Amazon Fine Food Reviews Analysis

July 15, 2018

1 [7] Amazon Fine Food Reviews Analysis

Data Source: https://www.kaggle.com/snap/amazon-fine-food-reviews

The Amazon Fine Food Reviews dataset consists of reviews of fine foods from Amazon.

Number of reviews: 568,454 Number of users: 256,059 Number of products: 74,258 Timespan: Oct 1999 - Oct 2012 Number of Attributes/Columns in data: 10

Attribute Information:

1. Id
2. ProductId - unique identifier for the product
3. UserId - unqiue identifier for the user
4. ProfileName
5. HelpfulnessNumerator - number of users who found the review helpful
6. HelpfulnessDenominator - number of users who indicated whether they found the review helpful or not
7. Score - rating between 1 and 5
8. Time - timestamp for the review
9. Summary - brief summary of the review
10. Text - text of the review

Objective: Given a review, determine whether the review is positive (Rating of 4 or 5) or negative (rating of 1 or 2).

[Q] How to determine if a review is positive or negative? [Ans] We could use the Score/Rating. A rating of 4 or 5 could be cosnidered a positive review. A review of 1 or 2 could be considered negative. A review of 3 is nuetral and ignored. This is an approximate and proxy way of determining the polarity (positivity/negativity) of a review.

1.1 [7.1] Loading the data

The dataset is available in two forms 1. .csv file 2. SQLite Database

In order to load the data, We have used the SQLITE dataset as it easier to query the data and visualise the data efficiently.

Here as we only want to get the global sentiment of the recommendations (positive or negative), we will purposefully ignore all Scores equal to 3. If the score id above 3, then the recommendation wil be set to "positive". Otherwise, it will be set to "negative".

```
In [25]: import sqlite3
         import pandas as pd
         import numpy as np
         import nltk
         import string
         import matplotlib.pyplot as plt
         import seaborn as sns
         from sklearn.feature_extraction.text import TfidfTransformer
         from sklearn.feature_extraction.text import TfidfVectorizer

         from sklearn.feature_extraction.text import CountVectorizer
         from sklearn.metrics import confusion_matrix
         from sklearn import metrics
         from sklearn.metrics import roc_curve, auc
         from nltk.stem.porter import PorterStemmer



         # using the SQLite Table to read data.
         con = sqlite3.connect('C://Users/vivekanandam/Desktop/applied AI/datasets/amazon-fine



         #filtering only positive and negative reviews i.e.
         # not taking into consideration those reviews with Score=3
         filtered_data = pd.read_sql_query("""
         SELECT *
         FROM Reviews
         WHERE Score != 3
         """, con)





         # Give reviews with Score>3 a positive rating, and reviews with a score<3 a negative
         def partition(x):
             if x < 3:
                 return 'negative'
             return 'positive'

         #changing reviews with score less than 3 to be positive and vice-versa
         actualScore = filtered_data['Score']
         positiveNegative = actualScore.map(partition)
         filtered_data['Score'] = positiveNegative

In [26]: print(filtered_data.shape) #looking at the number of attributes and size of the data
         filtered_data.head()

(525814, 10)
```

```
Out[26]:    Id   ProductId          UserId                      ProfileName  \
        0   1  B001E4KFG0  A3SGXH7AUHU8GW                        delmartian
        1   2  B00813GRG4  A1D87F6ZCVE5NK                            dll pa
        2   3  B000LQOCH0   ABXLMWJIXXAIN  Natalia Corres "Natalia Corres"
        3   4  B000UA0QIQ  A395BORC6FGVXV                              Karl
        4   5  B006K2ZZ7K  A1UQRSCLF8GW1T    Michael D. Bigham "M. Wassir"

           HelpfulnessNumerator  HelpfulnessDenominator     Score        Time  \
        0                     1                       1  positive  1303862400
        1                     0                       0  negative  1346976000
        2                     1                       1  positive  1219017600
        3                     3                       3  negative  1307923200
        4                     0                       0  positive  1350777600

                         Summary                                               Text
        0  Good Quality Dog Food  I have bought several of the Vitality canned d...
        1      Not as Advertised  Product arrived labeled as Jumbo Salted Peanut...
        2  "Delight" says it all  This is a confection that has been around a fe...
        3         Cough Medicine  If you are looking for the secret ingredient i...
        4            Great taffy  Great taffy at a great price.  There was a wid...
```

