Twitter sentiment Analysis

Ali Ghazal
CSCE department
the american university in cairo
Egypt
alighazal@aucegypt.edu

Omer Moussa
CSCE department
American University in Cairo
Egypt
omermosa@aucegypt.edu

Prof. Hossam Sharara
CSCE department
American University in Cairo
Egypt
hossam.sharara@aucegypt.edu

I. ABSTRACT

Sentiment analysis is the classification of emotions within voice or text. It allows businesses to identify customer attitudes toward products, in addition to many other potential benefits from the field. Sentiment analysis can take various forms. There are models that focus on polarity (positive, negative, neutral) and those that tell feelings more accurately(anger, happiness, sadness). They can also predict intentions or future decisions like whether the customer will be interested or not in buying a product. And it is also used in onion mining. which influences many political actions. In this paper we will be discussing the work we have done of detecting text polarity on the body of tweets. We have chosen to work with tweets because they provide a reach range of content and the ease of using its API.

Concepts and Subjects

machine learning, natural language processing

KEYWORDS

bag-of-words, word embedding, SVM, logistic regression, ANN.

II. Introduction:

As the common saying goes, there is so much data, yet too little information. so the challenge of turning all this massive amount of unstructured data into something useful has become extremely demanding. With the majority of content on the internet in text format, text mining techniques are extremely useful to generate useful insights from which fits into so many purposes. text mining and sentiment analysis is an interdisciplinary field which developed due to the parallel developments in the areas of Machine learning, Natural language processing, and information retrieval technologies.

III. Literature Review:

Text classification has been around for so many years. and there have been many approaches that researchers from different fields have been using. linguists have been working with lexical analysis, but the complexity and the rapid evolution of language proved that this approach is not efficient. So, researchers shifted into using statistical models which proved to fit the best to the problem at hand.

First, researchers were working on topic based classification. Which was used in publication on the internet so we would have a better mapping for our knowledge productions. And huge progress was done to this field. Yet, sentiment analysis was a subtle challenge. you can't easily classify words into emotions. because contexts played a huge role of determining the meaning and the underlying emotions.

The famous types of sentiment analysis are:

1- Fine-grained Sentiment Analysis
Usually used to interpret 5-star ratings in a review. For example, very positive is 5 stars and very negative is 1 star. And the numbers between 1 to 5 include levels (basically, a 5 star rating for something.)

2- Emotion Detection
Emotion detection aims at detecting emotions, like happiness, anger, sadness. Many emotion detection systems use lists of words and their corresponding emotions.

3- Multilingual sentiment analysis
Multilingual sentiment analysis is extracting sentiments from data with mixed languages or different languages.

Datasets used:

After careful research we have decided to work with the following two datasets:

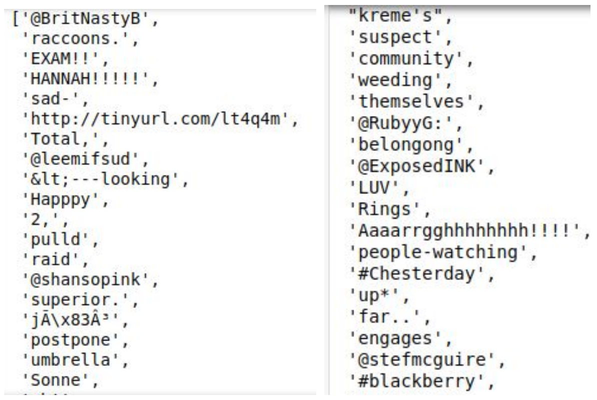
1. TwitterSentiment1.6M from Kaggle
2. 1.04M tweets Dataset uploaded on Github

In total, we had 2.64M labeled tweet to work with

IV. Feature Extraction and pre-processing

The data we contained in the aforementioned datasets was, just like most of the content on the internet, highly unstructured and non-standardized.

Here is a sample of the words used in some of the tweets:



```
['@BritNastyB',
 'raccoons.',
 'EXAM!!',
 'HANNAH!!!!!',
 'sad-',
 'http://tinyurl.com/lt4q4m',
 'Total,',
 '@leemifsud',
 '&lt;---looking',
 'Happpy',
 '2,',
 'pulld',
 'raid',
 '@shansopink',
 'superior.',
 'jÃ\x83Â³',
 'postpone',
 'umbrella',
 'Sonne',
```

```
 "kreme's",
 'suspect',
 'community',
 'weeding',
 'themselves',
 '@RubyyG:',
 'belongong',
 '@ExposedINK',
 'LUV',
 'Rings',
 'Aaaarrgghhhhhhhh!!!!',
 'people-watching',
 '#Chesterday',
 'up*',
 'far..',
 'engages',
 '@stefmcguire',
 '#blackberry',
```

figure[1] - sample of words in the dataset

Following the approaches detailed in [1], here is our preprocessing approach.

As known, twitter allows different formats of texts to be included in the tweet's body. For example, it allows hashtags, links, and mentions.

As mentioned in [1], those types of data do not provide much meaningful value to most models. so such features could be dropped. Those types of features are only relevant in the first place in selecting the tweets. So for example, using the twitter api one can select a certain account or hashtag to extract related tweets. however, this metadata becomes useless afterward.

also, we tried to standardize the language used although this imposed some limitations on our model. That is, by standardizing the language we mean the process of stemming and lemmatization. because on the internet non-standard ways of spelling contains details about the associated emotions with the tweet. for example, writing “YESSS” is quite different from just spelling it in its standard way. However, for simplicity we standardized all of the tweets.

further, we removed stopwords which are considered neutral in most contexts.

