Sentiment Analysis on Tweets
Pre-processing and Feature Generation

Amer Elsheikh
The American University in Cairo
Cairo, Egypt
amer.elsheikh@aucegypt.edu
900196010

Abdallah Abdelaziz
The American University in Cairo
Cairo, Egypt
Abdallah_taha@aucegypt.edu
900196083

1. The Dataset: Sentiment 140

After investigating different datasets, we decided to use the Sentiment140 dataset. The main reason for choosing this dataset is its size which makes it suitable for machine learning applications. Having 1.6M instances is suitable for the complex task of predicting the sentiment analysis of a tweet with reasonable accuarcy. In addition, the distribution of positive and negative tweets is uniform with 800K positive tweets and 800K negative tweets. Finally, the dataset is suitable for the purpose of the project which is sentiment analysis but specific to tweets. The dataset will be representative of problems encountered when analyzing informal writing such as the one used in writing tweets.

The dataset created by Alec Go, Richa Bhayani, and Lei Huang, who were Computer Science graduate students at Stanford University, and it was collected using twitter API. The sentiment of tweets was decided using emoticons such as :-). Tweets with positive emoticons were labeled positive and tweets with negative emotions were labeled negative. The method by which the tweets were labeled gives this dataset advantage over datasets of similar size. This is because datasets of such big size may depend on methods such as lexicon based which may not be as accurate as using the emoticons.

2. Feature Analysis

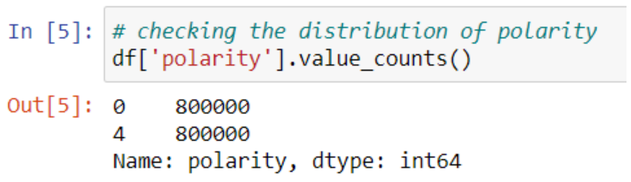
The data set is available in a csv file with the following features:

1. The polarity of the tweet: 0 for negative tweets, 2 for neutral tweets and 4 for positive tweets.
2. The id of the tweet: the id associated with the tweet obtained from twitter API; an example is 1467810369.
3. The date of the tweet: the date is a string formatted as (weekDay month day hh:mm:ss UTC yyyy). One example is (Sat May 16 23:58:44 UTC 2009).
4. Query: if there is no query, the value is NO_QUERY.
5. Username: the user who tweeted such as "scotthamilton".
6. Text: the text of the tweet.

Now, a deeper analysis of the features will be given:

Polarity

Tweet polarity is the label and the feature of interest for this project. In the dataset, this feature has **no null**



```
In [5]: # checking the distribution of polarity
        df['polarity'].value_counts()

Out[5]: 0    800000
        4    800000
        Name: polarity, dtype: int64
```

values. Moreover, the feature is distributed uniformly with 800K positive instances and 800K negative instances.

Moreover, we found no neutral tweets which mea ns there is no single tweet with polarity 2. That serves the purpose of the projects as we only classify positive and negative tweets.

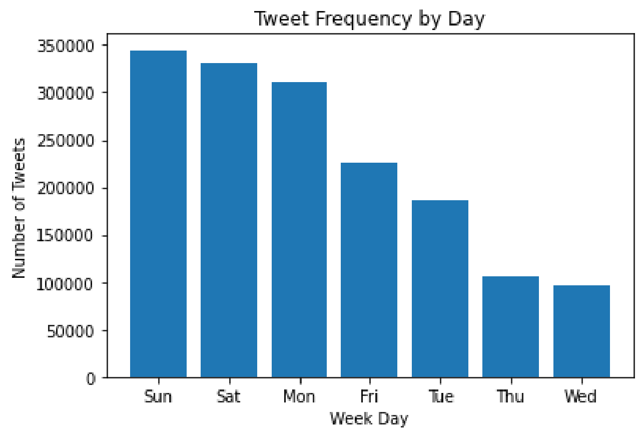
Tweet ID

The second feature is the tweet id. Every tweet has a unique id. However, in the dataset, for some tweets, a single tweet was assigned different polarities; hence, it appeared in the dataset twice. When investigating such instances, they seem to have two different sentiments. For example, one instance was "Haven't tweeted nearly all day Posted my website tonight, hopefully that goes well Night time!" which probably contained emoticons of different emotions. The number of such duplicates is 1685 which is a very small fraction of the dataset. Like the polarity, tweet ID has no missing values.

