Sentiment Analysis on Tweets Data During COVID Pandemic

Taixi Lu
Yvonne Qiu
Zoey Ye
Zenan Wang
Jiahuan He

I. Introduction

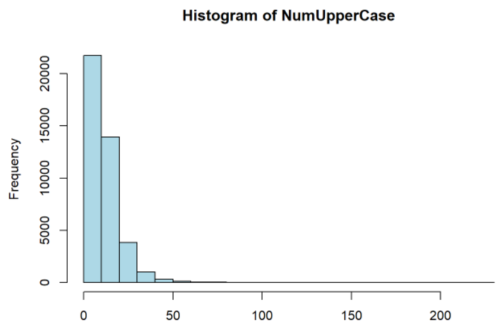
This report is about the sentiment analysis on Tweets data that are gathered during the COVID pandemic. The dataset used is the tweet posts collected from March 16th, 2020, to April 14th, 2020. There are 6 variables in the data which are UserName, ScreenName, Location, TweetAt, OriginalTweet, and Sentiment. The response variable is Sentiment, and it has five levels ranging from extremely negative to extremely positive. The first two variables are meaningless since they are just the IDs. The variable "Location" should also not be used because it has inconsistent criteria: the first location is a city name, the second one is a country, and the last one is the location coordinate. Thus, we choose to use variables "TweetAt", "Original Tweet" and "Sentiment" to do the analysis. The "Original Tweet" variable is the sentences users wrote in their tweets illustrating how they feel about the COVID. The link in the reference can lead readers to find the dataset we applied located on the Kaggle website. The dimension of the data is 41,157 x 3,798.

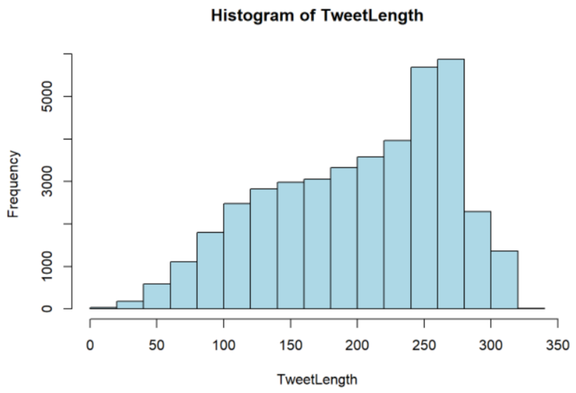
II. Goal

We believe such analysis is a meaningful process by conducting different models in the field of machine learning. The model with high accuracy helps us predict users' sentiments towards COVID based on their tweet posts. In the social sense, through the analysis, organizations like companies and schools can use such methods to prevent employees or students from being too stressed and maintaining good mental health conditions.

III. Data Pre-Processing

Besides data collection, EDA is a crucial part before formal analysis of data set. Since a whole sentence in english is far from readable and processable, we tokenize each tweet by separating the sentence into a list of words. Based on the list of words, we create three more predictors, which are "AtPresent", "TweetLength", and "NumUpperCase." "TweetLength" is the length of the original post, whose value being larger indicates the poster is likely to get angry and willing to share opinions with others. 'NumUpperCase' is the number of uppercase characters in the tweet, which basically works the same as 'tweetlength'. "AtPresent' signals the presence of '@' in the post. If @ is present in the post, the user may want others to see his post, which shows a tendency to spread his opinions toward the public. Below are some barplots exhibiting the distribution of the three factors we have created.



