given day, since the rank of these headlines is determined by upvotes from a primarily American group of users.

Algorithms and Techniques

In order to try to place more emphasis on words that may be more relevant to the classification for a day, a technique called subsampling was used. Subsampling probablistically ignores words with a probability proportional to the frequency of the words. So, the more frequently a word occurs in the dataset, the more likely it is to be ignored during training. This helps to mitigate dillution of our data by words which are seen very frequently for both classes. The probability of dropping a word during subsampling is calculated with the formula below, where t is a frequency threshold value that can be set as a hyperparameter, and f is the frequency of the word in the training corpus.[4] Words that have frequencies lower than the threshold value will never be ignored due to subsampling.

$$p = \frac{f-t}{f} - \sqrt{\frac{t}{f}}$$

As with subsampling, stop words were also removed from the text to prevent dillution of the data. Stop words include common words that add no significant meaning to a sentence, such as articles like "a" and "the." Stop words were removed from the text before the distribution characteristics discussed previously were calculated.

I needed to select a standard number of words which all of the combined headline strings would contain. Any headline strings shorter than this length were padded to equal the length, and any headline strings longer were truncated to equal the length. Based on the distribution of headline string lengths, a length of 420 words was used. This was long enough to contain all words for the majority of the data points. Also, it was not so large that performance was hindered unnecessarily by useless padding on the shorter strings.

The word clouds above indicate a reason that I used an RNN for this project. A bag of words classifier which makes predictions based on the occurrence counts of words would not have much of a chance for success, because the words used are so similar between classes. The RNN can learn the meaning or sentiment of a headline, which is impacted by the ordering of words. The network can make predictions based on what the headlines say, rather than simply what words they contain.

The RNN is a deep-learning approach to text analysis, which allows the model to persist information from one time step to the next. Strings of text are fed into the network one word at a time, and the network learns context and meaning from the words in the strings, as well as their ordering. A particular type of cell, called the Long Short-Term Memory (LSTM) cell was used for this implementation. The LSTM is designed to avoid problems that arise in simpler RNN implementations during back-propagation in training. The LSTM enables the network to remember information over longer sequences than a traditional RNN cell.

A common technique with deep neural networks is called dropout, and this was used in my implementation. By using dropout on a cell, that cell will probablistically drop its input at each time

[4]Mikolov, et al "Efficient Estimation of Word Representations in Vector Space" *Cornell University Library*

step. So, with a dropout keep probability of 0.6, each cell will have a 60% chance of passing the information on to the next layer and a 40% chance of dropping the information at each time step. Applying dropout to each of the LSTM cells in our network was an attempt to prevent any cell from overfitting to a specific feature. This helped the network avoid memorization of training data.

The dimensionality of the input space for this project is huge. This is because the textual data could include any word from the English language, as well as any names, including those of countries, companies, and people. Each possible word is an input dimension, and all of these possibilities would amount to an input space of somewhere from 100,000 to 200,000 dimensions. This is far too large to be practical for training a neural network, given the size of the dataset and the computing power available. For this reason, I used word embedding to reduce the input dimensionality. Word embedding is a popular technique, in which words are converted into vectors in a lower dimension, and words with similar meanings have similar vectors.

Data was divided into batches, in order to take advantage of efficiencies provided by parralel processing on the GPU. The batch size is the number of data points that were fed into the network in parallel at any time. A high batch size would allow for faster training. However, it could hamper the training performance, because the network would not benefit from using information learned from one data point during training with another data point in the same batch. The larger the batch, the more points suffer from this lack of shared information.

At first, I attempted to train the model over the entire dataset for multiple epochs, validating after each epoch, and testing after all epochs were complete. The number of training epochs needed to be high enough for the model to learn an effective fit, but not so high that the model became overfit and could not generalize.

In later trials, I used time series cross validation. To accomplish this, I performed “rolling” validation through the data as the model was trained. First, I trained the model on just the first training batch, and validated on the second training batch. Then, I trained on the first two training batches, and validated on the third training batch. I continued this process through all of the training data, so that in the final time step, the model was trained on all but the last training batch, and it was validated on the last training batch. For time series cross validation to be effective, batches cannot be seen by the model before they are used in validation. So, in order to allow the model to learn something from each new training batch, without having to reiterate over batches, I looped over each step in the training process. By this, I mean that I trained the network on the first training batch multiple times, before validating the first time step, and then trained on the first two batches multiple times before validating, and so on.