Lastly, due to some limitation on the computational power available, we chose to just work with the 1-gram representation. and we further reduced the dataset of words by removing the least frequently used words.

Data Distribution:

The polarity of the tweets is divided almost evenly among our dataset as can be seen in figure [2].

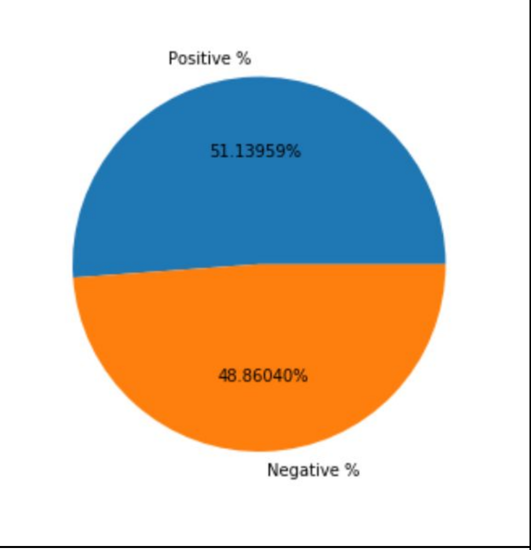


figure [2] tweets sentiments

yet, the words frequencies follows a skewed distributions as in figure [3]

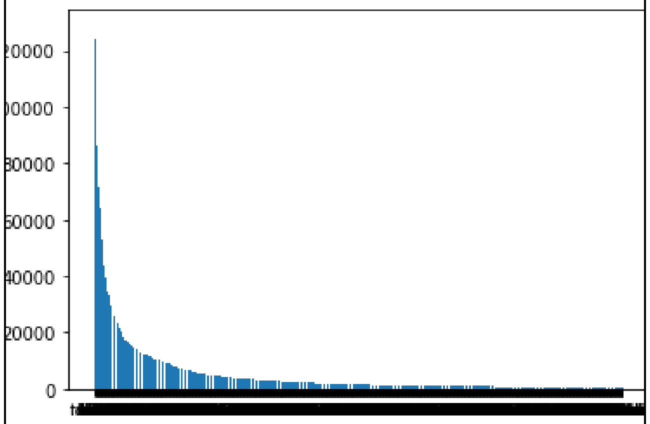


figure [3] - plot of words frequency

then, we searched for patterns between word’s usage and the overall emotions associated with tweet. and figure [4] has a plot of the obtained data:

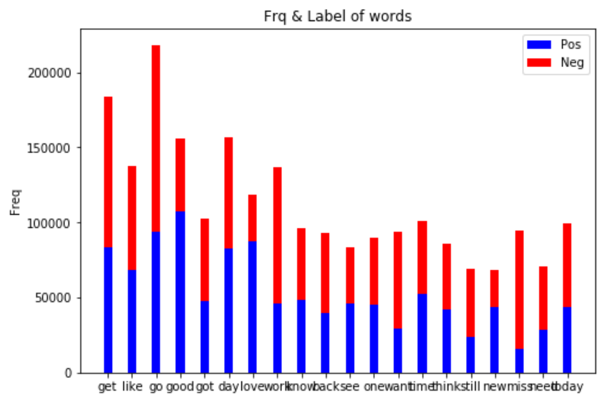


figure [4] - word and associated emotion

V. Feature Selection and Data Representation:

In our problem, features are basically words. In our dataset, we started with about 200k words. However, after doing the preprocessing we detailed above, we reached 13k words.

We tried two different representations for tweets: 1) bag-of-words 2) vectorized representation - doc2vec.

1. bag-of-words:

as stated in the book *Neural Network Methods in Natural Language Processing* [2]:

“A very common feature extraction procedure for sentences and documents is the bag-of-words approach (BOW). In this approach, we look at the histogram of the words within the text, i.e. considering each word count as a feature.”

in the BOW representation we encode tweets. However, this proved to be an extremely memory inefficient approach to encode tweets. That is, for each tweet of maximum size of 280 characters, we have a sparse vector representation of about 13k.

As would be detailed later, it was impossible to upload the matrix representation of our dataset into the memory at once because it needed a huge RAM to include (2.5M * ~13 K) table. so, we went with trying the models on a subset of the dataset and also tried partial fitting to train models with subsets of about 1000 instances at a time to be able to evaluate the BOW approach with the Doc2Vec approach that is discussed in the following subsection.

2. doc2vec

Doc2vec is just a generalization of word2vec in which we are embedding an entire array of strings into a vector represented from the average summation of the individual word2vec representation of each word.

We trained our word2vec [3] model using CBOW due to its fast performance and its ability to generalize compared to n-gram which is slower to train and is better with infrequent words.

Here is a diagram from the paper Distributed Representations of Sentences and Documents in which doc2vec was proposed.

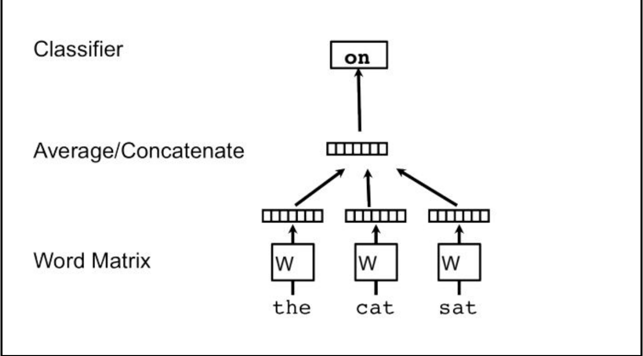
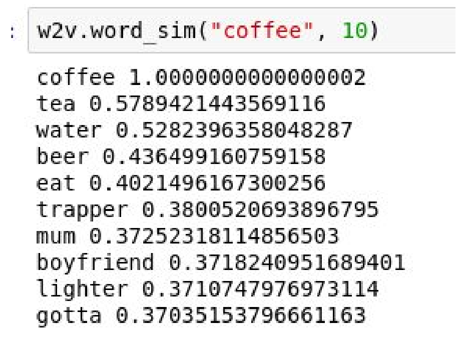