2 Exploratory Data Analysis

2.1 [7.1.2] Data Cleaning: Deduplication

It is observed (as shown in the table below) that the reviews data had many duplicate entries. Hence it was necessary to remove duplicates in order to get unbiased results for the analysis of the data. Following is an example:

```
In [27]: filtered_data[['Summary','Score','Text']].head()
```

```
Out[27]:                 Summary     Score  \
        0  Good Quality Dog Food  positive
        1      Not as Advertised  negative
        2  "Delight" says it all  positive
        3         Cough Medicine  negative
        4            Great taffy  positive

                                                        Text
        0  I have bought several of the Vitality canned d...
        1  Product arrived labeled as Jumbo Salted Peanut...
        2  This is a confection that has been around a fe...
        3  If you are looking for the secret ingredient i...
        4  Great taffy at a great price.  There was a wid...
```

```
In [28]: display= pd.read_sql_query("""
         SELECT *
```

```
FROM Reviews
WHERE Score != 3 AND UserId="AR5J8UI46CURR"
ORDER BY ProductID
""", con)
display
```

Out[28]:

```
       Id   ProductId         UserId      ProfileName  HelpfulnessNumerator  \
0   78445  B000HDL1RQ  AR5J8UI46CURR  Geetha Krishnan                     2
1  138317  B000HDOPYC  AR5J8UI46CURR  Geetha Krishnan                     2
2  138277  B000HDOPYM  AR5J8UI46CURR  Geetha Krishnan                     2
3   73791  B000HDOPZG  AR5J8UI46CURR  Geetha Krishnan                     2
4  155049  B000PAQ75C  AR5J8UI46CURR  Geetha Krishnan                     2

   HelpfulnessDenominator  Score        Time  \
0                       2      5  1199577600
1                       2      5  1199577600
2                       2      5  1199577600
3                       2      5  1199577600
4                       2      5  1199577600

                             Summary  \
0  LOACKER QUADRATINI VANILLA WAFERS
1  LOACKER QUADRATINI VANILLA WAFERS
2  LOACKER QUADRATINI VANILLA WAFERS
3  LOACKER QUADRATINI VANILLA WAFERS
4  LOACKER QUADRATINI VANILLA WAFERS

                                                Text
0  DELICIOUS WAFERS. I FIND THAT EUROPEAN WAFERS ...
1  DELICIOUS WAFERS. I FIND THAT EUROPEAN WAFERS ...
2  DELICIOUS WAFERS. I FIND THAT EUROPEAN WAFERS ...
3  DELICIOUS WAFERS. I FIND THAT EUROPEAN WAFERS ...
4  DELICIOUS WAFERS. I FIND THAT EUROPEAN WAFERS ...
```

As can be seen above the same user has multiple reviews of the with the same values for HelpfulnessNumerator, HelpfulnessDenominator, Score, Time, Summary and Text and on doing analysis it was found that ProductId=B000HDOPZG was Loacker Quadratini Vanilla Wafer Cookies, 8.82-Ounce Packages (Pack of 8) ProductId=B000HDL1RQ was Loacker Quadratini Lemon Wafer Cookies, 8.82-Ounce Packages (Pack of 8) and so on

It was inferred after analysis that reviews with same parameters other than ProductId belonged to the same product just having different flavour or quantity. Hence in order to reduce redundancy it was decided to eliminate the rows having same parameters.

The method used for the same was that we first sort the data according to ProductId and then just keep the first similar product review and delelte the others. for eg. in the above just the review for ProductId=B000HDL1RQ remains. This method ensures that there is only one representative for each product and deduplication without sorting would lead to possibility of different representatives still existing for the same product.

In [29]:

```
#Sorting data according to ProductId in ascending order
```

```
sorted_data=filtered_data.sort_values('ProductId', axis=0, ascending=True, inplace=Fal
```

In [30]:
```
#Deduplication of entries
final=sorted_data.drop_duplicates(subset={"UserId","ProfileName","Time","Text"}, keep=
final.shape
```

Out[30]:
```
(364173, 10)
```

In [31]:
```
#Checking to see how much % of data still remains
(final['Id'].size*1.0)/(filtered_data['Id'].size*1.0)*100
```

Out[31]:
```
69.25890143662969
```

Observation:- It was also seen that in two rows given below the value of HelpfulnessNumerator is greater than HelpfulnessDenominator which is not practically possible hence these two rows too are removed from calcualtions