Based on performance of the network, hyperparameters were adjusted for improvement. Most of these hyperparameters have been discussed above and include the following:

- The number of dimensions in the output of the word embedding
- The number of LSTM cells in the recurrent hidden layer
- Dropout keep probability
- Length of the headline strings
- Subsampling threshold
- Learning rate
- Batch size
- Number of training epochs
- Number of repetitions per time step in cross validation

Benchmark
For a benchmark model, a cue was taken from work produced by a Stanford student, Bryce Taylor, who attempted to create a model similar to what I produced for this project. For the benchmark in his work, Taylor compared the performance of two algorithms - one which always predicted an increase in stock value, and one which always predicted a decrease. He used the better performing of these two for his baseline.[5] This model can never have error above 50%, and in the relatively immature field of using natural language processing to predict stock behavior, this seems like a sufficient benchmark. This guarantee of at least 50% accuracy comes at the cost of F score, because the model blindly selects the same class for every point. My goal was to create a model which surpassed this benchmark in both accuracy and F score.

Methodology

Data Preprocessing
Some preprocessing steps were taken to prepare the data for training the network. In the Jupyter notebook accompanying this document, this occurs in the section titled "Data Preprocessing". As seen in the snippet of the data shown earlier, many of the headlines began with the letter "b," and were enclosed in quotation marks – double (") or single ('). So, each headline was stripped of b's and quotation marks. Next, all 25 headlines were concatenated into one string for each data point . The delimiter string, "||new_headline||" was inserted into the strings to start each headline. These combined strings were used to create a new .csv file, "headlines_combined.csv" which contained 3 columns, "Date", "Label", and "Combined".

Following this, the combined headline strings were converted into lists of words and punctuation marks with the nltk word_tokenize function. Each of these lists had stop words removed, remaining words shifted to all lower-case letters, and punctuation symbols tokenized into strings, such as "||comma||".

A set was then created of all the words and symbol tokens in the dataset. This set was enumerated and used to create a dictionary mapping words and symbol tokens to integer IDs. The lists of words were then used to create new headline strings, where each word and symbol was replaced by its corresponding numeric value from the dictionary. This was necessary because I had to feed numeric values into my network. These strings were used to create a new .csv file, called "tokenized_headlines.csv" which was saved in the data folder, as a checkpoint.

Implementation
In builders.py, I implemented some helper functions to be used in building the input to the network. Some of these are used in the "Get Data and Create Batches" section of the accompanying notebook. Following is an outline of how these functions are called and a brief description of the purpose of each.

1. **get_data_and_vocab**(file_path) : Takes the file path for the csv file with tokenized headline strings that was saved at the end of preprocessing. It returns a dataframe containing that file as well as a set of every word id that occurs in the document.

[5]Taylor, Bryce "Applying Machine Learning to Stock Market Trading" *Stanford.edu*

2. **get_data_splits**(split_frac, data) : Takes a float value for split_frac which indicates what portion of the data is to be used for training, and what portion is for validation and testing. The portion for validation and testing is then split in half. All of the data points are kept in chronological order, with training data first, followed by validation, followed by testing.
3. **create_lists_and_filter**(train_x, val_x, test_x, subsample_thresh) : Creates a counter of the words in the training data. For each portion of the data, it then splits every headline string into a list of integers, filtering out id's for words that occur less than 5 times in the training set. While creating the list, it also subsamples the data, using subsample_thresh to calculate the probability of discarding each word.
4. **get_batches_and_pad**(x, y, batch_size, length) : Called with train_x and train_y, followed by val_x and val_y. It makes every headline string in x have length equal to the length parameter. It does this by padding short strings, and truncating long strings. It then splits the x and y lists into batch_size subsets and returns a list of dictionaries containing headlines and labels, each dictionary representing a batch.

The rest of the functions in builders.py are used for building the neural network itself. Following is an outline of the use of these functions.

