Machine Learning Engineer — Emmanuel Dickson

Nanodegree — July 15th, 2020

Capstone Project

# TWEET SENTIMENT EXTRACTION

## Definition

### Project Overview

With all of the tweets circulating every second it is hard to tell whether the sentiment behind a specific tweet will impact a company, or a person's, brand for being viral (positive), or devastate profit because it strikes a negative tone. Capturing sentiment in language is important in these times where decisions and reactions are created and updated in seconds. But, which words actually lead to the sentiment description? [1]

Sentiment analysis can be defined as the process of computationally interpreting and categorizing the feelings, emotions and even the intentions behind a text. This is customarily achieved through the use of Natural Language Processing (NLP) and text analysis tools. With the increasing importance a good and positive online persona has on a person or company's brand, it has become more imperative for people to be tactful with their choice of words while conversing and airing their views on an online platform like Twitter.

This project was a featured code competition on Kaggle.com

### Problem Statement

For this project, we are required to go beyond analysing the sentiment of the text data obtained from tweets and provide accurate insight on which portion of the tweets influenced its sentiment classification. The problem as described on the Kaggle competition page is as follows:
"Your objective is to construct a model that can look at the labeled sentiment for a given tweet and figure out what word or phrase best supports it."

There are a number of ways this problem can be approached, but for the purpose of this project we are going to attempt to solve this problem by implementing a simple algorithm which uses word counts to make predictions. Particularly, we will convert the collection of tweet texts into a matrix of words with their token counts using sklearn's CountVectorizer. We can then assign weights to the top words for the various sentiment categories. With these weights assigned, predictions can be made on tweets which will return the subset of words with the largest weight sum. Obtaining these subsets of words is our ultimate goal in this project.

**Metrics**

This project was based on a kaggle competition and the metric used in the competition is the word-level Jaccard score. This metric was equally adopted for the evaluation of this project.

The Jaccard score also known as the Jaccard similarity coefficient is usually used for measuring the similarity and diversity of sample sets and it is defined as the size of the intersection divided by the size of the union of the sample sets.[2]

This is defined mathematically as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

$$0 \leq J(A,B) \leq 1$$

A Python implementation of the metric described above is provided below:

```
def jaccard(str1, str2):
    a = set(str1.lower().split())
    b = set(str2.lower().split())
    c = a.intersection(b)
    return float(len(c)) / (len(a) + len(b) - len(c))
```

The formula for the overall metric is then given as:

$$score = \frac{1}{n}\sum_{i=1}^{n} jaccard(gt_i, dt_i)$$

where:

n = number of documents

jaccard = the function provided above

$gt_i$= the ith ground truth

$dt_i$= the ith prediction

## Analysis

### Data Exploration

The dataset for this project was provided by Kaggle. It comprises of three csv files namely:

i. train.csv
ii. test.csv
iii. sample_submission.csv

The train.csv file comprises of 27,481 unique values with 40% labeled as neutral, 31% labeled as positive and the remaining 28% labeled negative.

There are four columns in this dataset:

- textID – unique ID assigned to each labeled tweet
- text – the text contained in each tweet
- sentiment – the labeled sentiment of each tweet
- selected_text – The text that supports the tweet's sentiment. (This column is only present in the train.csv file)

| | textID | text | selected_text | sentiment |
|---|---|---|---|---|
| 0 | cb774db0d1 | I`d have responded, if I were going | I`d have responded, if I were going | neutral |
| 1 | 549e992a42 | Sooo SAD I will miss you here in San Diego!!! | Sooo SAD | negative |
| 2 | 088c60f138 | my boss is bullying me... | bullying me | negative |
| 3 | 9642c003ef | what interview! leave me alone | leave me alone | negative |
| 4 | 358bd9e861 | Sons of ****, why couldn`t they put them on t... | Sons of ****, | negative |

*Figure 1: Sample of train dataset*

As seen from the above figure, our data is basically raw texts from different tweets, hence there is a heavy presence of punctuations, digits, emoticons, urls etc. in the text fields. Interestingly, the selected_text also includes these characters as well. This piece of information will be taken into consideration when carrying out analysis on the dataset.