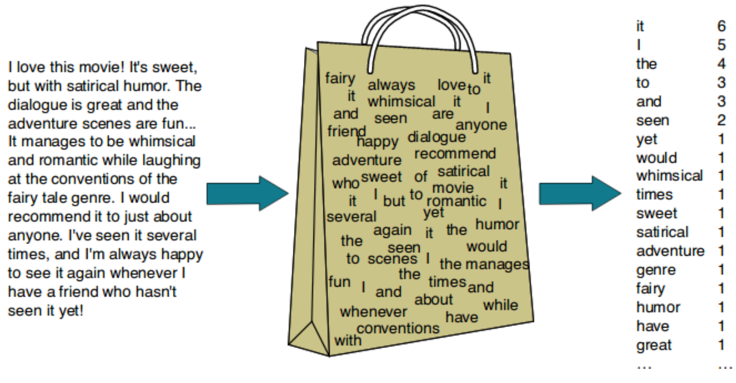


IV. Models, Results, and Analysis

A. Bag of Words Model

In our first modeling attempt, the highest accuracy we gained was 38.23% from the Naive Bayes Model, and the lowest was 14.68%, which is even lower than a random guess (20% since the response variable has 5 levels), from the logistic regression model. None of the models work well. The reason is that the three new variables we created from the text data only contain very limited information. From the length of the tweet, the number of uppercase characters, and whether there is an "@" character, we cannot have a clear figure of what the original tweet is. So, the question becomes how to extract information from the text data.

Let's take a look at an example in our data: "Please Please Don't Ignore My Tweet. Please Note That Don't Use Online Shopping. It's the Fastest And Easy Way For Spreading COVID 19. Reach to You Say No To Online Shopping." We can see this sentence as an array of factors (each distinct word is one level). Then, the text is no more than just a combination of those factors. And all the information in a combination can be answered by the number of each level and the ordering sequence. In another word, if we know how many times a certain word shows up in what sequence, we can always construct the original text. Here, we discard the ordering information, which takes up to the bag of words model.



I love this movie! It's sweet, but with satirical humor. The dialogue is great and the adventure scenes are fun... It manages to be whimsical and romantic while laughing at the conventions of the fairy tale genre. I would recommend it to just about anyone. I've seen it several times, and I'm always happy to see it again whenever I have a friend who hasn't seen it yet!

it	6
I	5
the	4
to	3
and	3
seen	2
yet	1
would	1
whimsical	1
times	1
sweet	1
satirical	1
adventure	1
genre	1
fairy	1
humor	1
have	1
great	1
...	...

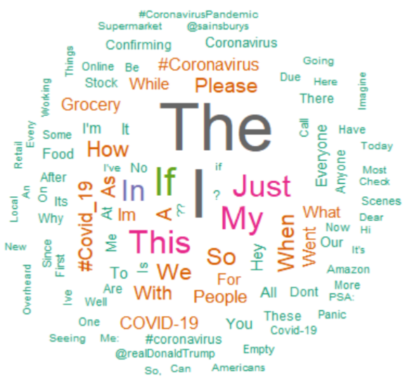
The bag of words model takes a sentence into a bag to remove the ordering information and then comes up with a dictionary data type. The keys are the levels (specific word), and the values are the frequency. This particular approach gives us more information about the text than our first attempt, but the drawback is it discards the sequence of the words. Two sentences with the same bag of words but different ordering can have extremely different sentiments.

We construct two classes: BagOfWords (preliminary data type to store the text), and Document (to read the full csv file to build a dictionary) to implement the Bag of Words model. The major function here is BagOfWords.add_word(self, word), and Document just loop through and tokenize every tweet and then call BagOfWords.add_word().

```
    def add_word(self,word):
        self.__number_of_words += 1
        if word in self.__bag_of_words:
            self.__bag_of_words[word] += 1
        else:
            self.__bag_of_words[word] = 1
```

```
words = np.array(list(dict((k, v) for k, v in words.items() if v <= 80000).keys()))
```

In this process of creating our dictionary, we found that there are more than 100,000 distinct words in our training data, which means the final data would be more than 41,157 x 100,000. When trying to use this huge data to fit models such as K Nearest Neighbors using cross validation, it requires a huge amount of computation power, and our devices couldn't find an answer. So, we decided to add a tuning parameter that discards all words with frequency higher than 80,000 in the dictionary. We did not use cross validation to find an optimal tuning parameter, but we compared several values, and 80,000 seemed to have the best performance. This makes sense because words that show up too frequently tend to be vague and have less predictive powers. For example, the word "I" is common in English. But the two sentences: "I am happy" and "I am sad" imply very different sentiments. Here is a word cloud graph of our data.