1. **build_inputs_and_targets**() : Builds placeholders for the network inputs and targets
2. **build_embedding_layer**(inputs, vocab_size, embedding_dim) : Builds an embedding layer to reduce input dimensionality from vocab_size to embedding_dim.
3. **build_lstm_cell**(rnn_size, batch_size, output_keep_prob) : Builds a layer of rnn_size LSTM cells, with dropout wrappers.
4. **build_rnn**(cell, inputs) : Builds the RNN out of the LSTM layers and input plaeholder.

Next, a fully connected layer with a sigmoid activation is built at the end of the LSTM layers. The mean squared error is used as the cost function, which is used in an AdamOptimizer to train the network.

Refinement

The initial hyperparameter settings were as follows.

- batch_size = 198
- length = 420
- subsample_thresh = 10^{-5}
- num_epochs = 100
- rnn_size = 64
- embed_dim = 150
- learning_rate = 0.01
- dropout_keep_rate = 0.7

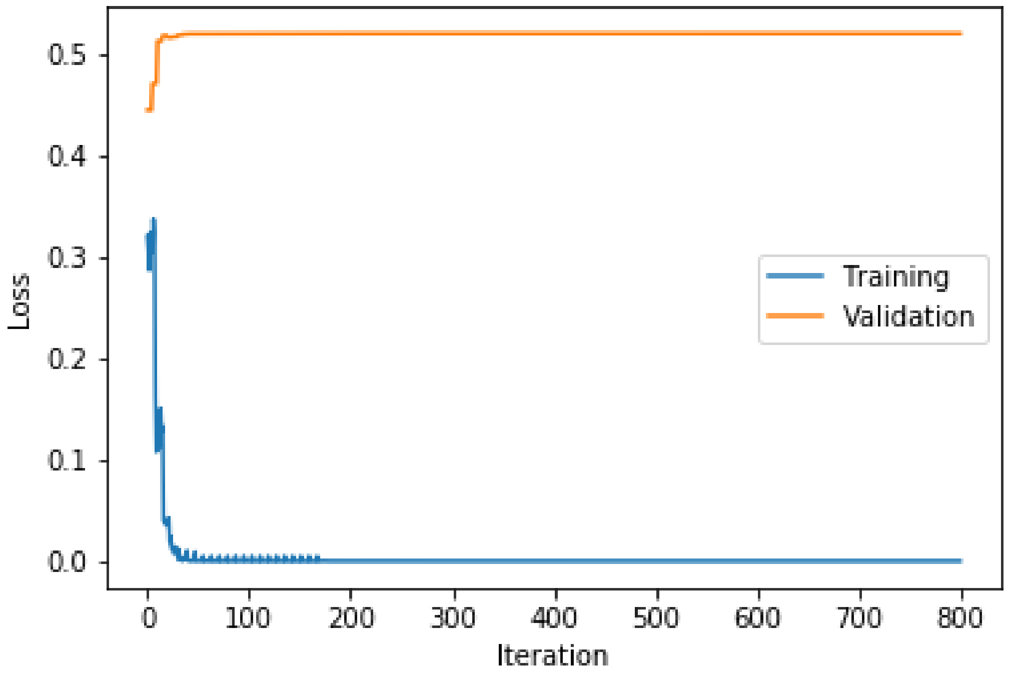


Figure 4: Learning curve of first network configuration

As can be seen in the learning curves in Figure 4 above, overfitting quickly became a problem with this configuration. Training loss fell to 0.000 by epoch 4, and validation loss increased very rapidly to a limit of about 0.52, where it leveled off.

To try to improve the problem of overfitting, some of the hyperparameters were adjusted. The size of the hidden layer, rnn_size, was reduced by half, to 32. The learning rate parameter was decreased by a factor of 10, and dropout_keep_rate was reduced to 0.5. The results of this were pretty similar to the first configuration. The training loss decreased more slowly and validation loss increased more slowly, but the values were the same after about 100 iterations.

This led me to think that the problem was likely due to an insufficient amount of training data. I needed to retain reasonably sized validation and test sets, so I could not increase the portion of data points used for training. The size of the training set would likely need to be increased by orders of magnitude, anyway, so taking points from the validation and test sets would have been pointless. Working with the options I had available, I increased the subsampling threshold to 10^{-1}, thus effectively turning off subsampling. This would result in more of the words from the dataset being used for training. This had no noticeable effect on results.

