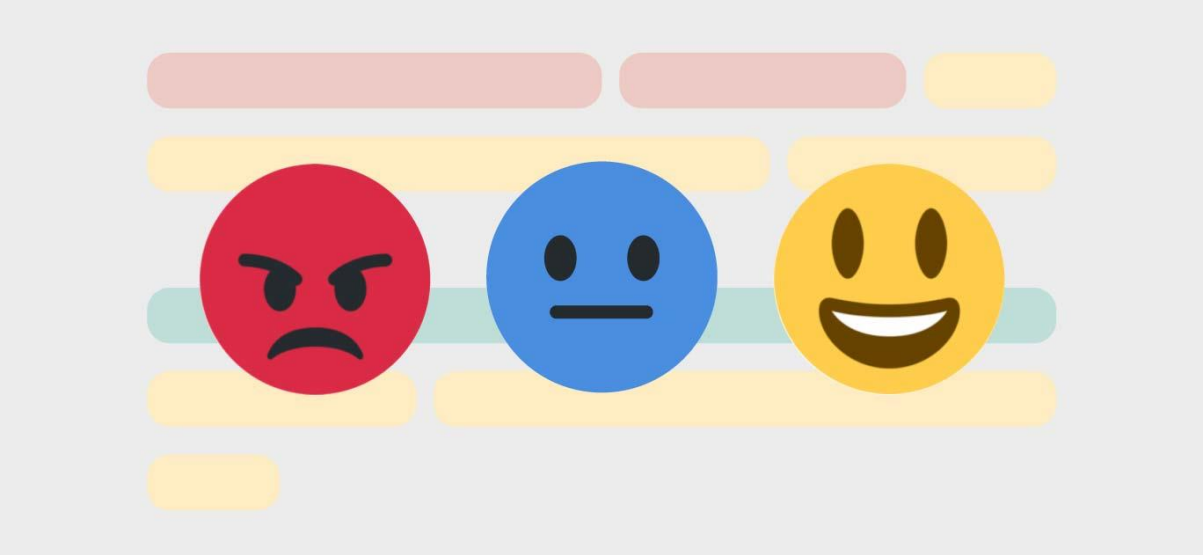


Sentiment analysis of Amazon Fine Food reviews using Python

Written by Arvid Magnusson, December 15, 2020

Sentiment analysis is the process of detecting positive or negative sentiment in text. People express their thoughts and opinions more than ever today. This type of data can be a goldmine for businesses if they have the necessary tools to handle it. By analysing the sentiment in surveys or social media conversations, brands can learn what makes customers happy or frustrated, so that they can tailor products after the customers opinions.

In this post we will look at how we can use sentiment analysis to determine whether a review is negative or positive. Is it possible to build a model that accurately classifies the sentiment of a review?

We will perform the following steps:

1. Dataset overview
2. Data pre-processing
3. Text cleaning
4. Modeling
5. Evaluation

1. Dataset overview

The dataset we will be using is the Amazon Fine Food Reviews. It consists of over 500,000 reviews of fine foods from Amazon. Some numbers about the data:

Reviews from Oct 1999 - Oct 2012
568,454 reviews
256,059 users
74,258 products
Number of Attributes/Columns in data: 10
Dataset source: https://www.kaggle.com/snap/amazon-fine-food-reviews

Before loading the data lets import a bunch of libraries we will use.

```
import numpy as np
import pandas as pd
import nltk
import sklearn
import matplotlib.pyplot as plt

from nltk.corpus import stopwords

from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import classification_report

import warnings
warnings.filterwarnings("ignore")#Ignoring unnecessory warnings
```

We load the dataset into a Dataframe using Pandas.

```
all_data = pd.read_csv("data/Reviews.csv")
all_data.head(3)
```

We can look at the first three rows of the dataset using .head() function.