Words that are bigger in this plot, such as "I" and "The" have higher frequencies, but they contain very limited information about the sentiment. Other words, such as "COVID-19", "Food", and "Stock" can give much more predictive power. After tuning the dictionary, we have a bag of word model data that contains more than 7,000 columns. Here, we use all the predictors in our models, which does not result in overfitting. This is because the data is very high-dimensional, and according to the definition of double descent, the error would eventually drop in high dimensions. Now we have a new data, and we fit the 4 models again to compare the result.

B. Random Forest Model

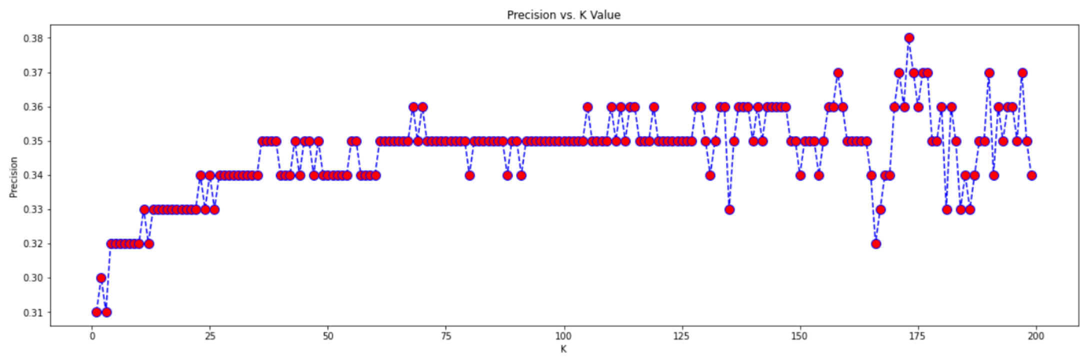
Random Forest method is usually used in classification or regression problems. A random forest algorithm is made up of many decision trees. The “forest” generated by the random forest algorithm is trained through bagging or bootstrap aggregating. It processes the data set including continuous variables as in the case of regression and categorical variables as in the case of classification. In our case, the random forest is built on different samples within data and takes their majority vote for classification.

```
sentiment.mod<-randomForest(Sentiment~.,data=new_train, importance=TRUE,ntree=100,mtry=(dim(new_train)[2]-1))
sentiment.mod$importance
importance(sentiment.mod)
predict=predict(sentiment.mod, newdata=new_test[-2])
table(new_test$Sentiment, round(predict, digits = 0))
```
```

```
    1   2   3   4   5
1   4 165 389  33   1
2   7 174 789  71   0
3   0  54 535  30   0
4   0  86 721 139   1
5   1  38 394 156  10
```

Based on the confusion matrix, we got an accuracy of 22.7% ((4+174+535+139+10)/3798), which is higher than a random guess (20%), but not the best model.

C. KNN Model

The next model we chose to implement is the k-nearest neighbor algorithm. To get some firsthand information of the effectiveness of applying KNN algorithm, we randomly pick the k value of 20 at first, and get the confusion matrix below. We get an accuracy of 33 percent, so far a decent accuracy. Furthermore, the confusion matrix suggests the model works most effectively on predicting the ‘neutral’ sentiment, partly due to the fact that posts written by people more or less indifferent with the Covid pandemic tend to be short and concise, with the ‘@’ character absent.

```
[[ 80 161 138 490]
 [ 84 193 144 601]
 [ 44 115 474 531]
 [133 271 369 956]]
                   precision    recall  f1-score   support

Extremely Negative      0.23      0.09      0.13       869
Extremely Positive      0.26      0.19      0.22      1022
           Neutral      0.42      0.41      0.41      1164
          Positive      0.37      0.55      0.44      1729

          accuracy                          0.36      4784
         macro avg      0.32      0.31      0.30      4784
      weighted avg      0.33      0.36      0.33      4784
```

However, a crucial part of this algorithm is to find the most suitable K value that guarantees the highest accuracy. The model is likely to get more accurate when K grows larger. To begin with, we chose cross-validation, and drew a figure that plots accuracy against each K value.
```

```
Maximum Precision: 0.38 at K = 173
```



Unfortunately, 200 is the upper bound of the K value range that we would like to test between; since when K is getting larger, the time the model takes to make predictions grows exponentially. Finally, the most suitable K value is 173, which contributes to an accuracy of 38 percent. It appears that larger K value does improve the accuracy of the model, but for the sake of time and the disproportionate growth of accuracy given the growth rate of K value (the accuracy increases only 5 percent even with a tremendous increase in the value of K and time to fit the model), we will stop here and stick with the K value of 173.