Date

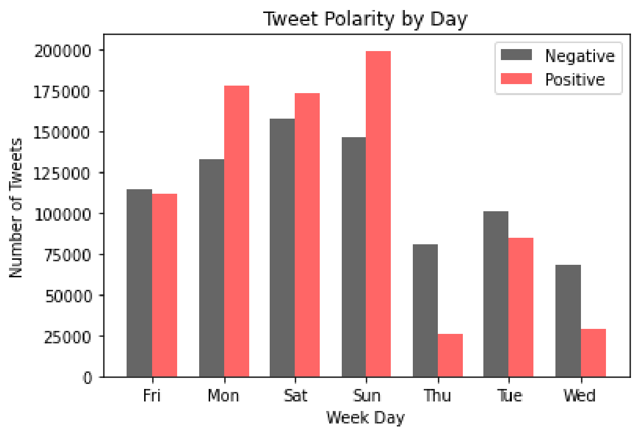
Every tweet has a timestamp which contains the weekday, month, day in month, time at which the tweet was posted, time zone and the year in which the tweet was made. To understand this feature, every sub-feature was analyzed individually.

To begin with, distribution of tweets per weekday wasn't exactly uniform. Days such as Sunday and Saturday had more tweets than days such as Wednesday and Thursday. The distribution is shown in the figure 1 below.



This makes sense as people have more time to spend on social media in weekends. Next, we see if there is a correlation between weekday and the sentiment of the tweet. To see this, observe the bar graph in figure 2.

From the graph, we can see that sentiment is not



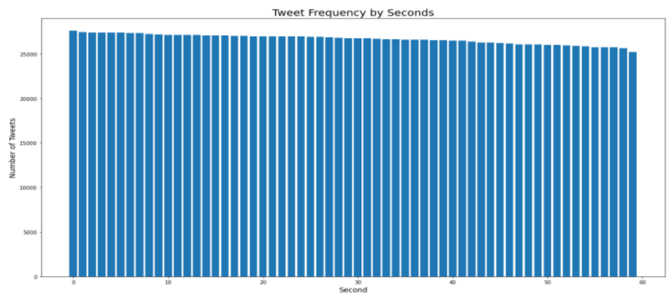
independent from weekday. Some days have more positive tweets such as Sunday which some other tweets have more negative tweets such as Thursday. So, the day of week seem to have some correlation with the sentiment of the tweet.

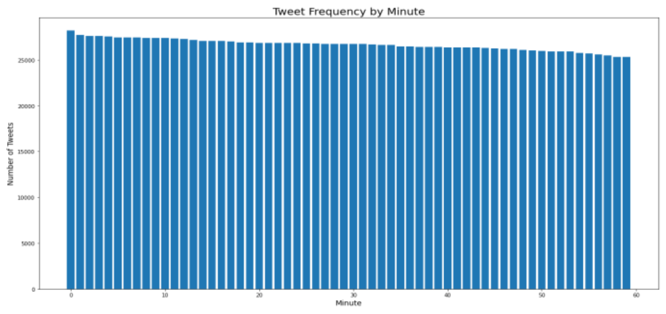
Next, we consider the month in which the tweet was posted. It seems that all the records came only from three months which are April, May and June. Because of incompleteness of this feature is does not seem to be of any value for our purpose.

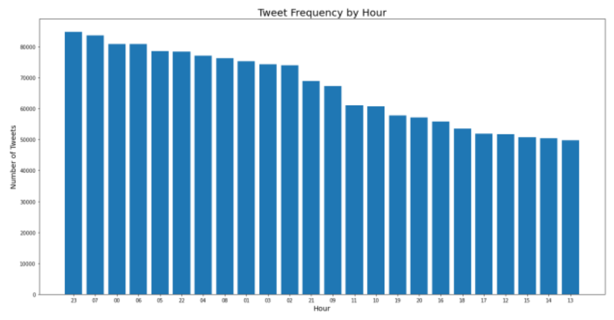
```
# distribution of tweets in months
df['month'].value_counts()
```

```
Jun     923608
May     576367
Apr     100025
Name: month, dtype: int64
```

Now for the exact time, it seems that tweets are distributed uniformly in terms of frequency among, minutes and seconds but not among hours as shown in the three figures below.