figure [5] form Distributed Representations of Sentences and Documents [7]

and here are some samples to demonstrated how the model captured the relations between words:



```
: w2v.word_sim("coffee", 10)

coffee 1.0000000000000002
tea 0.5789421443569116
water 0.5282396358048287
beer 0.436499160759158
eat 0.4021496167300256
trapper 0.3800520693896795
mum 0.37252318114856503
boyfriend 0.3718240951689401
lighter 0.3710747976973114
gotta 0.37035153796661163
```

figure[6] - similarity between word2vec representation after the first 2 iterations

This Dov2Vec approach transformed our dataset to have 100 features instead of 13K by assigning a 100 dimensional vector to each instance (tweet). This way, the Doc2Vec dataset is 2.64M by 100 Matrix, which is about a 4Gb CSV file.

VI. METHODOLOGY

As discussed in the previous section, we tried two forms of dataset representation to run different numbers of models against. The first one is BOW and the second one is Word2Vec. Therefore, we constructed 4 datasets, three of them are BOW datasets and one of them is a Doc2Vec dataset. The three BOW datasets were subsets of the original data because of the memory problem discussed before. So, we drew 3 different random samples totalling to 50% of the size of the original one. We also tried partial fitting (incremental learning) on one of these subsets to see the feasibility of applying it in case the BOW approach worked better than the Doc2Vec one. Long story short, we had 4 different datasets to try models on: (250K, 500K, 500K, Doc2Vec). The first three are drawn from the BOW approach; the second 500K sample was tried for incremental learning.

We tried the following sklearn models on these 4 datasets:

1. neural nets
2. perceptron
3. Logistic Regression
4. SVM
5. decision trees
6. naive bayes

After running each model we used a sklearn *classification report* to measure the performance of those models. This classification report, basically, returns a text summary of the precision, recall, F1 score for each class.

After evaluating these metrics for all models, we can choose which type of dataset to use and which model to implement from scratch to be able to tweek and enhance its performance.

In the following two sections, we present these results, the decision of which model to implement and the results of our model compared to the built-in onel.

V. *sklearn* Models Results :

In this section, we will show a detailed description of the results we got from running the sci-kit learn built in models on each of the 4 datasets. The tables below show each model with its performance and time taken on each of the 4 datasets.

1. Decision Tree (sklearn):

	250K	500K	Doc2Vec	Avgs
Accuracy %	52	53	53.4	53
Precision %	52.7	54	52.9	53
Recall %	52.5	53.5	53.5	53.25
Estimated Time (secs)	600	1000	1215	938

2. Random Forest (sklearn):

Number of trees =10

	Sample 250K	Sample 500K #1	Doc2Vec vectors	Avgs
Accuracy %	52.7	53.5	56	54
Precision %	52.6	52.8	54.5	53.3
Recall %	52.5	53	55.5	53.667
Estimated Time (secs)	620	1210	854	895

3. Model Logistic Regression (sklearn):

Max Number of iteration for GD=1000

	Sample 250K	Sample 500K #2	Doc2Vec vectors	Avgs
Accuracy %	53	72	63.5	60.55
Precision %	52.7	72.5	60	59.5
Recall %	52.5	70	65	60.375
Estimated Time (secs)	25	3200	47	36.75

4. Model MuliLayerPreceptron(SK-learn MLP):

num of hidden layers =2.

	Sample 250K	Sample 500K #2	Doc2Vec vectors	Avgs
Accuracy %	53	65	64.8	59.5
Precision %	53.1	65.5	60.4	58.3
Recall %	53.5	65	65	59.625
Estimated Time (secs)	600	10000	1000	1050

5. Naive Bayes (sklearn):

	Sample 250K	Sample 500K #2	Doc2Vec vectors	Avgs
Accuracy %	53.3	73	56.6	59
Precision %	52.8	72.5	54.7	58
Recall %	52.25	72.5	56.5	58.44
Estimated Time (secs)	15	2700	16.6	20

6. Linear SVM (sklearn):

	Sample 250K	Sample 500K #2	Doc2Vec vectors	Avgs
Accuracy %	53	72	57	59.25
Precision %	52.7	73	65	60.8
Recall %	52.25	72	56	58.56
Estimated Time (secs)	460	7000	1490	1187

Since our dataset is almost evenly divided between the two labels (48.7-51.3), we have chosen the accuracy to be the main determining factor. So, from the data collected above, we can see that this is the rank of models based on performance

1. Logistic regression
2. MLP
3. Linear Support Vector Machine
4. Naive Bayes
5. Random forest
6. Decision Tree

If we took a closer look at the results obtained from running the algorithms on different machines and with different samples of the same size, we can see that there is a notable difference in sample 500K #2 results and the other datasets. For example, using the Naive Bayes algorithm we had an accuracy of (.73) and of (.53). It is quite confusing to us why this happened. Here are some possible explanations:

1- This sample uses incremental learning, which might allow it not to overfit the data and to generalize better.

2- Maybe some documents are feature-rich while others are not. So, probably samples with more of those sample-rich documents were better at training.

3- As most of those models are based on approximation algorithms and iterative methods, we guess that probably while training some of those models reached a local minimum and got stuck there.