In [32]:
```
display= pd.read_sql_query("""
SELECT *
FROM Reviews
WHERE Score != 3 AND Id=44737 OR Id=64422
ORDER BY ProductID
""", con)
display
```

Out[32]:
```
      Id   ProductId          UserId              ProfileName  \
0  64422  B000MIDROQ  A161DK06JJMCYF  J. E. Stephens "Jeanne"
1  44737  B001EQ55RW  A2V0I904FH7ABY                      Ram

   HelpfulnessNumerator  HelpfulnessDenominator  Score        Time  \
0                     3                       1      5  1224892800
1                     3                       2      4  1212883200

                                 Summary  \
0            Bought This for My Son at College
1  Pure cocoa taste with crunchy almonds inside

                                             Text
0  My son loves spaghetti so I didn't hesitate or...
1  It was almost a 'love at first bite' - the per...
```

In [33]:
```
final=final[final.HelpfulnessNumerator<=final.HelpfulnessDenominator]
```

In [34]:
```
#Before starting the next phase of preprocessing lets see the number of entries left
print(final.shape)

#How many positive and negative reviews are present in our dataset?
final['Score'].value_counts()
```

```
(364171, 10)
```

```
Out[34]: positive    307061
         negative     57110
         Name: Score, dtype: int64
```

2.2 7.2.3 Text Preprocessing: Stemming, stop-word removal and Lemmatization.

Now that we have finished deduplication our data requires some preprocessing before we go on further with analysis and making the prediction model.

Hence in the Preprocessing phase we do the following in the order below:-

1. Begin by removing the html tags
2. Remove any punctuations or limited set of special characters like , or . or # etc.
3. Check if the word is made up of english letters and is not alpha-numeric
4. Check to see if the length of the word is greater than 2 (as it was researched that there is no adjective in 2-letters)
5. Convert the word to lowercase
6. Remove Stopwords
7. Finally Snowball Stemming the word (it was obsereved to be better than Porter Stemming)

After which we collect the words used to describe positive and negative reviews

```
In [35]: final=final[0:2000] #sampling the 2000 data points for further analysis
```

```
In [36]: # find sentences containing HTML tags
         import re
         i=0;
         for sent in final['Text'].values:
             if (len(re.findall('<.*?>', sent))):
                 print(i)
                 print(sent)
                 break;
             i += 1;
```

```
6
I set aside at least an hour each day to read to my son (3 y/o). At this point, I consider myse
```

```
In [37]: # Tutorial about Python regular expressions: https://pymotw.com/2/re/
         import string
         from nltk.corpus import stopwords
         from nltk.stem import PorterStemmer
         from nltk.stem.wordnet import WordNetLemmatizer

         stop = set(stopwords.words('english')) #set of stopwords
         sno = nltk.stem.SnowballStemmer('english') #initialising the snowball stemmer
```

```
def cleanhtml(sentence): #function to clean the word of any html-tags
    cleanr = re.compile('<.*?>')
    cleantext = re.sub(cleanr, ' ', sentence)
    return cleantext
def cleanpunc(sentence): #function to clean the word of any punctuation or special ch
    cleaned = re.sub(r'[?|!|\'|"|#]',r'',sentence)
    cleaned = re.sub(r'[.|,|)|(|\|/]',r' ',cleaned)
    return  cleaned
print(stop)
print('************************************')
print(sno.stem('tasty'))
```

```
{'all', 'being', 'a', 'off', 'as', 'no', 're', 'which', 'over', 'then', 'yourself', 'during',
************************************
tasti
```

In [38]:
```
#Code for implementing step-by-step the checks mentioned in the pre-processing phase
# this code takes a while to run as it needs to run on 500k sentences.
i=0
str1=' '
final_string=[]
all_positive_words=[] # store words from +ve reviews here
all_negative_words=[] # store words from -ve reviews here.
s=''
for sent in final['Text'].values:
    filtered_sentence=[]
    #print(sent);
    sent=cleanhtml(sent) # remove HTMl tags
    for w in sent.split():
        for cleaned_words in cleanpunc(w).split():
            if((cleaned_words.isalpha()) & (len(cleaned_words)>2)):
                if(cleaned_words.lower() not in stop):
                    s=(sno.stem(cleaned_words.lower())).encode('utf8')
                    filtered_sentence.append(s)
                    if (final['Score'].values)[i] == 'positive':
                        all_positive_words.append(s) #list of all words used to descr
                    if(final['Score'].values)[i] == 'negative':
                        all_negative_words.append(s) #list of all words used to descr
                else:
                    continue
            else:
                continue
    #print(filtered_sentence)
    str1 = b" ".join(filtered_sentence) #final string of cleaned words
    #print("***********************************************************************")
```

```
            final_string.append(str1)
            i+=1
```

```
In [39]: final['CleanedText']=final_string #adding a column of CleanedText which displays the 
```