By calling the `.isnull().sum()` method on the train set, we got to know that there was a row of the data with null values. This row was removed by calling the `.dropna(inplace=True)` method.

The sample_submission.csv file is what is used to make predictions. This file contains the unique IDs for items present in the test set. The test data is unlabeled as it is used only for prediction purpose. Hence, the train set will be split into training (80%) and evaluation (20%) sets to be used for training and evaluation of our model.

## Exploratory Visualization

We built wordClouds for the various sentiment classes. A word Cloud is a visual representation of text data. It usually comprises of single words, and the importance of each word is depicted by their color or font size.

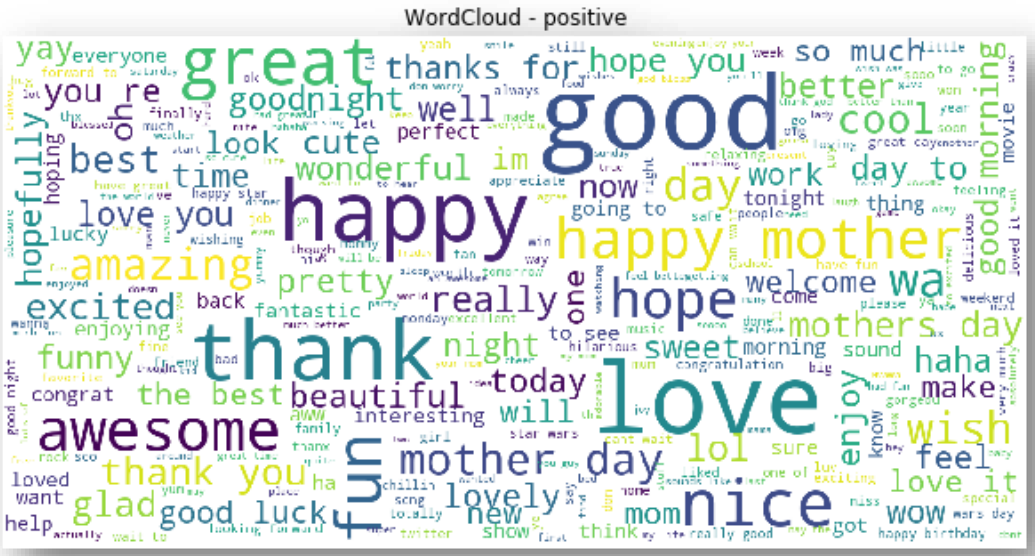


*Figure 2: WordCloud – positive*


*Figure 3: WordCLoud – negative*

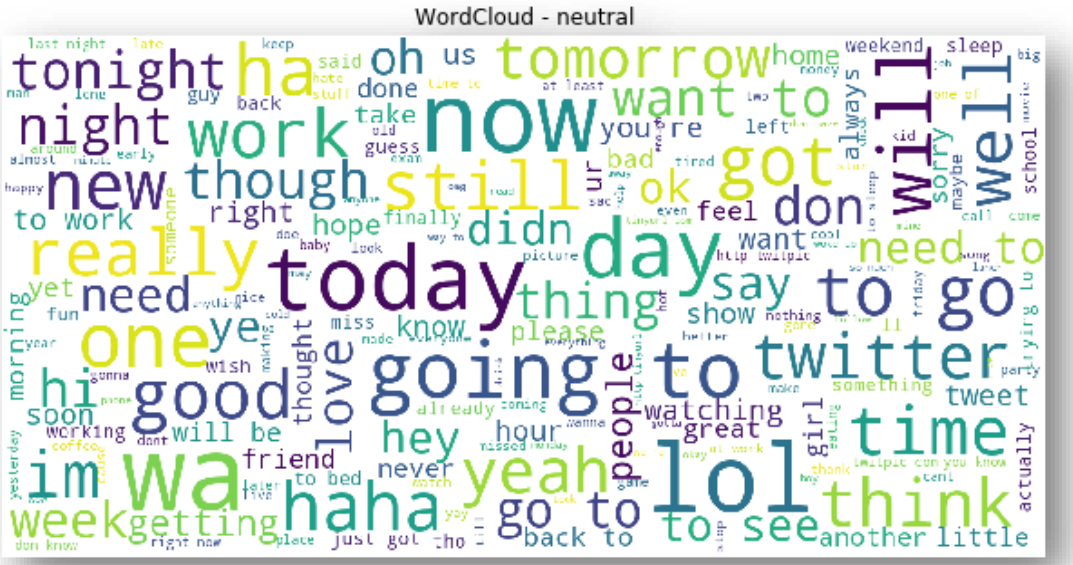


*Figure 4: WordCloud -neutral*

Also, for the purpose of exploration we defined a function named `common_words()` that takes in a sentiment as input and returns a list of the most common words in that category along with their count. This information is presented in the table below:

| **Positive** | | **Negative** | | **Neutral** | |
|---|---|---|---|---|---|
| Common_words | count | Common_words | count | Common_words | count |
| Good | 826 | Im | 452 | Im | 1041 |
| Happy | 730 | Miss | 358 | Get | 612 |
| Love | 697 | Sad | 343 | Go | 569 |
| Day | 456 | Sorry | 300 | Day | 492 |
| Thanks | 439 | Bad | 246 | Don't | 482 |
| Great | 364 | Hate | 230 | Going | 472 |
| Fun | 287 | Don't | 221 | Work | 467 |
| Nice | 267 | Cant | 201 | Like | 445 |
| Mothers | 259 | Sick | 166 | Got | 442 |