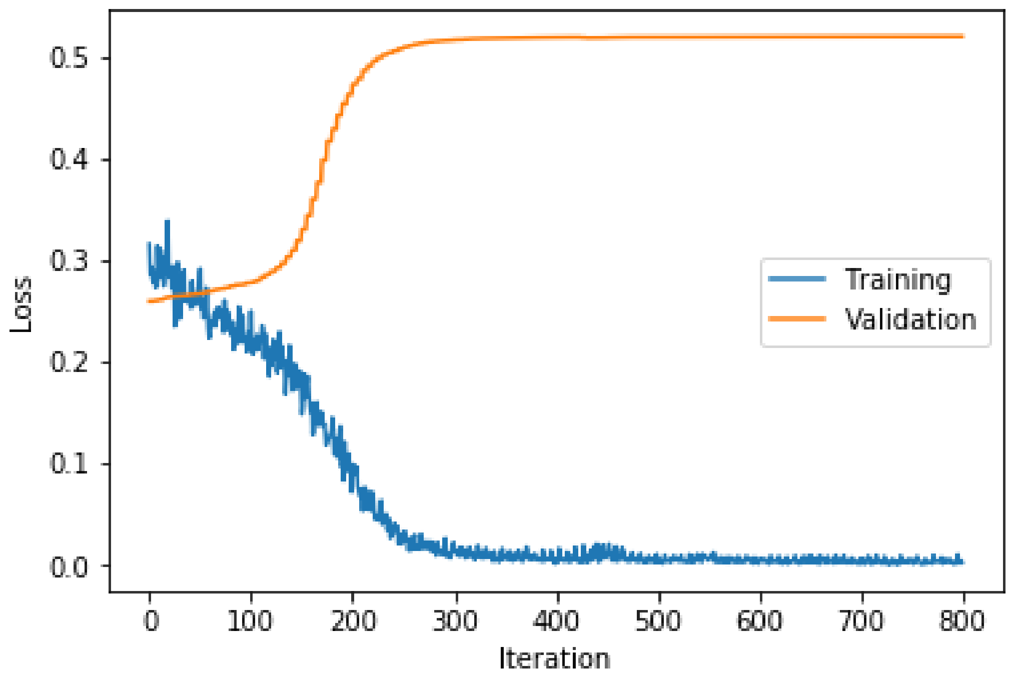


Figure 5: Learning curves for final configuration without cross validation

The best result I was able to attain with my initial methodology was a slower divergence of training and validation losses, as seen in Figure 5 above. This was achieved with the following set of hyperparameters.

- batch_size = 198
- length = 420
- subsample_thresh = 10^{-1}
- num_epochs = 100
- rnn_size = 16
- embed_dim = 50
- learning_rate = 0.001
- dropout_keep_rate = 0.3

The model suffered from overfitting with any combination of hyperparameters. After reducing the size of the network, I tried making the network larger, for sake of experimentation. I used rnn_size values of 64, 128, and 256. With each of these, I tried embed_dim values of 100 and 150. I also tried dropout_keep_rate values of 0.3 and 0.7, and learning rates of 0.01 and 0.001, for each of the combinations of rnn_size and embed_dim. As might be expected, the larger the network and embedding dimensionality, the quicker the model was overfit. The training and validation losses each converged on the same respective values for any combination of hyperparameters, they simply did so at different rates.

In an attempt to mitigate overfitting and test the robustness of the model as it was learning, I modified my code to use time series cross validation. I added another hyperparameter, ts_cv_step_reps, which sets how many times each training step is repeated before the validation for that step is performed. The hyperparameter num_epochs was no longer used, as this new technique replaced the need to train over the entire training set for multiple epochs.

The effectiveness of this change was not clear from the results. For the first trial of this technique, I used the following hyperparameter values, and observed the learning curves below.