Note that: Almost the models in all samples converged to a solution, which means that their cost function was minimized at this accuracy, leading us to conclude that running more iterations will not improve much.

Model Selection:

As we can see from the tables above, there is a significant improvement when using doc2vec to represent each tweet. Clearly, this is because doc2vec tries to capture some of the similarities between documents of the same or closer meaning.

Looking at the time taken by each method, it's noted that Logistic regression and NB take much less time than the other models, and the trade off in the accuracy is not much, and they all converged so even if linear SVM and MLP are close in accuracy, but there are way behind logistic regression in time taken to train the model.

So, we decided to use doc2vec combined with a logistic regression model which we believe would produce the best possible results. as detailed in figure [7]

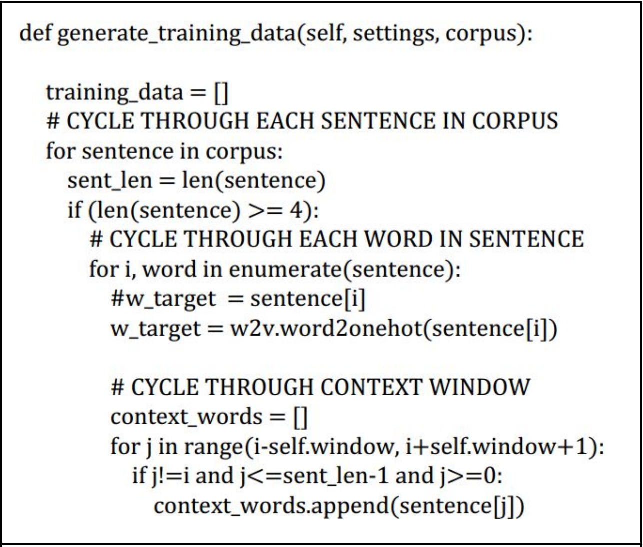
VI. Final Model

We discussed in the previous section why we chose the logistic regression model to implement from scratch and why we chose the Doc2Vec representation of the data. In this section, we explain the details of our own Doc2Vec model and our own logistic regression model, and then we elaborate on the results we got from running this implementation on the dataset. We divided the data into a 75% training set (around 1.75M instances) and 25% testing (~ 660K instances).

1. **Dov2Vec Implementation:**

Basically, we created a class word2vec [3], so we would be able to add some utility
functions over the vanilla implementation of neural networks. In our implementation, we have decided to go with CBOW instead of skip-gram
because CBOW is faster to train and gives better accuracy for the frequent words. On the other hand, skip-gram turned out to be orders of magnitude slower since it propagates the error of predicting n (in our case, 4) words instead of 1 in the case of CBOW.
As known, training word2vec required the same process that training neural nets
need. Further, it required one more utility function to pre-process each word in each tweet into a target-vector and a context vector.

We then began to extract the training words and process them using the function in figure 7.



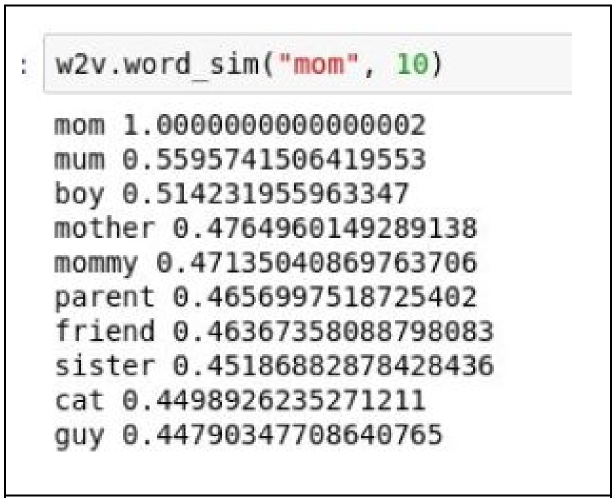
```
def generate_training_data(self, settings, corpus):

    training_data = []
    # CYCLE THROUGH EACH SENTENCE IN CORPUS
    for sentence in corpus:
        sent_len = len(sentence)
        if (len(sentence) >= 4):
            # CYCLE THROUGH EACH WORD IN SENTENCE
            for i, word in enumerate(sentence):
                #w_target  = sentence[i]
                w_target = w2v.word2onehot(sentence[i])

                # CYCLE THROUGH CONTEXT WINDOW
                context_words = []
                for j in range(i-self.window, i+self.window+1):
                    if j!=i and j<=sent_len-1 and j>=0:
                        context_words.append(sentence[j])
```

figure[7] - Extracting training words.

Figure 8 shows some similarities that our model captured between the word mom and other words from the context of our tweets.

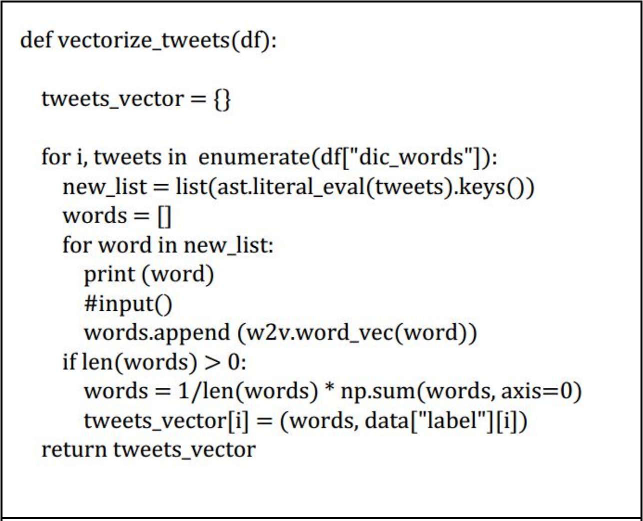


```
: w2v.word_sim("mom", 10)

mom 1.0000000000000002
mum 0.5595741506419553
boy 0.514231955963347
mother 0.4764960149289138
mommy 0.47135040869763706
parent 0.4656997518725402
friend 0.46367358088798083
sister 0.45186882878428436
cat 0.4498926235271211
guy 0.44790347708640765
```

figure[8] - The most similar words to mom from our Doc2Vec model.