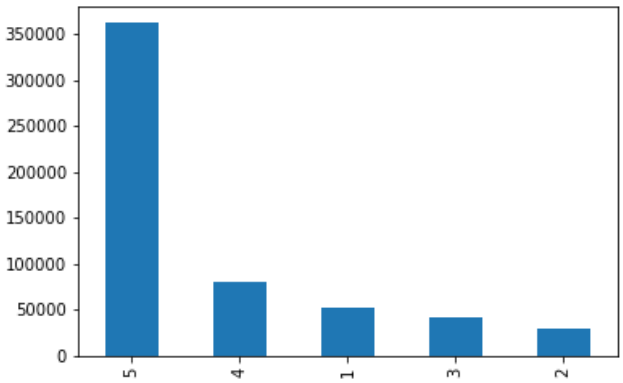
	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary	Text
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	1	5	1303862400	Good Quality Dog Food	I have bought several of the Vitality canned d...
1	2	B00813GRG4	A1D87F6ZCVE5NK	dll pa	0	0	1	1346976000	Not as Advertised	Product arrived labeled as Jumbo Salted Peanut...
2	3	B000LQOCH0	ABXLMWJIXXAIN	Natalia Corres "Natalia Corres"	1	1	4	1219017600	"Delight" says it all	This is a confection that has been around a fe...

2. Data pre-processing

Since the goal of this project is to classify the review as either positive or negative by only looking at the text, we will drop all attributes except “Score” and “Text”.

```
text_score_df = all_data[['Text','Score']].dropna()
ax=text_score_df.Score.value_counts().plot(kind='bar')
fig = ax.get_figure()
```

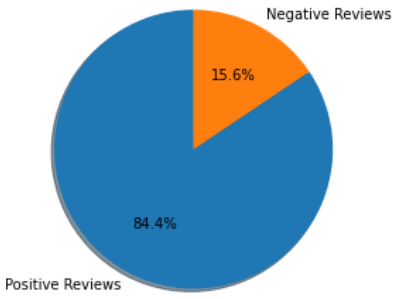
By plotting the score of the reviews, we can see that reviews are skewed towards positive, especially score 5.



We get rid of all the score 3 reviews because we assume that these are neutral and do not provide us with any useful sentiment information. Reviews with a score below 3 is seen as a negative review and a score above 3 is seen as positive.

```
text_score_df = text_score_df[text_score_df['Score'] != 3]
text_score_df.Score[text_score_df.Score < 3] = 0
text_score_df.Score[text_score_df.Score > 3] = 1
```

We then look at the distribution between positive and negative reviews.



As we can see there are a lot more reviews considered positive compared to negative. Most machine learning algorithms do not work very well with imbalanced datasets. In this dataset we have 84% positive and 16% negative reviews. Applying a model on this would make the model biased towards positive reviews. There are several ways of tackling this. In this case we have an abundancy of data. 500,000 reviews is a lot more than we need. We can therefore balance the dataset by using under-sampling. Under-sampling balances the dataset by reducing the size of the abundant class.

```
df_pos = text_score_df[text_score_df.Score==1][:10000]
df_neg = text_score_df[text_score_df.Score==0][:10000]
df = df_pos.append(df_neg) #Combine positive and negative
df = df.sample(frac=1).reset_index(drop=True) #Shuffle it and reset the index
```

Here we extract 10,000 positive and 10,000 negative reviews, join them together to a new dataframe and finally shuffle it. Now we have a balanced dataset which is much smaller than the original so we can work on it with reasonable execution times. Before we start cleaning the data we split it into 2 dataframes to separate the text from the score.

```
df_X = df['Text']
df_y = df['Score']
```