3 [7.2.2] Bag of Words (BoW)

```
In [40]: #BoW
         count_vect = CountVectorizer() #in scikit-learn
         final_counts = count_vect.fit_transform(final['Text'].values)
```

```
In [41]: type(final_counts)
```

```
Out[41]: scipy.sparse.csr.csr_matrix
```

```
In [42]: final_counts.get_shape()
```

```
Out[42]: (2000, 10800)
```

4 T-SNE on BoW

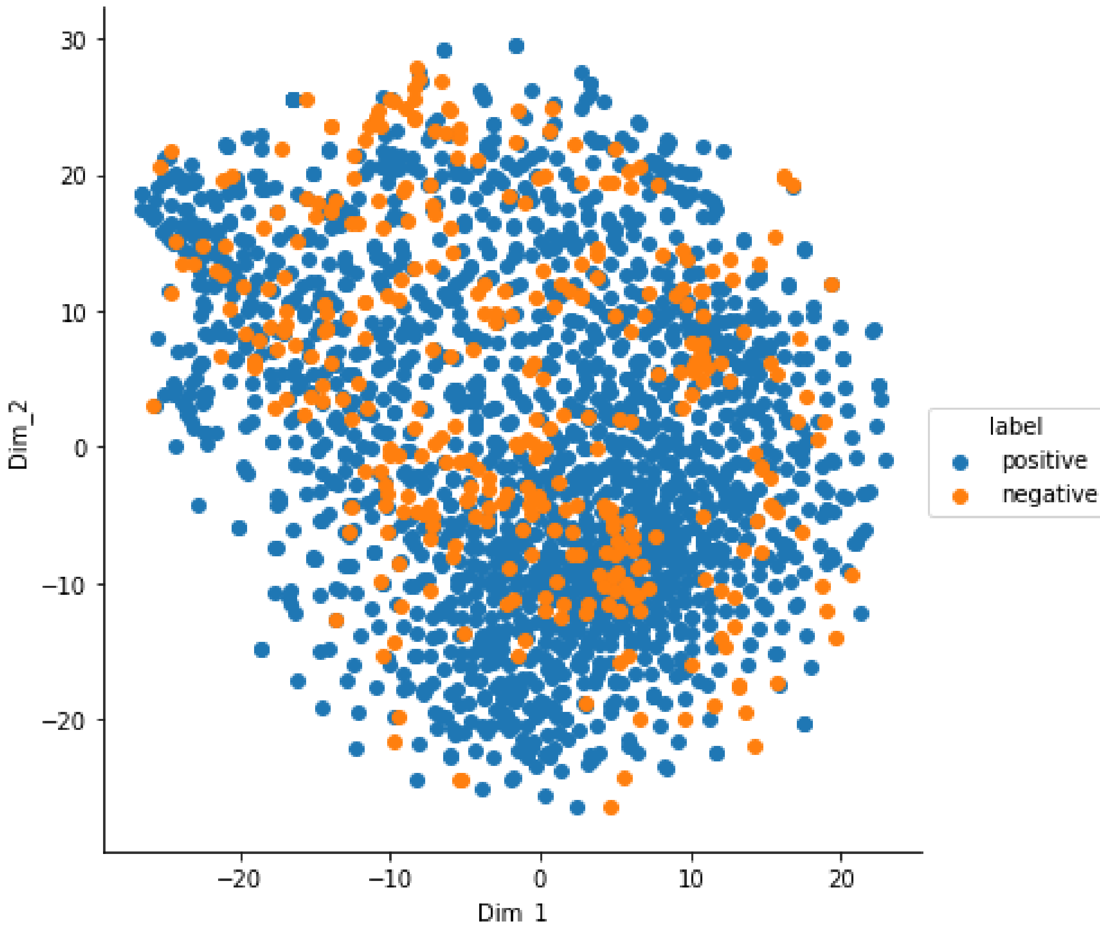
```
In [43]: from sklearn.manifold import TSNE
         data_2000 = final_counts[0:2000,:]
         top_2000 = data_2000.toarray()
         labels = final['Score']
         labels_2000 = labels[0:2000]

         model = TSNE(n_components=2, random_state=0,n_iter=5000)
         tsne_data = model.fit_transform(top_2000)

             # creating a new data frame which help us in ploting the result

         tsne_data = np.vstack((tsne_data.T, labels_2000)).T
         tsne_df = pd.DataFrame(data=tsne_data, columns=("Dim_1", "Dim_2","label"))

             # Ploting the result of tsne
         sns.FacetGrid(tsne_df, hue="label", size=6).map(plt.scatter, 'Dim_1', 'Dim_2').add_le
         plt.show()
```