D. Naive Bayes

Naive Bayes assumes that each variable is conditionally independent from one another. And it chooses the highest probability as the result.

$$p(f_1, \dots, f_n | c) = \prod_{i=1}^{n} p(f_i | c)$$

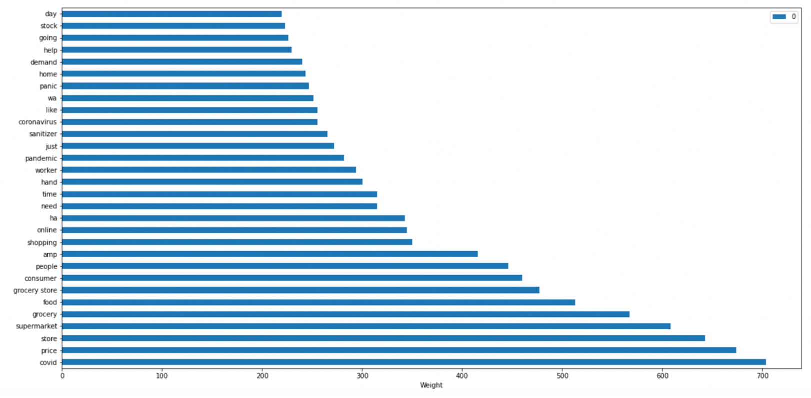
Here, our data is the count of the number of times a word shows up. So, a Multinomial Naive Bayes model could fit the data better than a Guassian Naive Bayes model. The model gives an accuracy of 49%.

```
                    precision    recall  f1-score   support

Extremely Negative       0.52      0.52      0.52      1134
Extremely Positive       0.51      0.58      0.55      1298
          Negative       0.43      0.45      0.44      1992
           Neutral       0.55      0.50      0.52      1484
          Positive       0.44      0.42      0.43      2324

          accuracy                           0.48      8232
         macro avg       0.49      0.49      0.49      8232
      weighted avg       0.48      0.48      0.48      8232

[[590  20 370  44 110]
 [ 27 756  93  43 379]
 [351 104 891 230 416]
 [ 55  66 285 738 340]
 [113 526 429 281 975]]
```

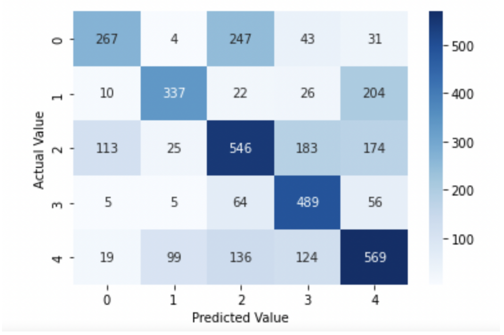
E. Logistic Regression

We started to model the logistic regression model using scikit-learn package in Python. For a model to be able to process text input, vectorization is an imperative step. Among the various

ways for the text feature representation, we chose Binary and TF-IDF as our vectorizers and compare the accuracy of these two different approaches.
To check the binary representation of all unigrams, we used sklearn’s CountVectorizer object to extract all the word features. If the word appears in less than 5 documents or more than 100% of the documents, it would be ignored. After fitting the first logistic regression model, we got an accuracy of 60%, which is a relative ideal result for our first model.
Then we used TF-IDF as a vectorizer to weight the importance of each text feature relevant to the document(as displayed in graph). The TfidfVectorizer object was used for extraction, and we got an accuracy of 57% for our second model. There is no improvement for the model when we use TF-IDF as the vectorizer.