After producing the vector representation of each word, we iterated over the tokenized dataset and calculated the averaged sum of each vector representation of each single word.
This produces a 100-D vector per tweet which is used to create the dataset features that we will feed into our model. Figure 9 shows the function used to get the vector for each tweet.



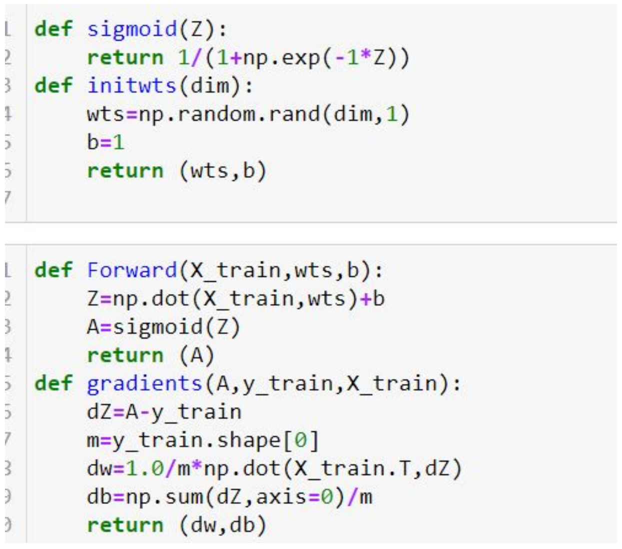
```
def vectorize_tweets(df):

    tweets_vector = {}

    for i, tweets in  enumerate(df["dic_words"]):
        new_list = list(ast.literal_eval(tweets).keys())
        words = []
        for word in new_list:
            print (word)
            #input()
            words.append (w2v.word_vec(word))
        if len(words) > 0:
            words = 1/len(words) * np.sum(words, axis=0)
            tweets_vector[i] = (words, data["label"][i])
    return tweets_vector
```

figure[9] - Vectorize tweets function.

Now, we show our logistic regression implementation. We tried to implement a vectorized version of the logistic regression to make it fast and reliable. . The idea is simply to process all the elements at once as
an array/ vector instead of looping over it; it is allowed by numpy arrays. Since there is one to one mapping between logistic regression and neural networks, we implemented simple feed forward backpropagation functions for the logistic regression. Figure 10 shows the core of this vectorized version. It simply calculated the sigmoid of the input in the forward function, and calculated the errors with the derivatives of the cost function with respect to the weights and the bias in the gradients function.



```python
def sigmoid(Z):
    return 1/(1+np.exp(-1*Z))
def initwts(dim):
    wts=np.random.rand(dim,1)
    b=1
    return (wts,b)
```

```python
def Forward(X_train,wts,b):
    Z=np.dot(X_train,wts)+b
    A=sigmoid(Z)
    return (A)
def gradients(A,y_train,X_train):
    dZ=A-y_train
    m=y_train.shape[0]
    dw=1.0/m*np.dot(X_train.T,dZ)
    db=np.sum(dZ,axis=0)/m
    return (dw,db)
```

figure[10] - Vectorize tweets function.

At first, the model used Batch gradient descent where it would use all the training set first then update the weights; it got accuracy slightly over 52%, which obviously needed to be optimized. It also took a lot of time- about an hour to complete 1000 iterations.

We applied the following modifications to try and optimize were done on the batch Logistic Reg. Note that every one of these modifications was tried individually to measure its effect on the model. Then they were combined together, which achieved the best result.

1. Normalizing the features: it was noted that some features are very small, so after many iterations, the model might not learn from them/ underflow can happen. Thus, all features were normalized to be between 0,1.

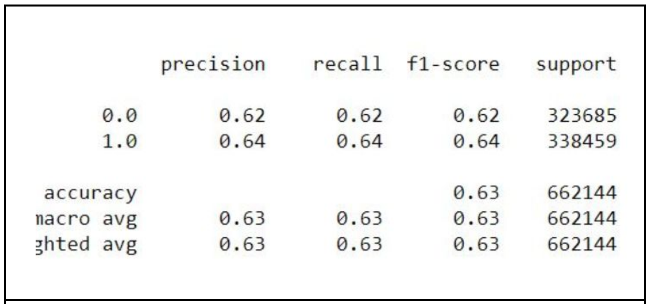
2. Varying the Learning rate: to avoid being stuck at a local minima, we tried varying the learning rate, where the model starts with a very high learning rate to keep oscillating, escaping local minimas. Then, it would decrease over time to stabilize the model.

3. Implementing Stochastic Gradient descent: The SGD takes a random number of instances and uses them to run one weight update. For example, it would take a random 10% of the training set, run a weight update and then in the next iteration picks up another random 10%, etc. Obviously, it runs much faster than Batch Gradient descent (5-10 mins)for 1000 iterations. Several random percentages taken from the training set were tried (20-10 -5 -1). The best was found to be % of the training sample (about 20K instances).

4. Using Stochastic Gradient descent First then Batch GD: Applying this method (many iterations of SGD then few iterations of BGD) is both fast and accurate. It makes the model achieves the optimum weights faster, then using BGD it makes sure to be at the global minima. Two functions were made to run after each other (Train_SGD, Train_BGD).