4.1 [7.2.4] Bi-Grams and n-Grams.

Motivation

Now that we have our list of words describing positive and negative reviews lets analyse them. We begin analysis by getting the frequency distribution of the words as shown below

```
In [44]: freq_dist_positive=nltk.FreqDist(all_positive_words)
         freq_dist_negative=nltk.FreqDist(all_negative_words)
         print("Most Common Positive Words : ",freq_dist_positive.most_common(20))
         print("Most Common Negative Words : ",freq_dist_negative.most_common(20))
```

```
Most Common Positive Words :  [(b'food', 1314), (b'dog', 1002), (b'cat', 994), (b'trap', 948),
Most Common Negative Words :  [(b'food', 312), (b'dog', 260), (b'trap', 207), (b'cat', 206), (
```

Observation:- From the above it can be seen that the most common positive and the negative words overlap for eg. 'like' could be used as 'not like' etc. So, it is a good idea to consider pairs of consequent words (bi-grams) or q sequnce of n consecutive words (n-grams)

In [45]:
```
#bi-gram, tri-gram and n-gram

#removing stop words like "not" should be avoided before building n-grams
count_vect = CountVectorizer(ngram_range=(1,2) ) #in scikit-learn
final_bigram_counts = count_vect.fit_transform(final['Text'].values)
```

In [46]:
```
final_bigram_counts.get_shape()
```

Out[46]:
```
(2000, 98992)
```

5 [7.2.5] TF-IDF

In [47]:
```
tf_idf_vect = TfidfVectorizer(ngram_range=(1,2))
final_tf_idf = tf_idf_vect.fit_transform(final['Text'].values)
```

In [48]:
```
print(final_tf_idf.get_shape())
type(final_tf_idf)
```

```
(2000, 98992)
```

Out[48]:
```
scipy.sparse.csr.csr_matrix
```

In [49]:
```
features = tf_idf_vect.get_feature_names()
len(features)
```

Out[49]:
```
98992
```

In [50]:
```
from scipy.sparse import csr_matrix
data1 = csr_matrix(final_tf_idf)
data2=data1.todense()
print(data2.shape)
```

```
(2000, 98992)
```