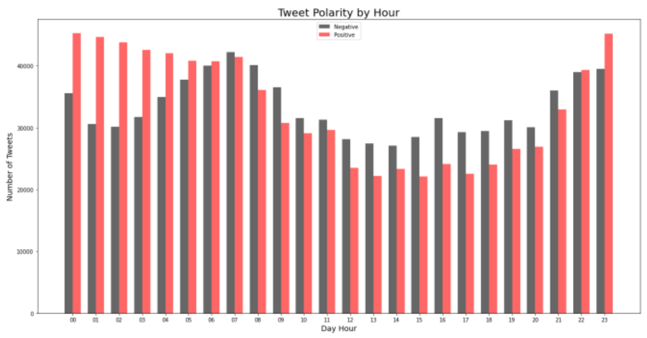






This as well makes sense as people tend to post during certain hours of the day but not during certain minutes of an hour or seconds of a minute.

To continue the analysis, it was found that there is no correlation between polarity and minutes or polarity and seconds. Different minutes or seconds had slightly different frequencies of positive and negative tweets. For hours however, some pattern was observed as shown in the bar chart below.



This implies there is a correlation between the hour at which the tweet was posted and its sentiment. As all other features, no null values exist.

Query

All data instances have the same value for this feature which is "NO_QUERY". There are no null values for this feature.

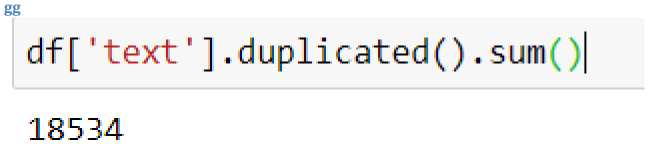
Username

Every tweet has a username which refers to the person who wrote the tweet. In the dataset multiple tweets share the same username. In fact, there is only 940225 unique usernames in the dataset. However, since our goal is to detect sentiment of speech, this feature doesn't seem to add information useful to our purpose.

Text

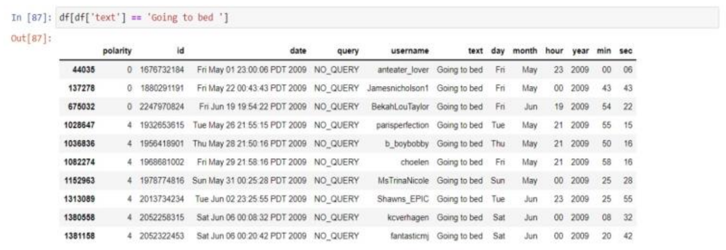
Text is the actual text of the tweet and is the hardest to analyze due to its unstructured format. To analyze the text, we need first to pre-process it in order to get useful insights. However, we found out that some tweets where duplicated. Even more, those duplicated texts can sometimes have different sentiments. The number of such tweets is 18534 which is a small number compared to the size of the dataset.