To increase the accuracy of our logistic regression model, we imported the GridSearchCv function to help with the hyperparameter tuning process. We found the best estimator across all searched params is “LogisticRegression(C=1, penalty='l1', random_state=6, solver='saga')” and produced an accuracy of 63%. The final prediction of the test data also returned an accuracy of 58% (shown in the figure on the left).



V. Neural Network/BERTS

Context

Bert which stands for Bidirectional Encoder Representations from Transformers is a transformer-based machine learning technique for natural language processing pre-training developed by google. The Bert model is useful in neural machine translation, Question answering, Sentiment Analysis, text summerization.A transformer is a deep learning model that adopts the mechanism of self attention. In neural networks, attention is a technique that mimics cognitive attention. Learning which part of the data is more important than others depends on the context and is trained by gradient descent. Like recurrent neural networks (RNNs), transformers are designed to handle sequential input data. However, unlike RNNs, transformers do not necessarily process the data in order.

Recurrent neural networks could be useful in translation, and LSTM is a modified version of RNN. Clearly RNNs are versatile but for language problems they have their disadvantages. 1. slow train in training reference, since the input words are processed one word at a time. 2. They don't truly understand the context of a word, since they only learn about words based on the words that come before it, but in reality the context of a word really depends on the sentence as a whole. Transformers work well for sequence to sequence problems, but for the specific complex natural language problems, like question answering and text summarization, the main drawbacks associated with these are we need a lot of data to train transformers from scratch and the architecture may not be complex enough to understand patterns to solve these language problems, after all transformers weren't designed to be language models, so the word representations generated can still be improved. To address these concerns BERT was introduced.
In order to make the BERT understand the language and solve the problems, it needs to experience two phases. The first is pre-training where the model understands what language and context. Actually, we used the pretrain model directly, but the principal is that people pretrained BERT with mask modeling language and next sentence prediction. For every word, they get the token embedding from the pre trained word piece embeddings, add the position and segment embeddings to account for the ordering of the inputs, these are then passed into bert, which under the hood is a stack of transformer encoders and it outputs a bunch of forward vectors for mass language modeling, and a binary value for an extended prediction.

Fine Tuning

Then we come to the second phase, fine-tuning, where the model has learned the "language", and we use it to solve problems by performing a supervised depending on the task we want to solve, which should happen fast. The "BERT squad" is the Stanford question and answer model that only takes about 30 minutes to fine-tune from a language model for a 91% performance. And performance depends on how big we want BERT to be.

We use Google Colab GPU for training. Since we want to train an extensive neural network, it's best to take advantage of this; otherwise, training will take a long time. For the torch to use the GPU, we need to identify and specify the GPU as the device. Later, we load data onto the device in our training loop. Next, we install the transformers package from Hugging Face, which gives us a PyTorch interface for working with BERT. We've selected the PyTorch interface because it strikes a nice balance between the high-level APIs, which are easy to use but don't provide insight into how things work and TensorFlow code which contains lots of details but often sidetracks us into lessons about TensorFlow when the purpose here is BERT. Currently, the Hugging Face library seems to be the most widely accepted and powerful PyTorch interface for working with BERT. In addition to supporting various pre-trained transformer models, the library also includes pre-built modifications of these models suited to your specific task. The library also has task-specific classes for token classification, question answering, next sentence prediction, etc. Using these pre-built classes simplifies the process of modifying BERT for your purposes.