| Hope | 245 | Like | 162 | Today | 427 |
| Awesome | 232 | Sucks | 159 | Lol | 427 |
| Im | 185 | Feel | 158 | Time | 414 |
| Thank | 180 | Tired | 144 | Know | 407 |
| Like | 167 | Really | 137 | Back | 402 |
| Best | 154 | Good | 127 | One | 394 |
| Wish | 152 | Bored | 115 | U | 376 |
| Amazing | 135 | Day | 110 | See | 349 |

From the above table we can see that there are words which appear common to all three sentiments. e.g. "im". Also there are words which we wouldn't expect to find as common under a particular sentiment whereas they were. An example of such occurence is "Good" appearing under the common negative words. Due to this, we defined another function called `unique_words()` which also takes in a sentiment as input and returns a list of the most unique words in that category along with their count. This information is presented in the table below:

| **Positive** | | **Negative** | | **Neutral** | |
|---|---|---|---|---|---|
| unique words | count | unique words | count | unique words | count |
| Congratulations | 26 | Ache | 10 | Hows | 33 |
| Thnx | 8 | Saddest | 7 | Store | 28 |
| Lov | 8 | Hated | 6 | Guitar | 26 |
| Happ | 7 | Weak | 6 | Green | 22 |
| Talented | 7 | Suffering | 6 | Catching | 16 |
| Brilliant | 7 | Rly | 5 | Myspace | 16 |
| Appreciated | 6 | Devastated | 5 | Planning | 15 |
| Good | 6 | Pissing | 5 | Random | 15 |
| Amazin | 5 | Allergic | 4 | Hrs | 15 |
| Mommas | 5 | Cramps | 4 | Shop | 15 |

Based on this information, we can speculate that the unique words have a strong influence on the sentiment labelling of their various source tweets.

### Algorithms and Techniques

For this project, we resolved to use a term-weighting scheme to solving the problem. The scheme selected for this project was **tf-idf**.

