Vijay Yedidi

Project: Intro to Machine Learning Applications

Due 11/13/2020

Professor Kuruzovich

**Executive Summary:**

Over the years Twitter has become a cornerstone of our lives. Over 330 million people use Twitter worldwide, and as we can see in this past US election cycle, the information posted on Twitter can be used to drastically sway public opinion. It’s sometimes hard to understand exactly how these tweets can impact public opinion. When these tweets go viral, they can drastically improve a person or company’s brand, or they can devastate their brands if the tweets are more negative in nature. In order to predict the impact viral tweets can have on the bottom line of a company, we need to thoroughly understand the connotation behind the tweets being put out so if/when they go viral, we can create models that automatically predict whether they positively or negatively impact the bottom line for the brands involved.

In order to do this, we need to understand the words or phrases Tweets contain that most significantly impact their sentiment, either positive, negative, or neutral, so we know what to look for when trying to analyze a Tweet and its impact on what it’s referring to. A machine learning algorithm can be employed to look at Tweets at a large scale to find the subtext that captures the connotation of the sentiment.

This problem was made into a competition by Kaggle, with Tweets and their corresponding “selected text”, or the phrases manually extracted by Figure Eight Data For Everyone [1], that most accurately signify the sentiment of the Tweet, which are to be used to train the model. The dataset also contains the sentiment of the Tweet, either positive, negative, or neutral. It is our job to train the model given the tweet and the sentiment to predict the subtext from the tweet that most accurately represents the sentiment. For this competition, neutral Tweets aren’t really important, since they don’t impact stakeholders, so in the final results, the selected text for the Tweets with a neutral sentiment was just the whole Tweet, with no subsets taken.

Overall, 3 different methods were looked at in order to complete this model. The first solution involved no traditional machine learning methods; it was a mathematical algorithm created where every word in the dataset was assigned a weight based on how often it appeared in its sentiment, and the selected text is created based on the substring that has the highest weight. This was a very clever model, with a score of 0.*5841617****.***

The second model was made using the Spacy NER model. This model performed very well, but there was no room for optimizing it. It outputted a score of 0.61835. This was decent, but there was plenty of room for error because this model couldn’t distinguish typos, and it could be thrown off by simple grammatical errors due to the nature of the model.

The final model was a BERT model. This model is similar to the SPACY NER, but with many more parameters, which results in a more accurate result. There were more options for hyperparameter tuning, which allowed some competitors to get some of the highest scores. Unfortunately, my computer was not able to handle the size of the BERT training model. I had to use a less accurate, but much more efficient version of the BERT model, called a distilBERT. This reduced my run time by 40% but decreased my model’s accuracy significantly.

**Benchmarking of other solutions:**

|  |  |  |  |
| --- | --- | --- | --- |
| Notebook Name | Feature Approach | Model Approach | Train/Test Perf |
| A Simple Solution Using Only Word Counts | Split by sentiment, positive, negative, and neutral. Then assigned weights to words in the “text” feature. | Mathematical Algorithm to assign weights to the words, outputted a substring based on the total weights of the words. | 0.64664 |
| Twitter Sentiment Extraction- Analysis, EDA and Model | Split by sentiment, positive, negative, neutral. | Used Spacy NER model, a convolution neural network to recognize entities given strings. | 61835 |
| Tweet Sentiment ROBERTA PyTorch | Inputted in the ROBERTA model as a question/answer. The sentiment was the question, the text was the context, and the selected text was the answer. | Stratified k fold with the ROBERTA training model, which is a convolution neural network with 82 million parameters. | 0.71439 |

***Model 1:***

The first solution I looked at was a relatively simple solution [2]. The author, Nick Koprowicz, used the algorithm seen in figure 1 to assign a weight to each word in every tweet. He started by splitting the Tweets into three different data frames based on sentiment (positive, negative, neutal). He then used the CountVectorizer function from Scikit learns “feature\_extraction” text package to find the total words counts for the tweets in each individual sentiment data frame, while removing “stop-words” like “an”, “the”, or “a”, which are only present for grammatical purposes, and would have no impact on the final sentiment.

He then proceeded to assign each word with a weight by calculating the portion how many Tweets the word was found in, and subtracting it with the portion of Tweets the word was not found in. So if a Tweet appeared in less than 50% of Tweets, its final weight would be a negative number. The Tweets were then split into random subsets of random lengths, and the sum of the weights for each subset were compared to each other. The subset of the Tweet that contained the highest total score was selected as the phrase most impactful to the sentiment of the Tweet and was put forth as the final solution (figure 2).