3. Text cleaning

Before making the prediction model we need to clean up irregularities from the text data. We begin by converting the text to lowercase. Then we use Pythons regular expression library to remove unwanted characters, such as html-tags and punctuations. Dealing with special characters can be very problematic and it is often not necessary to distinguish if a text is negative or positive.

```
import re #Regular expressions operations

temp =[]
snow = nltk.stem.SnowballStemmer('english')

for sentence in df_X:
    sentence = sentence.lower()                 # Converting to lowercase
    cleanr = re.compile('<.*?>')
    sentence = re.sub(cleanr, ' ', sentence)        #Removing HTML tags
    sentence = re.sub(r'[?|!|\'|"|#]',r'',sentence)
    sentence = re.sub(r'[.|,|)|(|\|/]',r' ',sentence)        #Removing Punctuations

```

The next step is to try to reduce words to their base form. For example, a text can use different forms of the same word, like “smell” and “smelled”. We humans know that this is the same word, just in different forms. But our sentiment model wont know and will treat these as two separate words. One way to reduce a word to its base form is to use stemming. *Stemming* usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time. Lets look at two examples:

```
print('smell => ',snow.stem('smell'))
print('smelled => ', snow.stem('smelled'))
```

```
smell =>  smell
smelled =>  smell
```

```
print('delicious => ',snow.stem('delicious'))
print('deliciously =>', snow.stem('deliciously'))
```

```
delicious =>  delici
deliciously => delici
```

As we can see it works well for smelled but for the word delicious it chops off the end and returns delici. This is the drawback of using stemming. It doesn’t use a dictionary for lookups to determine what word it is, it just chops off the end of words according to some rules. Another method is to use *Lemmatization*. It does things more “properly” with the use of a vocabulary and morphological analysis of words.

In this project we will use stemming because it is much faster and yields good enough results. Because in the end only different forms of “delicious” will be reduced to the word “delici”.

```
    words = [snow.stem(word) for word in sentence.split() if word not in stopwords.words('english')]
    temp.append(words)

df_X = temp
```

We stem all the words and remove the words that are contained in the stopwords dictionary from the NLTK library.

4. Modeling

We will first convert all reviews to vector space using TF-IDF. TF-IDF is an information retrieval technique that weighs a term’s frequency (TF) and its inverse document frequency (IDF). Each word has its respective TF and IDF score. The product of the TF and IDF scores of a word is called the TFIDF weight of that word. Put simply, the higher the TFIDF score (weight), the more unique a word is and vice versa.

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

tfidf = TfidfVectorizer(max_features=5000, ngram_range=(1,2))
pipe = Pipeline([('tfidf',tfidf),
                 ('clf', LogisticRegression ())
                  ])

X_train, X_test, y_train, y_test = train_test_split(df_X,df_y, test_size = 0.20, random_state = 0, shuffle=True)

pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
```

The ngram_range specifies that we want to take bigrams into account. This way the model will look at the sequence of two words, like “Not good”, and not treat it as two separate words but instead one word with a negative sentiment.

We apply Logistic regression as the learning method as it usually works well for sentiment analysis.

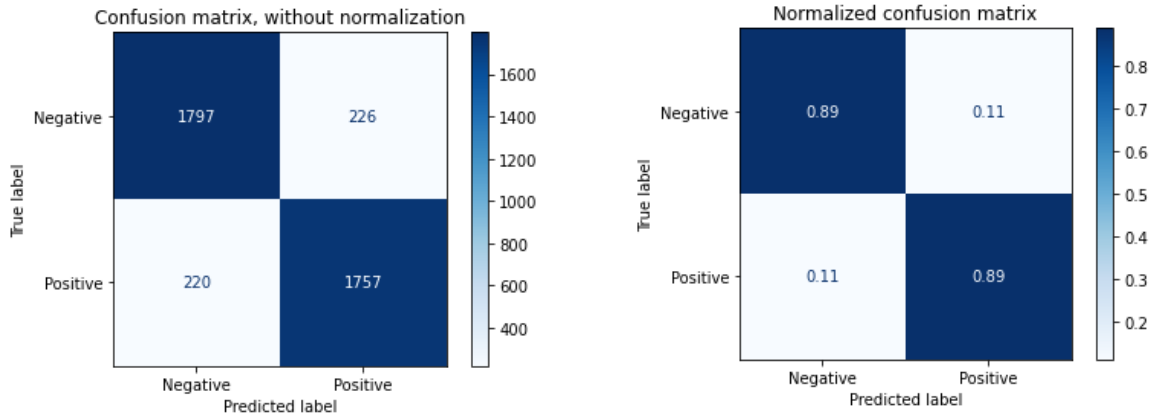
5. Evaluation

To evaluate how well our model performs we can use the accuracy score metric from the sklearn library.