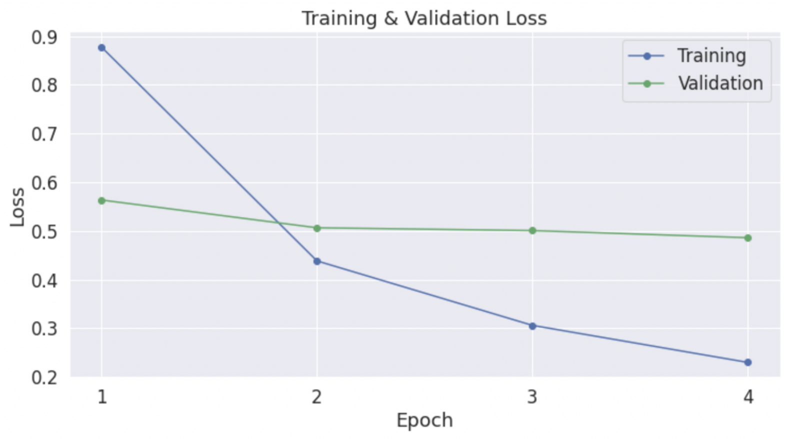
We use The Corpus of Linguistic Acceptability (CoLA) dataset for single sentence classification. It's a set of sentences labelled as grammatically correct or incorrect. It was first published in May of 2018 and is one of the tests included in the "GLUE Benchmark" on which models like BERT are competing. We can see from the file names that both tokenized and raw versions of the data are available. We can't use the pre-tokenized version because we *must* use the tokenizer provided by the model to apply the pre-trained BERT. The reason is that (1) the model has a specific, fixed vocabulary and (2) the BERT tokenizer has a particular way of handling out-of-vocabulary words.

BERT can train on our dataset into the format that we transformed. To feed our text to BERT, it must be split into tokens, and then these tokens must be mapped to their index in the tokenizer vocabulary. The tokenization must be performed by the tokenizer included with BERT. We use the "uncased" version here. Then we needed to finish the required formatting: Add unique tokens to the start and end of each sentence, Pad & truncate all sentences to a single constant length, Explicitly differentiate real tokens from padding tokens with the "attention mask". At the end of every sentence, we need to append the special [SEP] token for special tokens. For classification tasks, we must prepend the special [CLS] token to the beginning of every sentence. This token has special significance. BERT consists of 12 Transformer layers. Each transformer takes in a list of token embeddings and produces the same number of embeddings on the output. The sentences in our dataset have varying lengths. BERT has two constraints: 1. All sentences must be padded or truncated to a single, fixed length. 2. The maximum sentence length is 512 tokens.

The transformers library provides a useful encode function that handles most parsing and data prep steps. Before we are ready to encode our text, we need to decide on a maximum sentence length for padding / truncating.

We can fine-tune the BERT model when we have our input data adequately formatted. We first want to modify the pre-trained BERT model to give outputs for classification, and then we want to continue training the model on our dataset until the entire model, end-to-end, is well-suited for our task. Thankfully, the huggingface PyTorch implementation includes a set of interfaces designed for various NLP tasks. Though these interfaces are all built on a trained BERT model, each has different top layers and output types designed to accommodate their specific NLP task. We use BertForSequenceClassification, the regular BERT model with an added single linear layer for classification that we use as a sentence classifier. The entire pre-trained BERT model and the additional untrained classification layer are trained on our specific task as we feed input data.

After loading our model, we chose recommended value Batch size = 32, Learning rate = 2e-5, Epochs = 4, optimizer Adam W class as training hyperparameters within the stored models. Then we go through a training loop; specifically, we have a training phase and a validation phase for each pass in our loop. Validation loss is used to detect over-fitting.