Text, letter

Description automatically generated

Figure 1: The algorithm used to find the weight for each word in each sentiment.

Figure 1: The algorithm used to find the weight for each word in each sentiment.

Graphical user interface, text, application, email

Description automatically generated

Figure 2: The algorithm used to find the phrase most impactful towards the sentiment of the Tweet given the weights of each word.

This solution was interesting since it didn’t involve any traditional machine learning methods. The model was not trained using the training data, This results in a solution of 0.651. This is moderately accurate (#1801 on the competition leaderboard). A huge part of why this model isn’t as accurate as it could be might be because of the fact that it wasn’t trained using the training data. The training data was created by humans, and may include certain biases or perspectives based on who interpreted the Tweets and how they comprehended them. The phrases extracted were not created using any formulas, so it’s hard to create one and expect it to perfectly line up. In this case, a traditional model might be the best way to go, since the selected text from the Tweets were created from the perspective of people, and this needs to be taken into account when creating the final model.

**Model 2:**

This model was created by Mr\_KnowNothing; he used The Spacy NER (Named Entity Recognition) package to perform his model. The NER model is a convolutional neural network, which classifies “entities” from text based on how its trained. So the author labelled the “selected text” as the entity to be found, and the Tweets as the text. He then trained the model, iterating through it 3 times with the order randomized to prevent overfitting based on the order of the Tweets inputted into the model. This produced a final score of 0.664, which is relatively high, but not as much as some of the other notebooks using other methods. The author reasons that the data was terribly unclean; there were many typos found which could impact the final training model. Other models that did better tended to incorporate ways to work around these typos either by cleaning the data, which this author argued removes some of the underlying meaning behind the data, or by using Transformers in their models, which are neural networks that look at the words to the left and right of the selected text, in order to truly understand the context of the subtext regardless of typos.

**Model 3:**

This solution was created by sazuma, using a PyTorch Robustly Optimized Bidirectional Encoder Representations from Transformers, or a RoBERTa[4]. It was developed in 2018 by Facebook, and has since been open sourced [5]. This is neural network that looks at words to the right and left of the selected text in the text, in order to really understand the subtext. This is done to understand the subtext in spite of typos or grammatical errors that can throw off a traditional NER model. The model was trained in a stratified K-Folds method, with 10 splits, to ensure all the data is used, and to eliminate bias. The model was optimized using AdamW, which optimizes the learning rate with each iteration, slowing it down as the loss function decreases over time. Because of this, the author could set a relatively high learning rate for the BERT model (3e-5) without the fear of the learning rate “skipping over” the true solution[6]. This model was then trained, with a final score of 0.71439. Consider the highest score is 0.7361, this solution is pretty accurate. Unfortunately, since Tweets are filled with mistakes, either because of typos or intentionally mis-spelling words to emphasize them or to change the meaning of the Tweet, so increasing the accuracy from this point would be extremely difficult, and may need to involve a much larger dataset.

**Data Description and Initial Processing:**

In order to properly understand our dataset, and to see what types of words/phrases we are dealing with, we need to start by pre-processing the data. To begin with, I used the Regex package (Re) in order to initially clean up the Tweet text. I used the package to remove punctuation/special characters such as “-, /, :”, as well as unnecessary spaces. I realized that the “\*” character was used in may Tweets to substitute for cursing, so I decided to leave them in as they could have an impact on the sentiment. I then removed stop-words using the nltk package. Words like “the”, “and”, or “as are just filler words for the sake of grammar, but for the purposes of this model, these words don’t directly affect the sentiment. When looking for null values, it was found there is one null value in the training data, where there was no Tweet associated with a neutral sentiment. All the authors I came across in this competition decided it was appropriate to drop this row, so I did so as well.

The data was then split by sentiment into three datasets; one for positive tweets, one for negative tweets, and one for neutral tweets. Figure 3 shows the number of Tweets for each sentiment. As can be seen in both the testing and training datasets, the vast majority of Tweets are neutral. Positive and negative Tweets seem to be evenly distributed, with positive Tweets marginally outnumbering the negative ones.

Chart, bar chart

Description automatically generated

Figure 3: A bar graph to break down how many Tweets are found by sentiment. The graph on the left is from the training data, the graph on the right is from the testing data

I next created a plot with a breakdown of the word distribution by sentiment. Figures 3, 4, 5, and 6 show the total word distribution, the distribution for positive Tweets, the negative Tweets, and the neutral Tweets respectively. The results for the positive and negative sentiments weren’t too surprising; words like “good”, “happy”, “love”, “thanks”, etc. were much more likely to show up in positive tweets, while words like “bad”, “miss”, “sad”, “sorry”, and “hate” were more likely to show up in negative sentiments.

