CSC110 Project: Climate Twitter Analyzer

Aditya Mehrotra

December 13, 2020

Problem Description and Research Question

Social media is filled with many types of people, from different demographics and backgrounds. Each of them have their own opinions and thoughts. In the context of global warming, there are a lot of people who do and do not believe in climate change for various reasons. Understanding the sentiment of people on social media in relation to climate change allows us to better understand how people in certain social media spaces generally feel about climate change.

This leads me to my research question:

Can we create a model which can classify the author's sentiment regarding climate change for tweets?

I want to see if it is possible to take existing natural language processing methods and use them to preprocess data and perform multiclass classification on tweets to infer their sentiment.

To understand this problem statement, the reader needs to understand what sentiment means from a natural language processing standpoint (”Sentiment Analysis”, 2020). Sentiment analysis is a sub-field of natural language processing which works towards identifying certain opinions/feelings from text and quantifying them into a way that computers can understand and use for other models. Hence, for our model stated in the research question, we want it to be able to take twitter text, convert it into a numerical representation, perform a computation and output a class.

If the model performs well, we have shown that there's successful ways of classifying people's sentiment regarding climate change on twitter. This means that we can extend this to other types of social media, such as YouTube, Instagram and Facebook. Collecting sentiment data from a variety of social media sources and computing their classes allows us to see how user sentiment data varies from platform to platform.

I specifically chose this problem because I think this problem and the methodologies to solve it are very intuitive. It's clear what our goal and data is, as sentiment and what a tweet is composed of/represents is easily understood. Additionally, basic NLP methods such as filtration and cosine similarity are easy to understand and visualize/implement given what we learned in CSC110.

Dataset Description

My dataset is called Twitter Climate Change Sentiment Dataset, I downloaded it from Kaggle. The collection of the data was funded by a Canada Foundation for Innovation JELF Grant to Chris Bauch, University of Waterloo. It has tweets pertaining to climate change that were collected between Apr 27, 2015 and Feb 21, 2018 and in total it has 43943 tweets. The dataset itself is a .csv file, with its first column corresponding to the sentiment categories given below:

class number	class meaning
2	The tweet is related to news about climate change
1	The tweet supports the belief of man-made climate change
0	The tweet is neutral
-1	The tweet does not believe in man-made climate change

Here are the columns of the dataset as they appear in the CSV

Output class	Tweet	Tweet ID
class number	”Some text”	XXXXXXXXXXXX (each X is a number)

I will not be using the ID of the tweet in my computations.

Computational Overview

My program consists of three major components, which I will explain in order:

1. Preprocessing
2. Visualization
3. Modelling

Preprocessing

Before we do any modelling/visualization, the data needs to be cleaned first. There's a lot of things that aren't needed such as punctuation, special characters and emojis and so on.

I decided to package all of my preprocessing functions into a class called *DataLoader*, which resides in the *datareader.py* file. Within this class, I used the *pandas* python library to read the .csv dataset and access all elements within the first two columns, as it allows me to operate on the CSV with standard python indexing operations. When the user calls the *prepare_data* function, it iterates over all samples in the dataset and filters each sample to remove unnecessary features from the tweet, storing the filtered tweet and label in new lists. To perform the filtration, I defined a function called *_filter_tweet*. This function applies a series of filtrations to the data, that each have their own function. Here are the individual filters I defined:

- Remove the links
- Remove periods and commas from each word
- Remove words that have @ in them, as these are retweets, which aren't useful in predicting sentiment
- Remove punctuation from each word. This is done using the inbuilt *string* library, which gives us a set of all special characters.
- Remove all non-ascii characters
- Convert all uppercase characters to lowercase
- Remove all stop words. These are words which don't really contribute to our prediction of sentiment, such as “a” or “the”. We imported this from the *NLTK* library and converted it into a set, so we can have $\Theta(1)$ time complexity for searching if an element is inside the set of stopwords.
- Remove numbers from each word
- Remove all empty strings
- Remove all ampersands, as twitter represents these as “amp” rather than using the character ”&”. Therefore, we have to filter these individually.

Visualization

There isn't much to visualize in the modelling portion of this project, so I decided to visualize word frequency corresponding to each class for the dataset instead. I defined a function in *generate_dictionary.py* called *compute_class_word_frequency_dicts*. After applying my preprocessing to the dataset, This function computes a dictionary which maps a word to the frequency of that word in samples of a certain class. This is an example of what the frequency dictionary for the climate change nonbeliever class might look like:

```
{
climate: 2003,
change: 1632,
hoax: 300,
obama: 200
...
}
```

I define a separate function in *visualization.py* called *visualize_class_words* which plots a histogram of the 20 most commonly used words in a class, where the user passes the class as a parameter. This function uses the frequency dictionary described above to find the top 20 words that have the highest frequencies for a given class. The library used here is *matplotlib*, which has a function *matplotlib.pyplot.bar*, that lets us plot a bar graph if we pass in a list of words and a list of frequencies. This graph lets us see the words that people who have a certain sentiment tend to use in a visual manner.