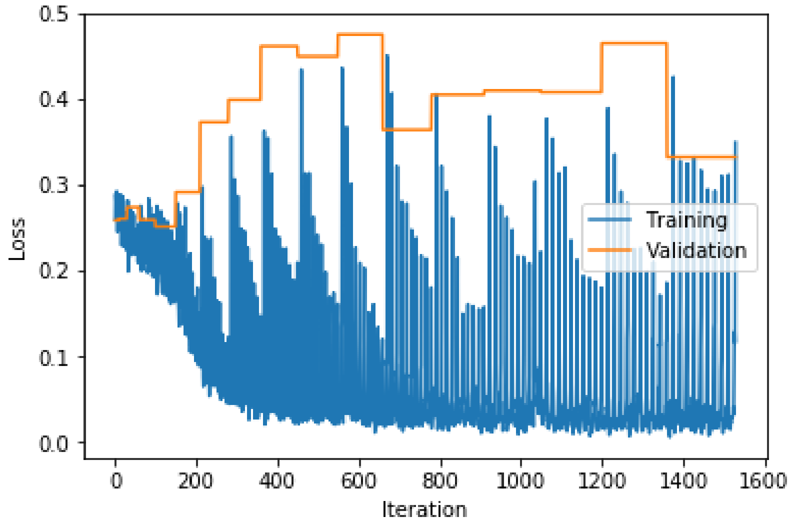


Figure 6: Learning curves for time series cross validation trial 1

- ts_cv_step_reps = 10
- batch_size = 99
- length = 420
- subsample_thresh = 10^{-1}
- num_epochs = 100
- rnn_size = 8
- embed_dim = 50
- learning_rate = 0.001
- dropout_keep_rate = 0.3

I found it promising that the validation loss did not immediately explode, and it even seemed to decrease initially. However, I was troubled by the lack of convergence of the validation loss, and its overall upward trend. It appeared that the validation loss began to increase as soon as the training loss decreased significantly – a telltale sign of overfitting. I tried tweaking the hyperparameters, particularly setting ts_cv_step_reps to lower values, to decrease the chance of overfitting. No trial performed significantly better than the first trial.

Results

Model Evaluation and Validation

The final parameters of the model were selected in an effort to get a useful model with a prohibitively small dataset. The subsampling threshold was raised to a level that would likely result in no words being discarded through subsampling. The value of 10^{-1} means that a word has to have a frequency of 1 occurrence for every 10 words, before it is even subjected to subsampling. This was done so that almost all of the words available in the dataset were used.

For the trials in which time series cross validation was not used, the number of epochs was left at 100 for the final model, because lowering it would serve no purpose. The number of epochs should be capped at the point when improvement of the model stops, and overfitting starts to become a problem. Since the model never improved, this would mean training the model for 0 epochs. I could have stopped training at any arbitrary time before validation loss reached its maximum value. However, any positive results from this would have been lucky guessing on the part of the model.

For the trials in which time series cross validation was used, ts_cv_step_reps was set to 10 for the final model. This value was used because higher values resulted in quicker overfitting by the model, and lower values resulted in similar performance by the model at the beginning of training, but they did not show as much improvement in the middle and end of training.

The RNN size was set at a very small value in an effort to avoid overfitting. This also limited the amount that the network could learn, but with a small dataset, the focus was on getting the network to learn *something*. With more data, a larger recurrent layer may have been useful, as well as more recurrent layers.

Embedding dimension was kept to the range of 50 to 150 dimensions, because research has shown that networks only have marginal improvement for embedding sizes above 50, and almost no improvement above 200 dimensions6. Due to concerns with the amount of data, using smaller input dimensionality seemed like the best option, and it did result in the slowest divergence of training and learning losses.

A learing rate of 0.001 was settled on because 0.01 was found to result in extremely fast overfitting. The value of 0.001 did not have much impact on results compared to 0.01, so no smaller values were used. Also, because an AdamOptimizer was used, this learning rate decays throughout training, so using values smaller than 0.001 would not be effective.

Dropout keep probability was tweaked and experimented with at different values. Similar to the other hyperparameters, concerns with overfitting dictated the decision with this value. A value of 0.3 is low for dropout keep probability, but this value resulted In the most promising performance of the model, where overfitting occurred the slowest.