Chart, bar chart, funnel chart

Description automatically generated

Figure 4: Word distribution for all Tweets

Chart, bar chart

Description automatically generated

Figure 5: Word Distribution for Positive Tweets

Chart, bar chart, funnel chart

Description automatically generated

Figure 6: Word Distribution for Negative Tweets

In the total sentiment graph (figure 3) it can be seen that positive words like “good” and “happy” showed up much more often than negative words like “sad” or sorry”. This can further be seen in figure 7, where I performed a distribution of word frequencies by sentiment. It is found that positive Tweets have many more unique words overall than negative tweets do. Neutral words have many more unique words than both the positive and negative Tweets, but for the purposes of this model we are only really focusing on positive and negative tweets; in the competition any Tweet that has a neutral sentiment has a selected text that is equal to the full text, so no data modelling is involved for processing neutral tweets.

Chart, bar chart

Description automatically generated

Figure 7: Total Word Count By Sentiment

Finally, I wanted to look at the character count distribution by sentiment. It can be seen that there is no discernable difference between Tweets in their character distribution. All sentiments are skewed to the right, due to the fact that the platform encourages shorter Tweets, so they are more likely to show up than longer ones.

Chart, bar chart, histogram, funnel chart

Description automatically generated

Figure 8: Character Count Distribution By Sentiment

**Modeling:**

**Solution 1:**

The first model I created followed the simple algorithm created by Nick Koprowicz. As was explained in the benchmarking, the model was created by assigning a weight to each word in each tweet, and adding them up together for each possible substring for each tweet, and finding the substring that adds up to the highest total weight. I used the CountVectorizer feature from Scikit Learn in order to accomplish this. This function is very useful in establishing a dictionary of all the words used in this dataset. Using the max\_df and min\_df arguments, I was able to filter out words that occurred too often or too little. Words like “the”, “but”, or “an” are just filler words, and don’t need to have a weight for this algorithm as they won’t affect the sentiment. They are the ones that appear more often, and so they can be filtered out by setting max\_df to 0.95. Words that happen less than twice (min\_df = 2) are most likely typos, and can be discounted from the dictionary.

I then used the fit\_transform function to collect all the words in the dataset and return a document-term matrix of all the words and their appearances. The transform function then transforms documents to a document-term matrix, which is a subset of the entire dictionary.

I used the transform function to find the document-term distribution for each sentiment, positive, negative, and neutral. I created a dictionary that contained all the words found by the transform function, along with their respective weight. The portion of the tweets containing each word was calculated by adding up how many Tweets the word appeared in(the sum of how often it appeared in the transform dataframe) for each sentiment, and dividing it by number of tweets in the respective sentiment.

I then calculated the final weight by subtracting the portion of tweets within each sentiment the word appeared in and subtracted the portion of times the word appeared out of the sentiment.

Finally, I created a function (Prediction) which took in a text and its corresponding sentiment to output the predicted selected text. This was a very naïve solution, including a variety of nested for loops that drastically increased the run time. I basically used two nested for loops to iterate through the Tweet and output all the possible combinations of words that can make up the selected\_text. Then for each possible substring, I ran another for loop to compare each word to their corresponding weights calculated previously. I added the weights up for each substring, and the algorithm outputted the substring that had the highest total weight. For neutral tweets and tweets with a word count of less than 3, it was found that the selected\_text column contained the exact same words as the Tweet. So if those conditions were met, the Tweet did not have to go through the algorithm. It was just outputted as is.

This solution was not ideal. As mentioned in the benchmarking section, this model does not use the training data to be generated, which is very important since the true selected\_text entities were created by people, and not with a model that can be replicated. I also found that I had to do a certain measure of data cleaning in order to accurately assign weights to the words. I had to convert everything to lowercase, and the selected text is not cleaned in the same way, which could have an impact on the results.

Overall, this was a very clever solution, and there is a chance it might reflect the true meaning of Tweets better than the training data given selected\_text was able to. But for the sake of this competition we are going to need a traditional machine learning model that trains on the given data, and predicts the selected text based on that.

***After testing this model on the training data, I was able to achieve a Jaccard score of 0.5841617***

**Solution 2:**

For this model, I decided to use Named Entity Recognition (NER). I started by cleaning the data. It was found that tweets with a word count less than 3 are the exact same as the selected text, so I filtered them from my model. I created two separate models; one to train on the positive Tweets, and one to train on the negative Tweets. The package “spacy” was used for this case. This model is an empty convolutional neural network, with not many options for hyperparameter tuning. It is a fairly simple model with a constant, predefined learning rate where we have to input the training data in a specific format as defined in the code. We are feeding the model the entire tweet, the start and end location of the selected\_text, and the selected text in a specific dictionary format. Once this is done, we can define a blank model, where we can set the interpreted language to English.