gg



```
df['text'].duplicated().sum()
```

```
18534
```



```
In [87]: df[df['text'] == 'Going to bed ']
Out[87]:
         polarity          id                          date     query         username         text  day  month  hour  year  min  sec
  44035         0  1676732184  Fri May 01 23:00:06 PDT 2009  NO_QUERY     anteater_lover  Going to bed  Fri    May    23  2009   00   06
 137278         0  1880291191  Fri May 22 00:43:43 PDT 2009  NO_QUERY  Jamesnicholson1  Going to bed  Fri    May    00  2009   43   43
 675032         0  2247970824  Fri Jun 19 19:54:22 PDT 2009  NO_QUERY    BekahLouTaylor  Going to bed  Fri    Jun    19  2009   54   22
1028647         4  1932653615  Tue May 26 21:55:15 PDT 2009  NO_QUERY    parisperfection  Going to bed  Tue    May    21  2009   55   15
1036836         4  1956418901  Thu May 28 21:50:16 PDT 2009  NO_QUERY        b_boybobby  Going to bed  Thu    May    21  2009   50   16
1082274         4  1968681002  Fri May 29 21:58:16 PDT 2009  NO_QUERY           choelen  Going to bed  Fri    May    21  2009   58   16
1152963         4  1978774816  Sun May 31 00:25:28 PDT 2009  NO_QUERY      MsTrinaNicole  Going to bed  Sun    May    00  2009   25   28
1313089         4  2013734234  Tue Jun 02 23:25:55 PDT 2009  NO_QUERY        Shawns_EPIC  Going to bed  Tue    Jun    23  2009   25   55
1380558         4  2052258315  Sat Jun 06 00:08:32 PDT 2009  NO_QUERY         kcverhagen  Going to bed  Sat    Jun    00  2009   08   32
1381158         4  2052322453  Sat Jun 06 00:20:42 PDT 2009  NO_QUERY        fantasticmj  Going to bed  Sat    Jun    00  2009   20   42
```

Duplicate text with different sentiments

3. Cleaning and Pre-processing

Removing Unnecessary Features and Data

We begin by dropping the duplicates which happened in two cases:

- Tweets with the same tweet id as we discovered they sometimes have the same label.
- Tweets with the same text as they also have sometimes the same label.

We dropped both of those to remove some noisiness, and they were more than 18000 instances

We, then, drop the unneeded features which do not add any value to our analysis represented in:

- Tweet ID
- User Name
- Tweet Date: we drop it after extracting the hour and day of the tweet from it. Other information is not useful. To be specific, the month is not useful as there were only 3 months. Also, there was only one year. Moreover, we showed the seconds and minutes are irrelevant. Out of date, we only have day and hour remaining.
- Query: all values were "No_QUERY"

Hence, the features remaining are text, day, and hour.

Adjusting Polarity

In the dataset, the polarity was either 0 for negative or 4 for positive. However, we decided to map every positive polarity to 1 in order to work with number 0 and 1 only for convenience.

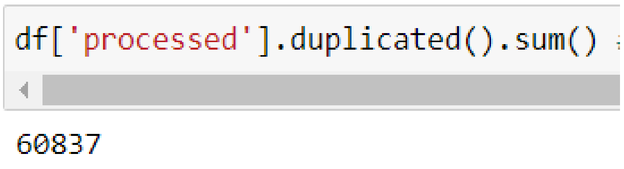
Text Preprocessing

Initial Processing

It remains to handle the text. To begin with, we followed these steps:

- All tabs and newlines were replaced with space characters.
- All mentions, hashtags, links, and special characters were totally removed as they are rather noisy and do not add value.
- Every accented letter was converted to the most similar ascii letter as the accented letters are hard to process.
- Contractions were replaced by their expansions. For example, “don’t” was replaced by “do not”. That will make us same the same words alike. Moreover, it will help a lot in handling negation using not in the next steps.
- All punctuation marks, special characters, digits, or any character other than the alphabet were totally removed as they would not add value to the sentiment as well.
- Multiple spaces were replaced by one space to reduce the characters used
- Lastly, all letters were lowered to avoid understanding the same word differently and to avoid repetitive words.

After the preprocessing, more than 60 thousand tweets with the same text appeared, so we dropped them as well specially to reduce time needed for Lemmatizing, the next step.



```
df['processed'].duplicated().sum()
```

```
60837
```

Tokenizing

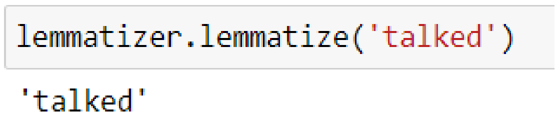
Following the initial preprocessing, tweet texts where split using space characters as delimiters to help us process and work on each word alone. The distribution of the words after this step is shown in the figure below.

It is clear from the figure that the most frequent words are stop words as for example the words “i, to, the, a, is, and you” are the most frequent words, so we need to handle those as follows.

Lemmatization

We begin by using a lemmatizer, specifically the Wordnet lemmatizer, to stem every word to its original form.