Modelling

Theory and formulas

We need to define a couple terms here:

- Training set: This corresponds to a portion of the dataset that we have set aside to fit our model to the data.
- Testing set: This corresponds to a portion of the dataset that we have set aside to test how well our model performs. Our model will not see these samples in the training
- Query: A tweet which we want to classify the sentiment of.

Here is the computational model I used to predict what sentiment a tweet belonged to:

1. Term Frequency (TF) scores:
   - This is defined from the following formula:

   $$TF(word) = \frac{\text{num of times word appears in a sentence}}{\text{num of words in a sentence}}$$

   - For example, if we have the sentence "I am a student I am", we would see the following TF scores for each word:

   ```
   I: 2/6
   am: 2/6
   a: 1/6
   student: 1/6
   ```

2. Inverse document frequency (IDF) scores:
   - This is defined as the following:

   $$IDF(word) = 1 + \log_e \frac{\text{Total number of tweets in the training set}}{\text{Total number of tweets in the training set which have } word \text{ in it}}$$

   - For example if we only had the sentences "I am a student" and "I am king" in our training set, our IDF score for each word would be:

   ```
   I: 1
   am: 1
   a: 1.69
   student: 1.69
   king: 1.69
   ```

3. TF*IDF scores:

- This is defined as
$$TF(word) * IDF(word)$$
- For example, if we had the sentence ”I student” (note that the TF score for each word is simply 0.5) and we wanted to compute the TF*IDF score of each word using the IDF scores from our previous example, we would get the following TF*IDF scores:

```
I: 0.5 * 1 = 0.5
student: 0.5 * 1.69 = 0.845
```

4. Cosine similarity:
   - Let $\vec{v}_1 \in R^n$ and $\vec{v}_2 \in R^n$, we define the *cosine similarity* between $\vec{v}_1$ and $\vec{v}_2$ as:
$$Similarity(\vec{v}_1, \vec{v}_2) = \frac{\vec{v}_1 \cdot \vec{v}_2}{||\vec{v}_1|| * ||\vec{v}_2||}$$
   The important thing about this formula is that the greater this number is, the more similar vectors $\vec{v}_1$ and $\vec{v}_2$ are.

Our goal is to find the sample in the training set that has the MAXIMUM cosine similarity with the query tweet. We output the sentiment of this sample as our prediction.

Worked example

Putting these concepts together gives us a functioning sentiment classifier, here’s an example walk through of how this works:

1. First we establish our training set and query:

   Training set: [”the”, ”warm”, ”room”] (sentiment: 1), [”the”, ”yellow”, ”duck] (sentiment: -1)

   Query: [”the”, ”room]
2. Then we compute the IDF scores for each element of our training set to create a mapping of words to IDF scores

```
"the": 1
"warm": 1.69
"room: 1.69
"yellow": 1.69
"duck" 1.69
```

3. Finally, we compute TF*IDF scores and cosine similarity. Let’s say we’re comparing our first training sample and the query, we get the following TF*IDF scores for both:

```
training sample : {the: 0.33, warm : 0.56: room: 0.56}
query: {the: 0.5, room: 0.85}
```

   Notice that these two lists are not the same size, since there is a word in the training set sample that isn’t in the query. Therefore, we remove this word and end up with the following two lists:

```
training sample: [0.33, 0.56]
query: [0.5, 0.85]
```

   From here on, these lists are now considered as vectors with two elements inside of them. Using these two vectors, we compute a cosine similarity of: 0.99.

   We repeat this for the other training set sample (not shown here), and end up with a cosine similarity of 0.55.

   Since the training sample with the highest cosine similarity has label of 1, we predict that this query has a label of 1.

Implementation details

The model above is handled through methods/classes defined over a few files, here they are in order:

1. I created training and testing sets using a function called *train_test_split*, defined in *model_metrics.py*. This function divided a certain percentage of the dataset decided by the user as the training set.

2. Using this training set, I computed a mapping of words to IDF scores in *generate_idf_dictionary* in *generate_dictionary.py*.

3. I define all the functions which take care of the modelling in *graph.py*, using the *Node* and *VectorGraph classes.* The *VectorGraph* class has a list of *Node* dataclasses, where each node contains a preprocessed tweet from the training set and the sentiment for that tweet.