```
1 sklearn.metrics.accuracy_score(y_test, y_pred)
```

```
0.8885
```

We receive an accuracy of around 89% which is pretty good! Accuracy doesn't always tell the whole story though so lets also look at the confusion matrix.



The confusion matrix tells us more about how the reviews were classified. We can see that the model is equally good at predicting negative reviews as positive reviews. This is because the model was trained on a balanced dataset. If the representative, and thereby imbalanced, dataset was used, where 84% of reviews were positive, we would end up with a lot more false positives. The accuracy would still be pretty good, but that's only because 84% of all reviews are labelled positive, so if the model were to classify all reviews as positive the accuracy would still be 84%.

Lets also have a look at the top positive and negative words to see if the sentiment of it makes sense.

```
classifier = pipe.named_steps['clf']
vec = pipe.named_steps['tfidf']
w = vec.get_feature_names()

coef = classifier.coef_.tolist()[0]
coeff_df = pd.DataFrame({'Word' : w, 'Coefficient' : coef})
coeff_df = coeff_df.sort_values(['Coefficient', 'Word'], ascending=[0, 1])
print('')
print('-Top 20 positive-')
print(coeff_df.head(20).to_string(index=False))
print('')
print('-Top 20 negative-')
print(coeff_df.tail(20).to_string(index=False))
```

```
-Top 20 positive-
           Word  Coefficient
          great     7.428453
         delici     6.014505
           best     5.957562
           love     5.815778
        perfect     5.587768
           good     4.855170
          excel     4.647476
           nice     4.523342
           amaz     3.943157
        favorit     3.880467
high recommend     3.689343
         wonder     3.489961
         smooth     3.436415
           easi     3.432654
          happi     3.265255
          alway     3.123667
          thank     3.112590
          yummi     3.078931
           glad     3.015911
         awesom     2.944945

-Top 20 negative-
           Word  Coefficient
           away    -2.654218
          stale    -2.672848
          throw    -2.808242
           mayb    -2.844984
          threw    -2.917322
           wast    -2.964869
           weak    -3.069429
          didnt    -3.072373
           tast    -3.195692
          bland    -3.283603
          money    -3.415524
        unfortun    -3.514123
             aw     -3.597575
        thought    -3.613508
        horribl    -3.694950
          worst    -3.864274
            bad    -3.909320
         return    -3.953236
        terribl    -4.035507
     disappoint    -7.310408
```

Summary

In this blog post we have looked at how we can use sentiment analysis to determine whether a review is either positive or negative. With the help of NLP libraries and machine learning models we were able to get an accuracy of 89% which must be seen as good result.

When looking at the top positive/negative words we can see that a lot of the ends of words have been chopped off as a result of the stemming, which was discussed earlier in this post. Though it may look weird and wrong, it is not really an issue for the model as long as all the reviews to be classified goes through the same cleaning process.

This method used TFIDF together with logistic regression to predict the sentiment but there are numerous other ways to go about this type of problem. Instead of logistic regression you can use Naïve Bayes or SVM to make the predictions. Another way is to use Deep learning. You can also choose to have a rule-based approach to the problem by using a lexicon-based solution. This method uses lists of polarized words where each word is annotated with an emotional value.

With that said it’s important to choose the approach that works well for your specific type of sentiment problem. Logistic regression + TFIDF is not always the best method but for this problem it worked well.