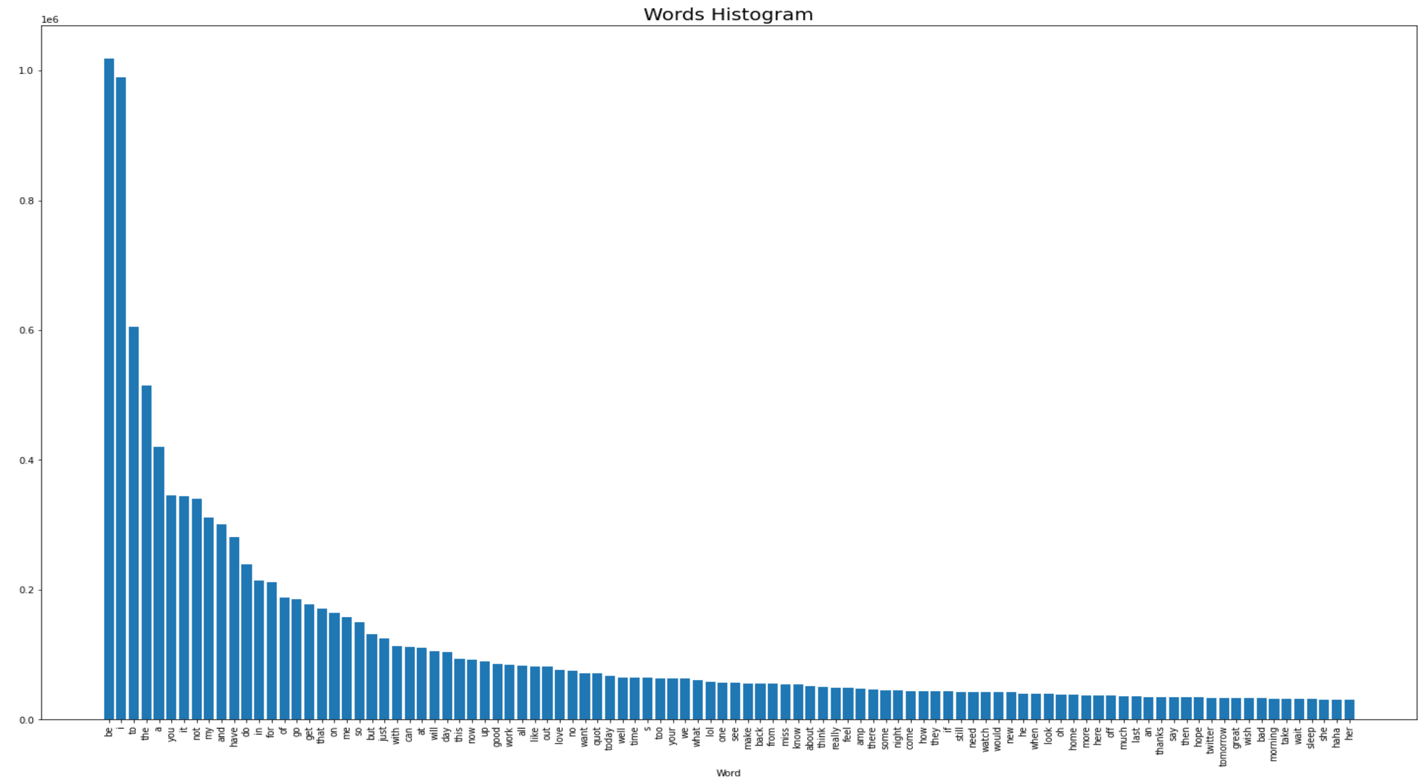
However, the lemmatizer does not always give the correct origin of the word. For example as shows, it stems the word ‘talked’ to itself:

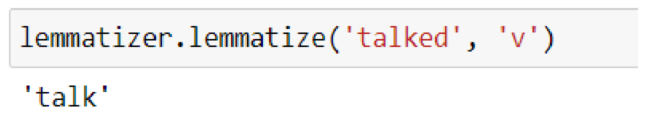


```
lemmatizer.lemmatize('talked')
```

```
'talked'
```

To resolve this, we provide the part of speech of each word to the lemmatizer so that it can do the conversion in a more accurate way. For example, if we specified ‘talked’ was a very, we would have got correct result as shown:





```
lemmatizer.lemmatize('talked', 'v')
```

```
'talk'
```

To get the part of speech of each word, we will use the pos_tag library of Natural Language Toolkit, nltk. The pos will be given as an argument to the lemmatizer to do its job.