[6]Lai, Siwei, et al. "How to Generate a Good Word Embedding" *arXiv*

Justification

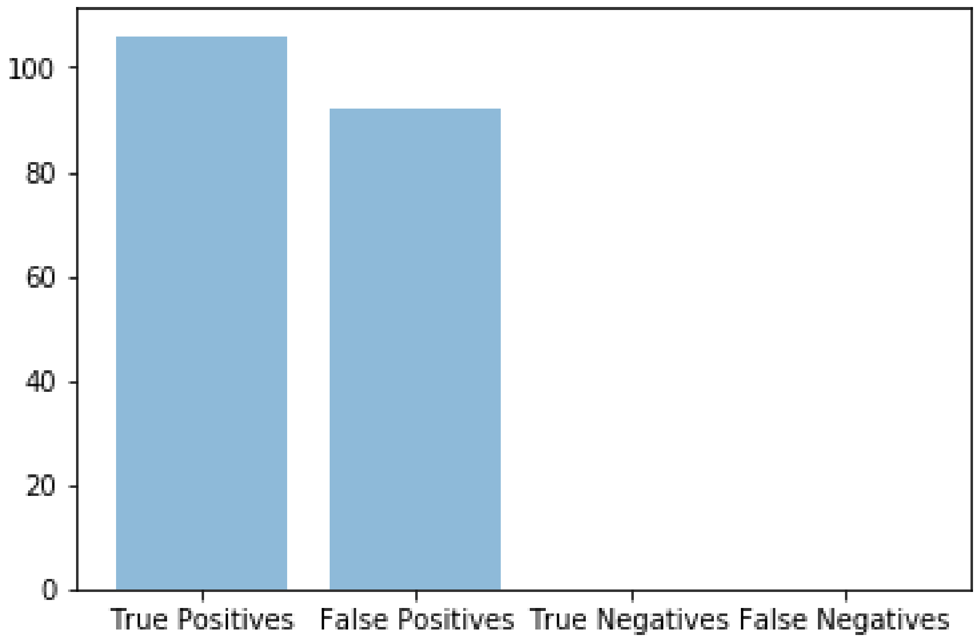
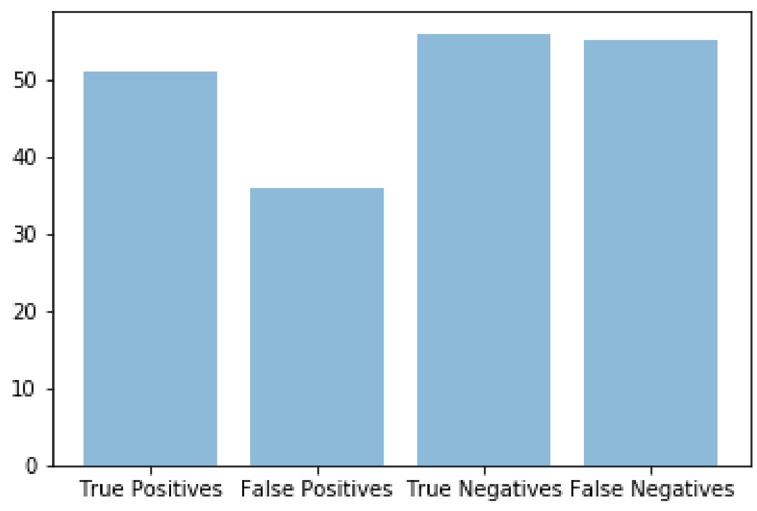


Figure 7: Predictions of final model trained without cross validation

For the trials where time series cross validation was not used, the best performing model seen throughout the refinement process performed exactly the same as the benchmark model, with an accuracy of 0.535 and an F score of 0.349. I assumed that this meant the model learned to blindly guess the same label for each point. Upon inspection of the test predictions, I found this to be the case. As can be seen in the graph in Figure 6 above, the model made a positive prediction for every test point. This shows that the model did not truly learn anything useful from the data, and it is not a solution to the problem.



For the trials in which time series cross validation was used, the performance varied from trial to trial, but all of the trials exceeded the benchmark in F score, for every validation step as well as for the testing set. These F scores ranged from 0.42 to 0.60. The accuracy varied from 0.42 to 0.61 for these trials. The best performance on the test set resulted in an F score of 0.54 and an accuracy of 0.54. This shows that the model may have learned something from the data, because at least it was not blindly favoring one class over the other. However, the variation in predictions between trials indicates again that more data is necessary for effectively training and testing the model. These results showed promising potential, but I do not have enough confidence in the model to feel comfortable using it with new, real-time data.