The data is broken up into “minibatches”, or batches of items. This is similar to K-Folds from Sklearn, where the data is broken into batches, and is trained/validated on itself to further improve the model. On Kaggle, every author I looked at who used this model kept a batch size of 2, and after experimenting around with other iteration numbers, I found that this led to an optimal solution.

The Spacy NER is a great solution to this challenge; it is a relatively simple neural network that has been pre-created; it just needs to be trained with the parameters. It doesn’t respond too well to typos, however, and some of the error may be as a result of grammatical errors or spelling mistakes than skew the relationship between the entities and the text as a whole that is being trained. In order to further improve on this model, a BERT model is suggested. These models look at the text surrounding the selected\_text as well, to further understand the context of the entity within the text. This can reduce error from typos and grammatical errors in the text and can increase the accuracy of the model.

***After testing this model on the training data, I was able to achieve a Jaccard score of 0.61835***

**Solution 3:**

This model adds onto the Spacy NER model, by using a BERT method of classification. It was mainly inspired by an internet article I read on BERT models [7]. They applied this model the Stanford Question Answering Dataset (SQuAD) which is a very popular dataset to learn natural language processing on. Sourav Kumar Goyal, an author on Kaggle [8] had the idea that we could treat this model as a standard Question and Answer Model (QA). Basically the sentiment would be inputted as the question, the full Tweet text would be the “context”, and the selected text is the answer. The model will then create a model that looks for specific entities within the context that properly fit the pre-trained models based on the sentiment. It’s a very clever way to go about solving this problem, and it has been used by many people to use the ROBERTA models, which are models created by Google and Facebook to answer search queries. The pretrained BERT model was found at Huggingface [9].

I started by properly formatting the training data into the appropriate format. I made sure to specify the text as the context, the sentiment as the question, and the selected\_text as the answer. I then fed the data into the QuestionAnsweringModel, which trained the data on a convolution neural network to understand the relationship between the question and the answer. One problem I ran into was the computational power needed to run this model. I initially wanted to use a ROBERTA model, as it is proven to be the most accurate. But unfortunately, my computer could not handle the size of the data with the parameters needed for the ROBERTA model. This Is a 25-layer neural network with 355 million parameters, and had the computer running for 24 hours without even making 50% progress. I decided to use the smallest model possible; the distill-bert. This model is still very powerful, but it reduces the run time by 40% while sacrificing only 5% in accuracy [10]. Unfortunately, I could not optimize the hyperparameter tuning very well. Even with the distbert, the runtime was extremely long (more than a day) so I had to bring the number of epochs down to 1, and the learning rate was 3 \* 10 ^ -02 which is very low. This is a very promising model, and it was used by many authors to get an accuracy score of up to 0.73. My score was 0.6123, which is comparable to the NER. But I believe that in the future, if I optimize the parameters fully and allow the computer to run through a fully optimized ROBERTA model instead of a sub-optimal distilled BERT, it is highly possible for this model to get to the top of the leaderboards.

***After testing this model on the training data, I was able to achieve a Jaccard score of 0.6123***

***Final code can be found at***

[*https://github.com/fall2020-intro-ml-apps/final-project-yedidv*](https://github.com/fall2020-intro-ml-apps/final-project-yedidv)

**References:**

1. <https://www.kaggle.com/c/tweet-sentiment-extraction>
2. <https://www.kaggle.com/nkoprowicz/a-simple-solution-using-only-word-counts>
3. <https://www.kaggle.com/tanulsingh077/twitter-sentiment-extaction-analysis-eda-and-model>
4. <https://www.kaggle.com/shoheiazuma/tweet-sentiment-roberta-pytorch>
5. <https://pytorch.org/hub/pytorch_fairseq_roberta/>
6. <https://www.fast.ai/2018/07/02/adam-weight-decay/>
7. <https://towardsdatascience.com/question-answering-with-bert-xlnet-xlm-and-distilbert-using-simple-transformers-4d8785ee762a>
8. <https://www.kaggle.com/souravkgoyal/tweet-extraction-eda-simple-transformer-squad>
9. <https://huggingface.co/transformers/pretrained_models.html>
10. https://huggingface.co/transformers/model\_doc/distilbert.html