4. Given a query tweet, I compute the TF*IDF scores for the query and iterate through each node in the graph, computing the list of TF*IDF scores for the node. This gives me two lists of TF*IDF scores, one for the node and one for the query. The lists are converted into one dimensional *numpy* arrays (mathematically, these are vectors), which let me perform linear algebra computations. Using these *numpy* arrays, I compute the cosine similarity. This repeats for every node in the list within the *VectorGraph*, I store the sentiment of the node that has the highest cosine similarity. Once I have iterated through all nodes in the graph, I output the sentiment of the training set sample with the highest cosine similarity to the query.

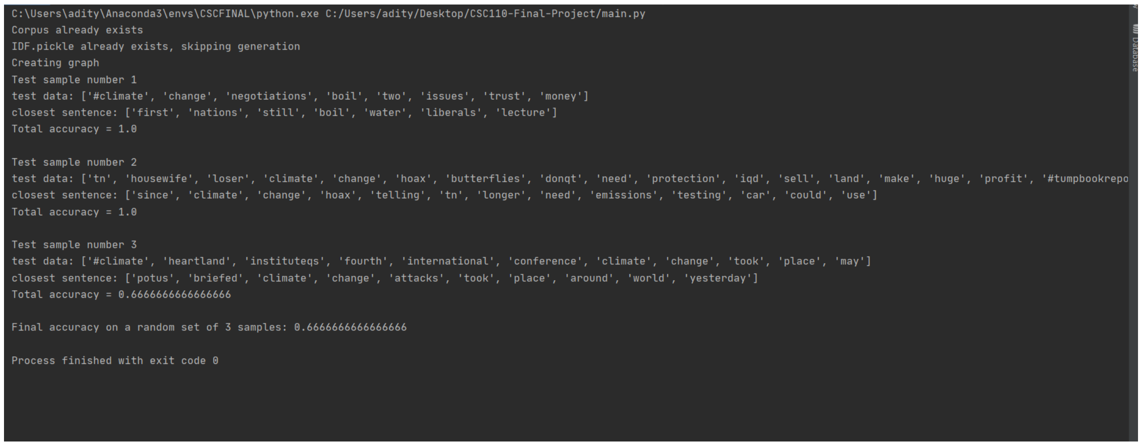
5. In order to test this model, I defined a function in *model_metrics.py* called *compute_accuracy* which loops over a testing set tweets and predicts the sentiment for each tweet using the *VectorGraph*. The number of tweets it predicted correctly are stored and at the end of the iteration over the testing set, the final accuracy of the model is outputted. This lets us test how well our model is performing.

How to run the program

1. Install all libraries instructed in the requirements.txt

2. Download the dataset from this url https://www.kaggle.com/edqian/twitter-climate-change-sentiment-dataset. You should get a zip file named "archive", please unzip the .csv file inside and put it in the same folder as the rest of my program. Everything should be in the same folder.

3. Run main.py. You should see that I've separated two blocks of function calls, one under the comment that has "VISUALIZATION" and another that has "MODELLING". **There are instructions on how to use these two blocks in the file**. Essentially one is for viewing matplotlib graphs and another is for training and testing the model. The "MODELLING" portion will train a model and iterate through the testing set, displaying the current accuracy of the model as it iterates, reporting a final accuracy in the end. This will all be shown in the console. On the other hand, the "VISUALIZATION" portion, should show matplotlib graphs.



```
C:\Users\adity\Anaconda3\envs\CSCFINAL\python.exe C:/Users/adity/Desktop/CSC110-Final-Project/main.py
Corpus already exists
IDF.pickle already exists, skipping generation
Creating graph
Test sample number 1
test data: ['#climate', 'change', 'negotiations', 'boil', 'two', 'issues', 'trust', 'money']
closest sentence: ['first', 'nations', 'still', 'boil', 'water', 'liberals', 'lecture']
Total accuracy = 1.0

Test sample number 2
test data: ['tn', 'housewife', 'loser', 'climate', 'change', 'hoax', 'butterflies', 'donqt', 'need', 'protection', 'iqd', 'sell', 'land', 'make', 'huge', 'profit', '#tumpbookrepo
closest sentence: ['since', 'climate', 'change', 'hoax', 'telling', 'tn', 'longer', 'need', 'emissions', 'testing', 'car', 'could', 'use']
Total accuracy = 1.0

Test sample number 3
test data: ['#climate', 'heartland', 'instituteqs', 'fourth', 'international', 'conference', 'climate', 'change', 'took', 'place', 'may']
closest sentence: ['potus', 'briefed', 'climate', 'change', 'attacks', 'took', 'place', 'around', 'world', 'yesterday']
Total accuracy = 0.6666666666666666

Final accuracy on a random set of 3 samples: 0.6666666666666666

Process finished with exit code 0
```