5. Using K-fold Cross Validation: This was done to avoid overfitting; 10-fold validation was used, and the weights of each running was saved. Then, the average weight of all the 10 was obtained. This was found to be the most stable set of weights.

Below, in fig. 11, we show the classification report of running the Kfold approach on the test set. The accuracy given is similar to that of the best model in section V, but we will also compare it to the SKlearn Log-Reg on this dataset.



```
              precision    recall  f1-score   support

         0.0       0.62      0.62      0.62    323685
         1.0       0.64      0.64      0.64    338459

    accuracy                           0.63    662144
   nacro avg       0.63      0.63      0.63    662144
   ghted avg       0.63      0.63      0.63    662144
```

figure[11] - Kfold Results.

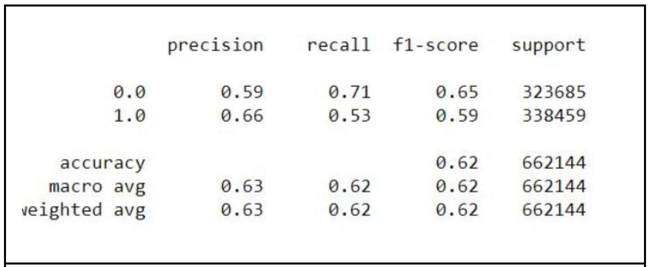
To make sure that this is the best the model we can get from the corresponding
dataset (the loss can’t be decreased more), we tried several tests:

1. Running a very high number of iterations in both SGD and BGD: This was we
can figure out if the loss can get any smaller, which wasn’t the case.

2. Trying different initializations of the weights: was done to see if the weights
initial values affected it. We tried (initializing to zero, several constant values
for all, random value between [0,1], normal random value with). Again, they
almost achieved the same

3. Tried different dataset: tried 1M dataset with embedding vectors created
from gensim library (as in phase 3). The accuracy achieved was 69%, which
means that generalizing over 2.5x larger dataset as ours with 63% accuracy
is probably the best the model can get.

4. Tried a SciKit Learn Logistic Regression: We tried running the training with
sklearn to compare; the sklearn model got slightly less accuracy.Figure 12 shows the performance of sklearn model.



```
              precision    recall  f1-score   support

         0.0       0.59      0.71      0.65    323685
         1.0       0.66      0.53      0.59    338459

    accuracy                           0.62    662144
   macro avg       0.63      0.62      0.62    662144
veighted avg       0.63      0.62      0.62    662144
```

figure[12] - Sklearn Results

Getting a performance slightly higher than the sklearn one was our final proof to be convinced that this is the best the model can get; we discuss the limitations that we think caused that in an upcoming section.

The next step is to build a utility application for the user to input a tweet and use the model we trained to predict his mood. The summary of the steps that were done in this section and the previous one is shown in figure 13. In the upcoming section, we explain the process of building the server application.

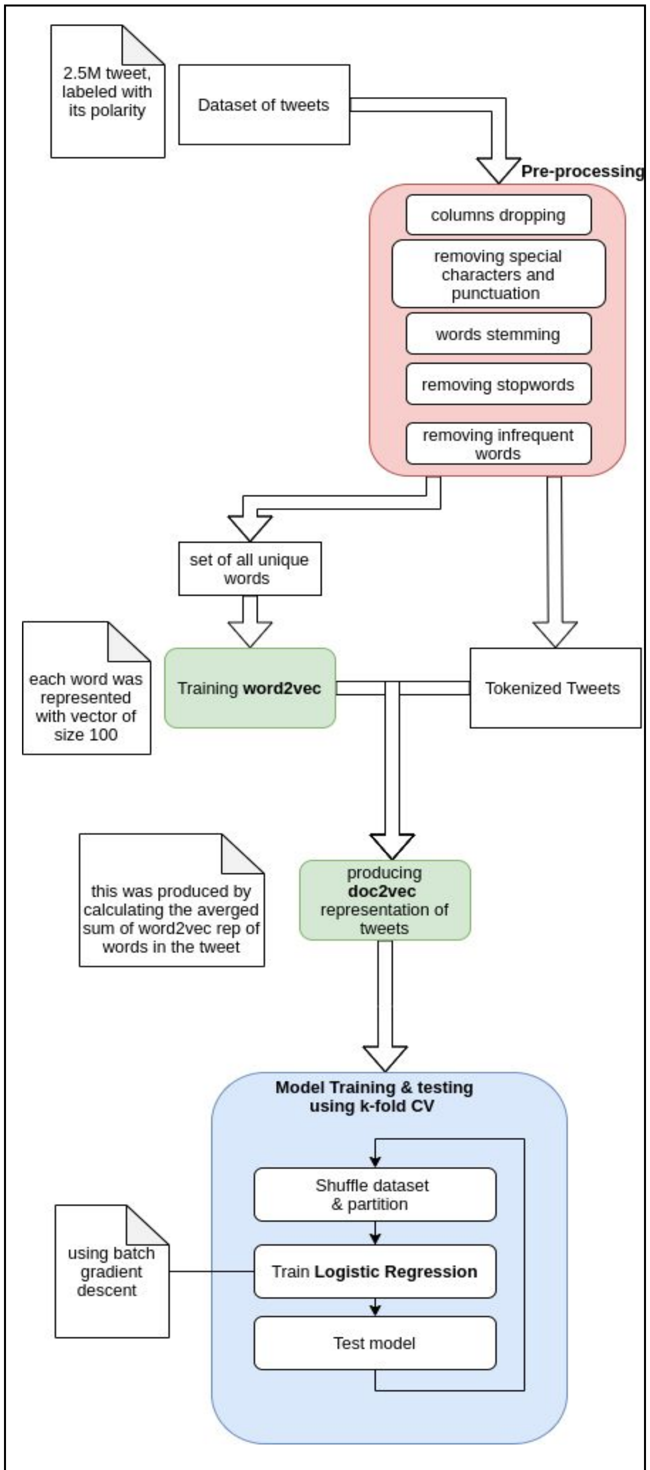
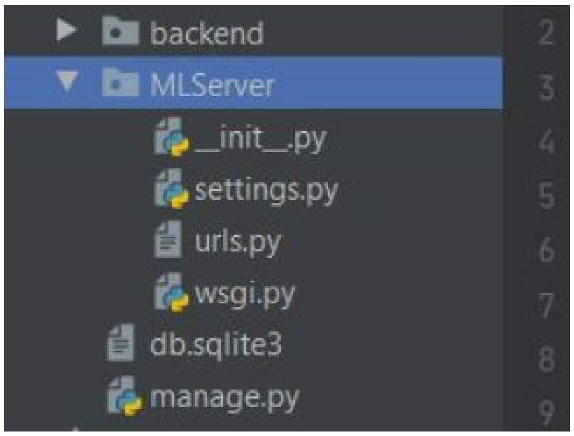