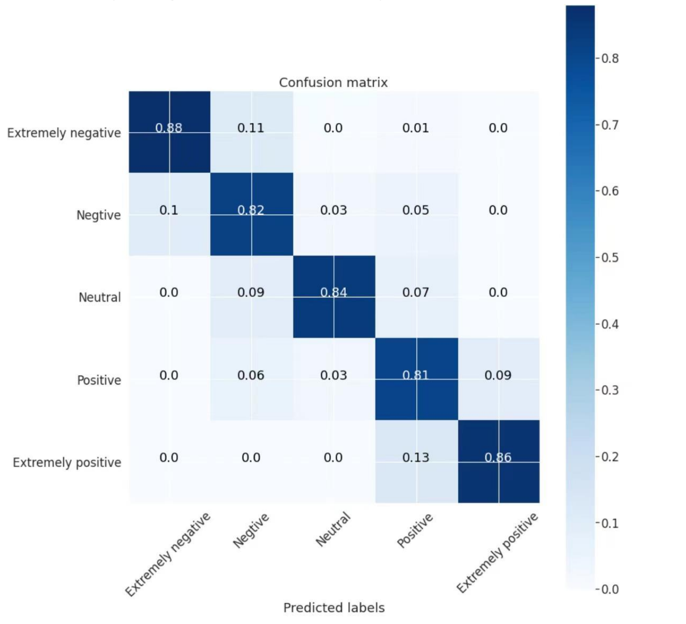


Both the training loss and validation loss are decreasing over time, which means the model is improving. It also indicates that the model is not overfitting the data because the validation loss stays relatively stable after the second epoch.Validation Loss is a more precise measure than accuracy, because with accuracy we don't care about the exact output value, but just which side of a threshold it falls on.If we are predicting the correct answer, but with less confidence, then validation loss will catch this, while accuracy will not.

At last, we apply all of the same steps for the training data to prepare our test data set. We can use our fine-tuned model to generate predictions on the test set with the test set prepared.

<u>Results</u>

In the end, we have obtained a pretty high accuracy results. And from the confusion matrix, the diagonal have dark blue color, meaning they have all above 80% accuracy. To be specific, if a person that is reported as extremely negative, then there will be 88% of predicting exactly extremely negative, and nearly zero chance of predicting either neutral or positive.



	Extremely negative	Negtive	Neutral	Positive	Extremely positive
Extremely negative	0.88	0.11	0.0	0.01	0.0
Negtive	0.1	0.82	0.03	0.05	0.0
Neutral	0.0	0.09	0.84	0.07	0.0
Positive	0.0	0.06	0.03	0.81	0.09
Extremely positive	0.0	0.0	0.0	0.13	0.86

VI. Conclusion

This project allowed our group to perform a methodical investigation on the sentiment of Covid-19 that was expressed by twitter users. Twitter is a great source to extract data since it is useful in understanding the opinion of the people about specific topics, like the pandemic. The preprocessing and feature extraction techniques were used to train the machine learning models, and we used the pre-trained data to predict five sentiment classifications: extremely negative, negative, neutral, positive, and extremely positive. Based on our comparison, the BERT model has the best prediction accuracy of 83.88%. Unlike the traditional machine learning models, such as logistic regression or KNN model, the BERT model increases its accuracy by considering the context from both the left and the right sides of each word. In addition to the tweets sentiment analysis task, we can see the strength of the new cutting-edge model BERT in a wide range of Natural Language Processing problems, for example, providing a better customer/patient experience as a chatbot.

Improvement

Even though the BERT model produced an ideal accuracy on the sentiment prediction, there are still few changes we can make to improve our analysis.

- Since our dataset was downloaded from Kaggle that had already separated into train and test datasets for us, we cannot make sure the data is random, reliable, and credible. In the future, we would like to use Twitter API to retrieve test and train data by ourselves. After gathering our own datasets, we can use Textblob to calculate positive, negative, neutral, polarity and compound parameters from the text, and split the data set into test and train data. The approach can also help us limit the location of tweets by specifying the "coordinates" attribute. This improvement aims to target the population in specific areas and provide more personalized wellness support based on their locations.

- Removing trivial stemmed words is an imperative step for traditional machine learning models. Even after the preprocessing process, our dataset still contains a vast amount of text features. In addition to ranking the text based on their frequency, we can also focus on the user-defined functions that elevate the efficiency of our Lemmatization. Some specific elements we will remove are username(@username) and numeric values that are not valuable for topic analysis, yet we will keep the Alpha-numeric work (Covid19) since most of them are technical terms that are important for our sentiment analysis. This approach could minimize the data processing time for our model as well as increase the efficiency of our prediction.

- There is a rapid increase of emoji usage on social media over the last few years, and many users like to include emoji in their tweets to express their sentiment. We can also include an additional emoji-embedding model to see if emoji co-occurrences can be used as a feature to investigate sentiments.

Reference

Google. (n.d.). *Google colaboratory*. Google Colab. Retrieved May 1, 2022, from https://colab.research.google.com/drive/1MKHzDbNgnwVRchRg0OnVm4jOLI-SZe0N#scrollTo=_QXZhFb4LnV5