Figure 1: A sample output of running the MODELLING block of code

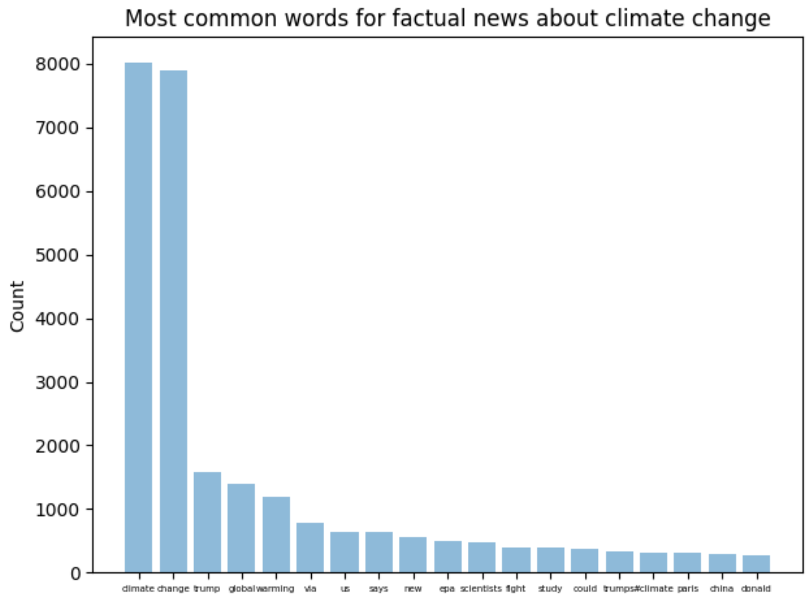


Figure 2: A sample output of running the VISUALIZATION block of code

Changes between the project proposal and my submission

The dataset and goal are the same between my initial draft and this submission. However, a major change is the computational model that I am using. I had a simpler method in my proposal, though the TA feedback gave me some ideas to proceed in the direction of cosine similarity, which is the direction I decided to pursue.

Discussion

Restating my initial research question:

Can we create a model which can classify the author's sentiment regarding climate change for tweets?

To start off, a "model" is a very generic term. A sequence of if/else statements or even a random sampler can count as a "model". Any function $f(x)$ which takes in a tweet and outputs a prediction for sentiment within our desired codomain (the set $\{-1, 0, 1, 2\}$), is a valid model, the process in which it does so does not matter. Therefore, we technically answered our research question with the cosine similarity model. What is more important however, is to see how well our model performs.

Looking to model performance

My *train_test_split* function and *model_metrics.py* file exist to quantify how well our model performs on data it has not seen before, using a simple accuracy metric. However, to see whether our model is "good", we need to compare it to some sort of baseline. In this case, I will define a very simple model for this sentiment classification task as a baseline – Random sampling.

We define $f(x)$ as a random sampler, which for a given tweet x, randomly samples from the set $\{-1, 0, 1, 2\}$. The output is the predicted sentiment. Therefore, we expect the probability for each class to be outputted as $\frac{1}{4}$ or 25%. This means, it has a 25% chance of predicting the right class. In summary, we expect our random sampler model $f(x)$ to get roughly 25 samples right out of 100.

Now that we've defined a baseline for our model, we can compare the cosine similarity model to it. In order to do this, I ran 100 iterations of testing a model using a test set of size 100, where the model is trained on 80% of the dataset. After doing this, I computed the mean test set accuracy over the 100 iterations and got a mean of 0.57 or 57%. Comparing this to the random sampler, which we would expect to have an accuracy of roughly 25%, this sentiment classifier performs much better. Therefore, we can say that we've created a model which performs better than a baseline of a simple random classifier.

Limitations

It's clear that we've built a model that performs better than a simple random model. However, its accuracy is still very low. A model with 56% accuracy is not something that would be deployed in a professional setting, as it will make a substantial number of mistakes. The root cause of this can be two things:

1. The dataset has issues (class imbalance, missing values, unclean data etc..)
2. The model is not suited for this task

Potential Issue 1: Dataset

Throughout the dataset, I noticed a few things that seemed to interfere with the predictive power of my model. The first thing was that there were a lot of samples in the dataset which were obfuscated by random group of non-ascii characters, here's an example:

```
RT @SeneddCCERA: WeÃƒÂ¢Ã¢â€šÂ¬Ã¢â€žÂ¢ve announced the creation
of a new expert panel to help scrutinise @WelshGovernment
progress on climate change https://ÃƒÂ¢Ã¢â€šÂ¬ÃÂ¦
```

I did create filters for non-ascii characters, but they simply removed any word that included them from the data. This means any word that was obfuscated by ascii characters was filtered out and not used by the model. This causes a loss of information.