Conclusion

Visualization

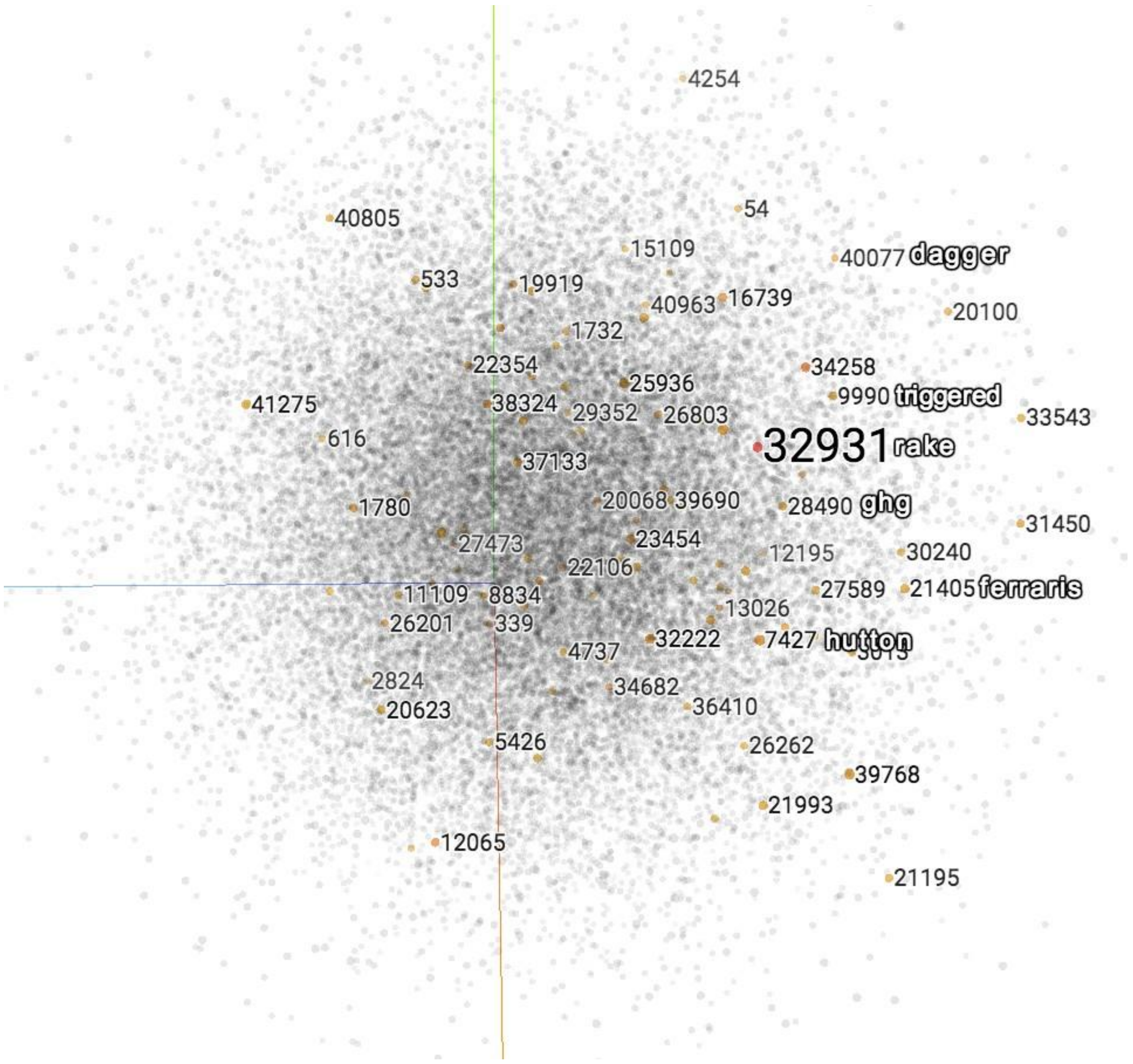


Figure 8: Visualization of word embedding vectors

Using TensorBoard, I was able to create a visualization of the word embedding from this project. This visualization is created using principal component analysis to find the 3 components that represent the most variance in the embedding space. All of the words in the vocabulary are then graphed in this 3 dimensional space. In TensorBoard, you can rotate the graph, and you can click words to select them. Selecting a word will highlight other words that are similar. In the image in Figure 7 above, a screenshot of this embedding visualization is shown, with the word 32931 selected. Because the words were converted to integer IDs, these IDs are shown in the visualization. I have superimposed the words corresponding to some of the IDs. This visualization shows that the word embedding would have likely benefited from more training data. When selecting the word "rake," we would expect to see highlighted words like "comb," "glove," "hoe," or "shovel." The highlighted words should be words with similar meanings to the selected word. Nonetheless, it is an interesting image, and it illustrates another result of the insufficient training data.

Reflection

I learned a lot about RNNs through this project. I found that the data preprocessing steps that are required for RNNs are very similar to those used for other natural language processing techniques. The techniques of tokenization and stop word removal are tools that I have used with more traditional machine learning algorithms. Word embedding, however, is a technique that was new to me, and I find it fascinating. The ability to reduce the dimensionality of the input space by 3 or 4 orders of magnitude, without much loss of information, is very powerful. Despite the shortcoming of word embedding in this project, it has been shown to be a very powerful technique, and I am excited to try it in a case with more training data.

There is a relatively high amount of overhead involved in building a neural network in TensorFlow, at least compared to algorithms from other libraries, such as SciKit-Learn. With SciKit-Learn, you simply set the hyperparameters that you want to use. If a particular algorithm does not prove to be effective, you can quickly and easily try a different one. Implementing the network in TensorFlow gives you the ability to control things such as what types of nodes are at each layer, and what initial values the weight parameters have. This would be very useful if you had a unique network architecture that you wanted to build. However, for the purposes of a simple recurrent neural network, most of the customization could be controlled by hyperparameters. Hyperparameters could be used for the number of layers, types of cells in each layer, initial weights, and any of the features that I did use hyperparameters for in my implementation. In the future, for uses similar to this, I will likely use a higher level library that is built on TensorFlow. I am familiar with two such libraries: Keras and TFLearn, and I am sure that others exist or will be created.