According to Wikipedia: "tf–idf or TFIDF (short for term frequency–inverse document frequency), is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus[3]". This technique is usually used as a weighting factor in projects that include information retrieval, text mining, etc. The tf-idf value increases in direct proportion to the number of times a word appears in the text and then this value is offset by the total number of texts in the collection that also contains the word.

Particularly, we implemented the tf-idf algorithm defined by Nick Koprowicz[4] for assigning weights to the words. This algorithm is given below:

$$for\ each\ class\ j\ \in \{positive, neutral, negative\}$$

a. Find all the words $i$ in the tweets belonging to class *j*.
b. Calculate $n_{i,j}$

Where:

- $n_{i,j}$= the number of tweets in class *j* containing word *i*.

c. Calculate $p_{i,j} = \frac{n_{i,j}}{d_j}$,

Where:

- $d_j$ = the number of tweets in class *j*.
- $p_{i,j}$ = the proportion tweets in class *j* that contains word *i*.

d. Calculate $w_{i,j} = p_{i,j} - \sum_{k \neq j} p_{i,k}$

Where:

- $w_{i,j}$ = the weights assigned to each word within each class.

With these weights assigned to each word, predictions would be made on tweets which would return the subset of words with the largest weight sum.

The algorithm used for finding the selected text is given below:

$$for\ every\ tweet:$$

a. Let *j* be the sentiment of the tweet and *i* is the set of words in the tweet.
b. If j == neutral, return the entire text.
c. Otherwise, for each subset of words in the tweet, calculate $\sum_i w_{i,j}$
d. Return the subset of words with the largest sum, given that it exceeds some tolerance.

These algorithms were implemented in our notebook as will be shown in a later section.

**Benchmark**

Since the project was based on a Kaggle competition, I used the Kaggle leaderboard as a tool for comparing the performance obtained from my solution. Although knowing the nature of Kaggle competitions that it was highely unlikely that I beat the score, I resolved to make sure my solution was not more than 10% short of the winning score. Thankfully my final solution came close on the leaderboard falling behind by just 9.29%.

Also since the problem wasn't directly to make sentiment predictions, implementing a simple neural network didn't work out as I'd expected so I stuck with using the leaderboard for comparison.

## Methodology

### Data Preprocessing

We defined a function called `clean_text()` which takes in a text as input and makes the text lowercase, removes any punctuations or urls found within. The function also removes emoticons and texts containing numbers only. This function which is displayed below, was applied to the train set using a lambda function.

```
def clean_text(text):
    '''make text lowercase, remove punctuations, urls, words containing numbers,
emoticons, etc
    '''
    text = text.lower()
    text = re.sub('https?://\S+|www\.\S+', '', text)
    text = re.sub('[%s]' % re.escape(string.punctuation), '', text)
    text = re.sub('\n', '', text)
    text = re.sub('\w*\d\w*', '', text)
    text = re.sub('<.*?>+', '', text)

    return text
```

The training dataset was then split into a train set (X_train) and validation set (X_val) using sklearn's train_test_split function. The train size was assigned 80% of the data. Hence, the number of data points assigned to the train set was 21984 while the number of data points assigned to the validation set was 5496.

| | textID | text | selected_text | sentiment |
|---|---|---|---|---|
| 0 | cb774db0d1 | id have responded if i were going | id have responded if i were going | neutral |
| 1 | 549e992a42 | sooo sad i will miss you here in san diego | sooo sad | negative |
| 2 | 088c60f138 | my boss is bullying me | bullying me | negative |
| 3 | 9642c003ef | what interview leave me alone | leave me alone | negative |
| 4 | 358bd9e861 | sons of why couldnt they put them on the rel... | sons of | negative |