Additionally there was severe class imblance in the dataset. From the Kaggle page, the following graphic shows the tweet distribution:

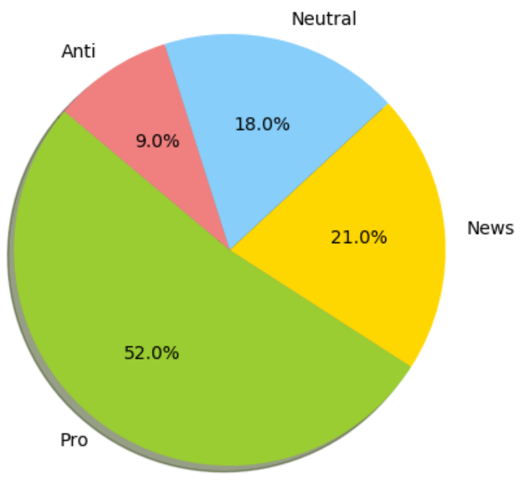


Figure 3: Dataset class distribution

As shown, there's a severe class imbalance, as the pro climate change tweets take up more than half of the dataset while the anti climate change tweets are only 9% in comparison. In the context of our model, this means that there are less nodes in our *VectorGraph* for less represented classes. This means that tweets with negative sentiment have a high chance of being closest to a tweet with pro, factual or neural sentiment as the graph is populated by so many nodes corresponding to other classes. The solution to this would be more data for underrepresented classes.

Potential Issue 2: Algorithms weaknesses

There are common issues with TF-IDF and cosine similarity that have been recognized over the years. A key issue is that it requires a lot of training data to have a good predictor that uses this algorithm. This is especially true in twitter tweets, since there is so many different word combinations, slangs, there's not always a tweet in the training set that is very close to a given query tweet. This means that a lot more data is required if we want to increase how well our predictor performs for something as variable as tweets.

Next steps and other directions to explore

There are a variety of more powerful NLP models out there that work very well for sentiment classification. An example of this is deep learning models. These models still do require a lot of data to train, but they have been performing very well in problem domains such as sentiment classification for text. Additionally, there also is the option of collecting more data and seeing how the model performs, though this will most likely cost money and take a lot of time.

Conclusion

Throughout this research project, we've found that the TF-IDF model with cosine similarity (with ample text filtering) which performs better than randomly selecting a sentiment class for a tweet. We've also addressed the limitations of this model/dataset and what we can do to get a better performance.

References

Natural Language Toolkit¶. (n.d.). Retrieved November 06, 2020, from https://www.nltk.org/

Qian, E. (2019, November 13). Twitter Climate Change Sentiment Dataset. Retrieved November 06, 2020, from https://www.kaggle.com/edqian/twitter-climate-change-sentiment-dataset

NumPy. (n.d.). Retrieved November 06, 2020, from https://numpy.org/

Pandas documentation¶. (n.d.). Retrieved November 06, 2020, from https://pandas.pydata.org/pandas-docs/stable

Vembunarayanan, J., Says:, S., Says:, S., Says:, J., Says:, S., Says:, H., . . . Says:, S. (2016, November 16). Tf-Idf and

Cosine similarity. Retrieved December 14, 2020, from https://janav.wordpress.com/2013/10/27/tf-idf-and-cosine-similarity/

Rit19Check out this Author's contributed articles., Rit19, &amp; Check out this Author's contributed articles. (2019, August 07). Tf-idf Model for Page Ranking. Retrieved December 14, 2020, from https://www.geeksforgeeks.org/tf-idf-model-for-page-ranking/

Os - Miscellaneous operating system interfaces¶. (n.d.). Retrieved December 14, 2020, from https://docs.python.org/3/library

Thomas Kimber (https://stats.stackexchange.com/users/59641/thomas-kimber), Drawbacks with Cosine Similarity, URL (version: 2017-05-16): https://stats.stackexchange.com/q/279931

Sentiment analysis. (2020, December 13). Retrieved December 14, 2020, from https://en.wikipedia.org/wiki/Sentiment_analys

String - Common string operations¶. (n.d.). Retrieved December 14, 2020, from https://docs.python.org/3/library/string.htm