Another concern with RNNs is the time and resource requirements for training. I used an Amazon Web Services machine with a GPU, and training still took at least about 5 minutes, and much longer for networks with more cells. This, combined with the building the network consumed a lot of my time.

Improvement

I have some ideas for how this project could be improved upon. First, more data is necessary. Even with the input space being reduced using word embedding, the dataset was far too small. Getting enough data for this purpose may be difficult, because the stock market is only open for about 250 days a year, and the data only exists for a limited number of years. One approach may be to get more granular on the time scale. Perhaps, grouping headlines and market behavior into 1 or 2 hour windows would allow for more useful data to be gathered. Another approach for gathering more data would be to get more granular in the market scale. By this, I mean to focus on news articles pertaining to specific companies or sectors, and use the stock performances of those companies or the performance of indexes for the respective sectors. Combining this approach with the smaller time windows would allow for significantly more data to be collected.

The approach of narrowing the market focus for gathering more data leads me to another idea for improving this project, which is to focus its domain. The top headlines for world news is a broad category, and the Dow Jones Industrial Average is a very broad stock index. This is by design, as it is intended to provide information about the performance of the stock market as a whole. Rather than taking such a broad approach, focusing on specific companies or sectors might yield more useful results.

One more idea for this project that could be experimented with is adding some latency to the predictions. By this, I mean that the network could be used to predict market behavior tomorrow, based on news today. Another option is that a time window could be used. The network could predict market behavior for a day based on the news from the previous 5 days, for example.

Finally, the idea that stock predictions can be based solely on news articles may be a naiive one. The goal of this project was to show that a tool like this RNN could be useful for augmenting the quantitative analysis currently done by computers. This qualitative information may not provide any useful foundation for predictions on its own, but combined with numeric data, perhaps it would be found to be very powerful.