2.5M tweet, labeled with its polarity
Dataset of tweets
Pre-processing
columns dropping
removing special characters and punctuation
words stemming
removing stopwords
removing infrequent words
set of all unique words
Tokenized Tweets
each word was represented with vector of size 100
Training word2vec
this was produced by calculating the averged sum of word2vec rep of words in the tweet
producing doc2vec representation of tweets
Model Training & testing using k-fold CV
Shuffle dataset & partition
using batch gradient descent
Train Logistic Regression
Test model

figure [13] - Architecture of Final Model

VII. Utility Application.

We implemented our interface using Django, which is a python based tool [6]. The hierarchy of its project is divided into a server (base) and a number of applications (i.e. parts of the website). The server itself contains the settings, the database, and the sub urls that the website has and where each one of them directs the user.

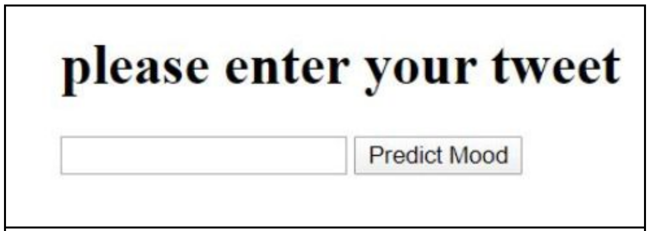
In the project folders, you will find an application called backend that contains the backend and an html form code to be able to take input and process them.


```
▶ backend
▼ MLServer
    __init__.py
    settings.py
    urls.py
    wsgi.py
  db.sqlite3
  manage.py
```

figure[14] - ML Server Architecture

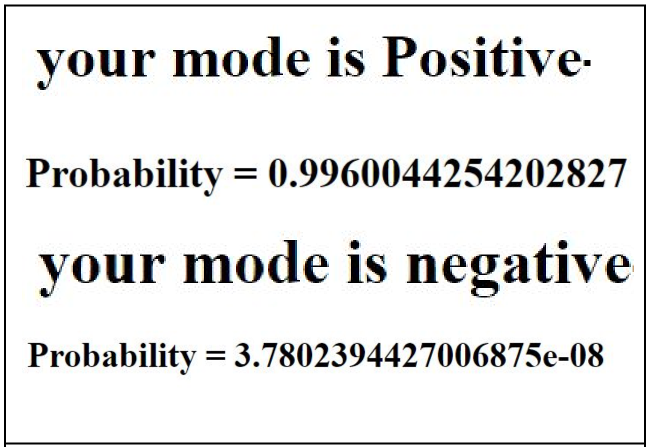
Figure 14 shows this hierarchy. The apps.py file is responsible for the details of the backend (the functions mentioned above). It reads the CSV files found in (CSV folder), pre-process, and predict.The views.py file is responsible for communication with the server, and returning the output to be displayed in the page (application) .

Then one can access a minimalist HTML form asking the user to enter the tweet. And after he clicks on the "Predict Mood" button, it sends a GET request. This GET request moves to "classify" subdomain in which the tweet is processed and the result is displayed on the screen. Figure 15 shows that form.



figure[15] - Application Interface.

The output will be the predicted mood with the probability output from the sigmoid function as shown in figure 16 that displays the result of two runnings of the application, one with the sentence "I love the weather today", and the other with the sentence "I miss My friends so much." The output was that it is positive in the first one with a very high probability and negative in the second one with a very low probability from the sigmoid function (extremely negative class).



```
your mode is Positive.

Probability = 0.9960044254202827

your mode is negative

Probability = 3.7802394427006875e-08
```

figure[15] - Runs from the program.

VIII. Discussion

The model we implemented turned out to be slightly better than the one with SKlearn. In this section, we will try to explain why the accuracy of the model is not high and what the limitations of the model are. Also, we will compare it to another work done at the university of Columbia.

First, we will discuss the main points of strength and weaknesses of our model to be able to compare it better.

Strengths:

- The model runs fast relative to other models (ANN) and almost as fast as the sklearn model.
- Using Gradient Descent ensures that the model gets close to the best possible from the dataset.
- The implementation is simple and can be updated easily (no huge number of parameters to save).
- Can handle pretty large datasets without causing memory/overflow issues.
- Easy to trace the errors since it's not complex like the ANN.
- Can be increased in complexity to handle many features (by adding layers to it only if needed).