Stop Words

Now, we used nltk's stopwords dictionary to remove the stop words from every tweet. However, we excluded two specific stop words which are not and no, which will be handled in the next step, because they can totally change the sentiment of the tweet, so we should not remove them. A clear example would be "I am not happy." If we removed not, then the tweet would be of positive tone although it had essentially negative tone.

Handling Negation

We will now handle the appearance of the three most common negating words in a sentence which are not, not, and never. Each of these words can transform the meaning of the sentence. To account for that, the idea of bigrams will be used.

Whenever we meet one of the three negating words, we will remove them from the tokenized list of words along with the next non-negating word. Instead of all of that, we will put 'not' concatenated with that non-negating word in the list. For example, if our list is ['I', 'am', 'never', 'sad'] will be transformed into ['I', 'am', 'notsad']. Another example would be ['no', 'not', 'cool''] which will be transformed into ['notcool'].

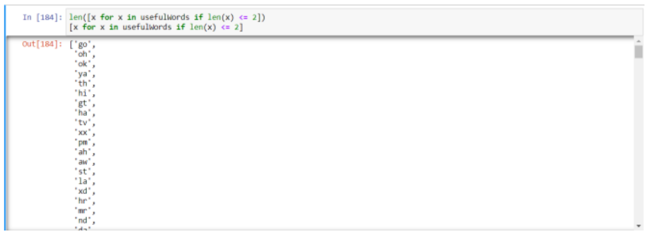
Notice that we did not leave space in the bigram as this is more convenient in the coding as we sometimes need to join the list into one sentence, and we do not wan to expand not again. Also, we tried to map the word into its antonym, but antonyms dictionaries existing were rather inaccurate and took a lot of time. That is why we decided to use bigrams in order not to lose the tone.

Frequency Analysis After Removing Stop Words

After removing the stop words and handling negation, words number has decreased substantially, and the distribution of words became as in the bar chart at the end of the page.

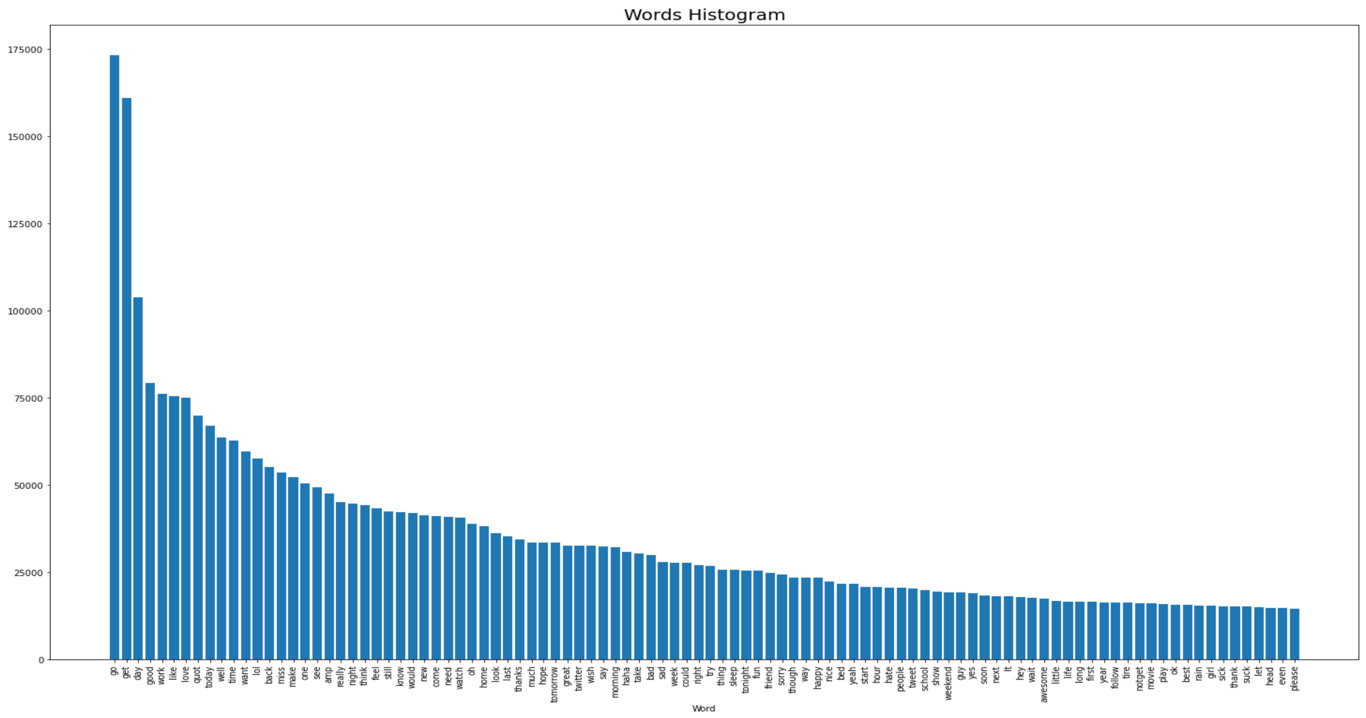
From the chart, it is clear there seem to be more stop words like "go", "get", and "day". These words along with any word consisting of two letters or less (except important words like "hi" and "ok") are farther removed from the data.