*Figure 5: a displayed portion of the dataset after preprocessing*

### Implementation

- The implementation of the the pre-defined tf-idf algorithms began by initializing the sklearn's CountVectorizer function to convert the words to a matrix of numbers. The hyperparameters specified for the model are as shown below. We then fitted and transformed each category's data points in the train set to the function.

```
# initialize CountVectorizer model
cv = CountVectorizer(analyzer='word',
                     max_df=0.95,
                     stop_words ='english')

# fit and transform training data
x_vectorized = cv.fit_transform(train_x)


# transform data based on sentiment
x_positive = cv.transform(train_positive['text'])
x_negative = cv.transform(train_negative['text'])
x_neutral = cv.transform(train_neutral['text'])
```

- Next we defined a function called `proportion_calculation()` which took in a dataset and its corresponding vectorised data points and returned a dictionary whose keys were the words under a given sentiment category and their values were the term frequency-weights assigned to them based on the predefined algorithm.

```
def proportion_calculation(data, cv_transformed):
    '''function to calculate the proportion of each words in a category'''

    # dictionary of words with corresponding count values
    words_dict = {}

    # create dataframes of vectorized arrays and their counts
    df = pd.DataFrame(cv_transformed.toarray(), columns=cv.get_feature_names())

    for i in cv.get_feature_names():
        # obtain sum of values in the given class
        count_sum = df[i].sum()
        # calculate overall proportion of items in the given class
        words_dict[i] = count_sum/len(data)


    return words_dict
```

- A function called `weight_calculation()` was then defined to calculate the adjusted inverse document frequency-weights for each value in the passed-in tf-dictionary. The function is displayed below:

```
def weight_calculation(target, cat_1, cat_2):
    adjusted_weight = {}
    for key, value in target.items():
        adjusted_weight[key] = target[key] - (cat_1[key] + cat_2[key])

    return adjusted_weight
```

- With the weights now obtained for each sentiment category, the next function that was defined was the function to predict the selected text based on the tf-idf weights

assigned to the various words in the dataset. The algorithm for this function was predefined in a previous section and the implementation is as shown below:

```
    def get_selected_text(row, word_dictionary, tolerance=0):
    '''algorithm for obtaining the selected text given the text ID'''

    # initialize tweet and corresponding sentiment given ID
    text = row['text']
    sentiment = row['sentiment']

    # define dictionary of words to use depending on sentiment
    if(sentiment == 'neutral'):
        return text # return entire tweet if sentiment category is neutral

    elif(sentiment == 'positive'):
        word_dict = word_dictionary['positive']
    elif (sentiment == 'negative'):
        word_dict = word_dictionary['negative']

    selected_text = str() # obtained result from our algorithm
    score = 0
    words = text.split()
    subset = [words[i:j+1] for i in range(len(words)) for j in range (i,
len(words))]
    sorted_list = sorted(subset, key=len) # sort subset of text according to length

    for i in range(len(subset)):
        n_sum = 0
        # calculating the sum of weights for each word in the subset of text
        for p in range(len(sorted_list[i])):
            if(sorted_list[i][p] in word_dict.keys()):
                n_sum += word_dict[sorted_list[i][p]]

        # update current selection if the sum is greater than the score
        if(n_sum > score + tolerance):
            score = n_sum
            selected_text = sorted_list[i]

    # if no good subsets, return whole text
    if(len(selected_text) == 0):
        selected_text = words


    return ' '.join(selected_text)
```

- Finally, the jaccard score code was implemented as given in a previous section of this report. Then a function for carrying out the evaluation of the model was defined. The tolerance value adopted for our model was 0.001.

```
 # calculate score for validation set
def validation(df, word_dictionary):
    tolerance = 0.001

    df['predicted_selection'] = str()

    for index, row in df.iterrows():
        selected_text = get_selected_text(row, word_dictionary, tolerance)
        df.loc[df['textID'] == row['textID'],['predicted_selection']] =
selected_text

    df['jaccard'] = df.apply(lambda x: jaccard(x['selected_text'],
x['predicted_selection']), axis = 1)
    jaccard_score = np.mean(df['jaccard'])

    print('jaccard score is: {}'.format(jaccard_score))
```

- The model was evaluated using the validation set and once we had obtained a satisfactory jaccard score, we then set out to apply the algorithm to the submission file to make predictions.
  This time around, we loaded the entire training set to obtain a more comprehensive tf-idf weight dictionary for the dataset using the functions we already defined while carrying out model evaluation and then we passed the submission file to the algorithm and obtained our predictions for the selected texts based on the provided Text IDs in the csv file.