Weaknesses:

- The accuracy is not great, but we believe that this is almost the best that can be done from the dataset based on what was discussed above and what we discussed in the previous phase.
- It is sometimes dependent on the weights initialization if no sufficient number of iterations was guaranteed.

Now, we discuss the reasons why we think the model performance is quite low; i.e. the limitations and the things that the model cannot handle.

Limitations:

1. The program is not likely to detect the negation of the statement (i.e. it treats “I love school” and “I don’t love school” the same way). This is because not and its variations are not all included in the embedding vectors (some of them are), and if they are found, their vector when added to a strong positive word such as “love” still doesn’t affect the outcome much.

2. When sentences are long, the addition of relatively neutral vectors (e.g. sell, buy, walk, home, ...etc) can affect the tweet vector and shift it to a wrong direction (i.e. writing a long positive sentence with many words might make the application think of it as negative because of the added weights of the other neutral words). This is a problem with the embedding in general; however, it happened with our experimentation a few times.

3. The dictionary is quite limited and does not contain a lot of words (only about 20K words), so some texts might go without finding its words in the dictionary, which might produce wrong results.

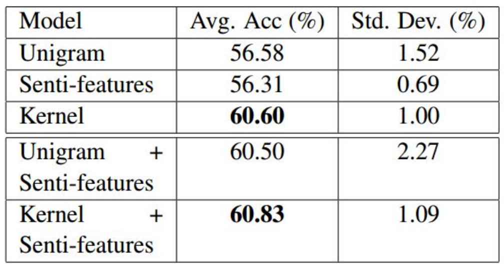
4. The vectors of the individual word are built from a limited context, so its vector might be misleading or biased towards a certain polarity despite the word being a very neutral word.

We tried to solve all of these problems, especially the second one because it can affect the performance greatly. We tried to fix it by trial and error. Basically, we tried to find the range of the neutral words (i.e. the return that the sigmoid predicts for the neutral words). Since the overall probability is nothing but an accumulation of the probabilities of each individual vector, removing them should make the model invariant with them. We did many iterations to find out their range; the model performance was raised a bit, but it wasn't completely independent from the neutral words since we can’t exactly know the range of all the words.

A final note is that one of the main reasons of this accuracy was the data processing step; we used the natural language processing toolkit (nltk) to remove stop words and to stem the words; it turned out that their stemmer and lemmatizer are really bad, and they resulted in a quite messy dictionary, which made the capturing of the real features harder for the model.

We will now compare our model to a work done by the team at the university of Columbia; they worked on twitter sentiment analysis as well except they used Support vector machines with kernels not logistic regression. The difference is that they were expecting three modes (positive, negative and neutral) instead of positive or negative as in our case [8]. They used an 11K dataset, which is orders of magnitude less than ours; and it was collected from one source not from two different sources with two different time frames like in our case.

As shown in figure 16 [8], the performance of their models accuracy were around 60% as well, despite the fact that the data is small, which makes the detection of patterns in it an easier task. Anyways, it seems that the problem of sentiment analysis can be expected to produce high accuracy models because it still has many challenges unlike the pictures task.



Model	Avg. Acc (%)	Std. Dev. (%)
Unigram	56.58	1.52
Senti-features	56.31	0.69
Kernel	**60.60**	1.00
Unigram + Senti-features	60.50	2.27
Kernel + Senti-features	**60.83**	1.09

figure[16] - Columbia team results.

IX. Conclusion

We have tried in this paper to explain our approach to the twitter sentiment analysis problem. We have gone through the standard steps of data processing, feature extraction, trying different models, tweaking the highest performance model, and building a user application for real life use. We have explored in the way some of the challenges the nature of the problem imposes and the challenges that our model faces. We have demonstrated that the main two problems that caused our model to converge at a quite low accuracy are the processing problem of the stemmer, and the nature of the Doc2Vec approach. Finally, we explored another team's solution to get a sense of the performance of these kinds of models.

X. Future Work

We recommend anyone working on the same model to avoid the challenges our model faced. First, try a better stemmer and lemmatizer to make the data features and dictionary as unique and representatives as possible. Second, work on a way to eliminate the effect of the neutral word vectors on the overall model prediction prediction.

XI. REFERENCES

[1] Z. Jianqiang and G. Xiaolin, "Comparison Research on Text Pre-processing Methods on Twitter Sentiment Analysis," in *IEEE Access*, vol. 5, pp. 2870-2879, 2017, doi: 10.1109/ACCESS.2017.2672677.

[2] Yoav Goldberg. 2017. Neural network methods in natural language processing, Morgan & Claypool Publishers.

[3]Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In Proceedings of Workshop at ICLR, 2013.

[4] Anuj Sharma and Shubhamoy Dey. 2012. A comparative study of feature selection and machine learning techniques for sentiment analysis. Proceedings of the 2012 ACM Research in Applied Computation Symposium on - RACS 12 (2012). DOI:http://dx.doi.org/10.1145/2401603.2401605

[5] Bo Pang and Lillian Lee. 2008. Opinion Mining and Sentiment Analysis. (2008). DOI:http://dx.doi.org/10.1561/9781601981516

[6] Piotr Płoński. Deploy Machine Learning Models with Django. Retrieved May 15, 2020 from https://www.deploymachinelearning.com/

[7] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML'14). JMLR.org, II–1188–II–1196.

[8] Agarwal, Apoorv & Xie, Boyi & Vovsha, Ilia & Rambow, Owen & Passonneau, Rebecca. (2011). Sentiment Analysis of Twitter Data. Proceedings of the Workshop on Languages in Social Media.