More clearly, the list of most frequent words, words with frequency >=100, contains a lot of meaningless 2 lettered words as shown in the figure below



```
In [184]: len([x for x in usefulWords if len(x) <= 2])
          [x for x in usefulWords if len(x) <= 2]
Out[184]: ['go',
           'oh',
           'ok',
           'ya',
           'th',
           'hi',
           'gt',
           'ha',
           'tv',
           'xx',
           'pm',
           'ah',
           'aw',
           'st',
           'la',
           'xd',
           'hr',
           'mr',
           'nd',
```

As said, we will remove them with the stopwords. After removing these, our dictionary had around 260K words. Now, notice that we decided not to drop words with highest frequencies empirically (for example highest 5%) as there are important words for sentiment occurring in a high frequency such as the word "happy." However, it is worth mentioning that majority of most frequent words were stop words, and we already got rid of them.



Removing less frequent words

Now, our dictionary contains around 260K words as stated. However, the number of words with frequency greater than 100 is only 7K. That makes sense as there are a lot of non-frequent words which is clear in the median and mean of the frequency graph of words:

```
from statistics import *
# Mean and Median => The graph is right skewed
print('Mean:', mean(fdist.values()))
print('Median:', median(fdist.values()))
```

```
Mean: 40.4375254047504
Median: 1.0
```

The median is far to the right of the mean which means the graph is right skewed. Thus, many words with low frequency are there. Those words typically add no value and would need a lot of memory and computation to save, so we decided to remove them from the text.

Thus, what remained at the end is words that have reasonable frequency such as the word "happy", "like", "hate" and so on which were around 7k words. These words now represent our dictionary.

Removing rows with empty values

Lastly, after all this preprocessing some features had empty values. More specifically, there were more than 15K instance with an empty value in one of its features, so we dropped them.

4. Feature Generation

Now, we have clean data, so we begin to extract features from it as following :

Words

As stated, we have more than 7K total words. At first, we tried to extract features from these words using Bag-of-words model, but that needed a tremendous amount of memory. Moreover, it will produce a sparse matrix with a lot of zeros since tweets naturally have small number of words. That would rather inefficient and will require a lot of time in training.

To solve this issue, we used a dictionary (or a map) for each instance(tweet) that maps the words of that tweet (after cleaning) to its frequency in that tweet. That saved a lot of memory and still captured features we needed.

Moreover, we decided that TFIDF might not be the best model to use to serve our purpose since we have important crucial words like "happy" that can determine the sentiment but occurs frequently. So, that model will give less weight to it.