**Refinement**

For our initial solution, the only preprocessing applied to the dataset was the lowering of the letter cases. Also we implemented the CountVectorizer with its default hyperparameter values unchanged. This was done in order to test the veracity of the proposed solution. With this initial solution, we obtained a jaccard score of 0.5689.

The score was improved on by adding more instructions to the preprocessing function for cleaning the data files more properly. This include removing punctuations, urls, numbers and lowering the text case. Also, we passed a value of 0.95 to the max_df parameter of the CountVectorizer and we also added the 'english' stopword to be used for tokenization. Finally after experimenting with various figures we settled for 0.001 as our tolerance value in the algorithm.

This preprocessing steps greatly improved the score of the model as we were able to reach the final value of 0.6673.

## Results

### Model Evaluation, Validation and Justification

The final jaccard score obtained on our validation set was: **0.66773.**
Given that this was a simple solution using just word counts to solve the defined problem and devoid of any pre-trained neural network, this was an impressive score. The winning score on the Kaggle competition was 0.73615. Our model was just 9.29% less of the winning score from over 2,227 team entries.
Passing the submission file to the model for prediction equally produced a decent result as shown below:

```
# display results
sample.head(10)
```

| | textID | selected_text |
|---|---|---|
| 0 | f87dea47db | last session of the day |
| 1 | 96d74cb729 | exciting precisely skyscrapers galore good twe... |
| 2 | eee518ae67 | shame |
| 3 | 01082688c6 | happy |
| 4 | 33987a8ee5 | i like it |
| 5 | 726e501993 | great |
| 6 | 261932614e | hates |
| 7 | afa11da83f | completely |
| 8 | e64208b4ef | and within a short time of the last clue all ... |
| 9 | 37bcad24ca | what did you get my day is alright havent do... |

## Conclusion

### Reflection

This project is actually my first attempt at solving an NLP problem on my own. So, I can liken my experience to a journey, one which was both exciting and frightening at the same time.

I particularly enjoyed the fact that the project allowed me the opportunity to work on a real-life dataset. I learnt how to clean up my data and make it ready for use in a machine learning model.

In researching for possible solutions to the stated problem, I was able to acquire some knowledge in the field of NLP. Implementing the algorithms from scratch further strengthened my python programming skills and helped me understand the inner workings of a typical ML algorithm.

My final problem produced a decent result, but I wouldn't recommend it to be used in a general setting for tweet extraction as there are far better approaches.

**Improvement**

Due to my limited knowledge in the field of NLP, I was unable to implement any of the more sophisticated solutions that were suggested online, but I have resolved to learn more on the subject and attempt to implement those solutions as well in the nearest future.

One of the solutions suggested was to model the problem as a Question-answering system where given a question (is sentiment positive or negative?) and a context (tweet text), we train a transformer model to find the answer (selected_text).

Another approach was to model the problem as a Named Entity Recognition (NER) problem. NER involves spotting named entities from a collection of text and then classifying them into a predefined set of categories.

Other solutions include utilizing the power of deep learning and training a deep convolutional neural network to learn the appropriate weights for each word.

---

**References**


1. https://www.kaggle.com/c/tweet-sentiment-extraction/overview
2. https://en.wikipedia.org/wiki/Jaccard_index
3. https://en.wikipedia.org/wiki/Tf-idf
4. https://www.kaggle.com/nkoprowicz/a-simple-solution-using-only-word-counts