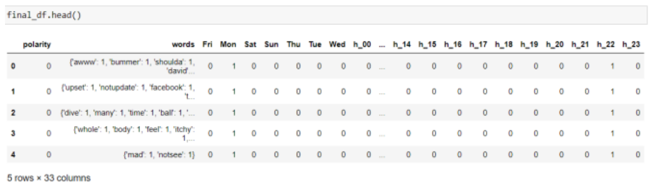
Days

As stated earlier, the day of the week is correlated with the sentiment, so we used one hot encoding to convert the categorical feature of day of the week to 7 features which are the 7 days of the week.

Hours

Similarly, the hour of the tweet is correlated with the sentiment, so we used one hot encoding to convert the hour of the tweet to one of 24 values h_00 through h_23, inclusive.

These are our **final features** now as shown in the below



```
final_df.head()
```

	polarity	words	Fri	Mon	Sat	Sun	Thu	Tue	Wed	h_00	...	h_14	h_15	h_16	h_17	h_18	h_19	h_20	h_21	h_22	h_23
0	0	{'awww': 1, 'bummer': 1, 'shoulda': 1, 'david'...	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	1	0
1	0	{'upset': 1, 'notupdate': 1, 'facebook': 1, 't...	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	1	0
2	0	{'dive': 1, 'many': 1, 'time': 1, 'ball': 1, '...	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	1	0
3	0	{'whole': 1, 'body': 1, 'feel': 1, 'itchy': 1,...	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	1	0
4	0	{'mad': 1, 'notsee': 1}	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	1	0

5 rows × 33 columns

data frame

5. Final Metrics

After extracting the features, we made final analysis on our data and features to find the following

Data Size

At the end, we have around 150K instance with 33 columns. One of which is the label represented in the polarity. The other 32 are features: 24 of which are for the hours, 7 are for the week days, and the last important one is for the words represented as dictionary of frequencies for each tweet.

```
final_df.shape
```

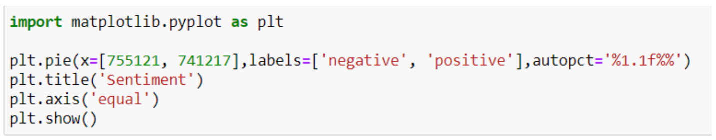
```
(1496338, 33)
```

Polarity Distribution

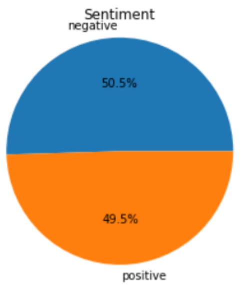
As we dropped around 100K instances, we needed to check the new distribution of polarity to make sure it is balanced. So, we checked it as follows:

```
final_df['polarity'].value_counts()
```

```
0    755121
1    741217
Name: polarity, dtype: int64
```



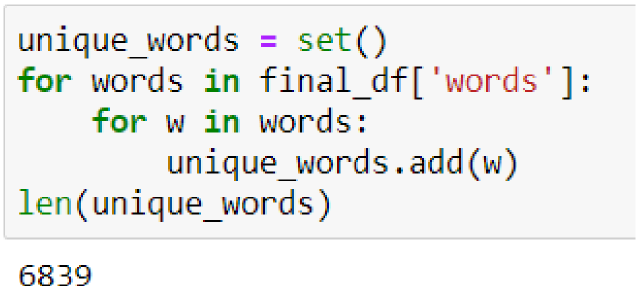
```
import matplotlib.pyplot as plt

plt.pie(x=[755121, 741217],labels=['negative', 'positive'],autopct='%1.1f%%')
plt.title('Sentiment')
plt.axis('equal')
plt.show()
```



It is clear from the numbers and the pie chart that even after removing those 100K, the polarity is still perfectly balanced (nearly), and that will help the model not to be biased.

Words Count

At the end, we exactly **6839** important words in our dictionary. We believe the words we have are representative of the tweet sentiment, and that this number of words will be large enough for model to train and produce good results



```
unique_words = set()
for words in final_df['words']:
    for w in words:
        unique_words.add(w)
len(unique_words)
```

```
6839
```