In [51]:
```
labl = final['Score']
```

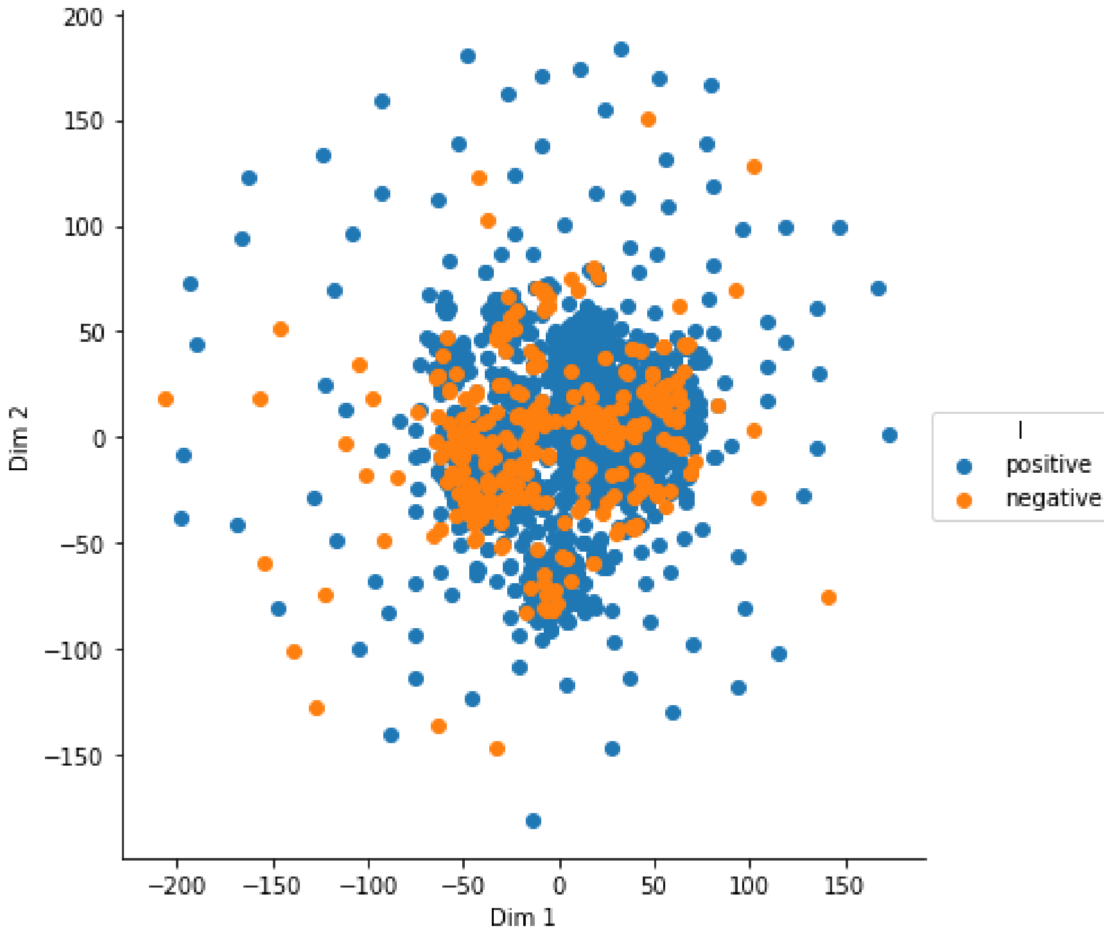
6 T-SNE on TF-IDF

In [52]:
```
from sklearn.preprocessing import StandardScaler
standardized_data = StandardScaler().fit_transform(data2)
print(standardized_data.shape)
```

```
(2000, 98992)
```

In [53]:
```
from sklearn.manifold import TSNE
model=TSNE(n_components = 2,random_state=0, perplexity=10,  n_iter=2000)
tsne_data = model.fit_transform(data2)
tsne_data = np.vstack((tsne_data.T, labl)).T
```

```
In [54]: tsne_df = pd.DataFrame(data=tsne_data, columns=("Dim 1", "Dim 2", "l"))
         sns.FacetGrid(tsne_df,hue="l",size=6).map(plt.scatter,'Dim 1','Dim 2').add_legend()
         plt.show()
```



7 [7.2.6] Word2Vec

```
In [ ]: # Using Google News Word2Vectors
        import gensim
        from gensim.models import Word2Vec
        from gensim.models import KeyedVectors
        import pickle

        # in this project we are using a pretrained model by google
        # its 3.3G file, once you load this into your memory
        # it occupies ~9Gb, so please do this step only if you have >12G of ram
        # we will provide a pickle file wich contains a dict ,
        # and it contains all our courpus words as keys and  model[word] as values
```

```
# To use this code-snippet, download "GoogleNews-vectors-negative300.bin"
# from https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit
# it's 1.9GB in size.


model = KeyedVectors.load_word2vec_format('C:/Users/vivekanandam/Downloads/Compressed/(
```

```
C:\Users\vivekanandam\Anaconda3\lib\site-packages\gensim\utils.py:1209: UserWarning: detected W
  warnings.warn("detected Windows; aliasing chunkize to chunkize_serial")
```

```
In [ ]: model.wv['computer']
```

```
In [ ]: model.wv.similarity('woman', 'man')
```

```
In [ ]: model.wv.most_similar('woman')
```

```
In [ ]: model.wv.most_similar('tasti')  # "tasti" is the stemmed word for tasty, tastful
```

```
In [ ]: model.wv.most_similar('tasty')
```

```
In [ ]: model.wv.similarity('tasty', 'tast')
```

```
In [ ]: # Train your own Word2Vec model using your own text corpus
        import gensim
        i=0
        list_of_sent=[]
        for sent in final['Text'].values:
            filtered_sentence=[]
            sent=cleanhtml(sent)
            for w in sent.split():
                for cleaned_words in cleanpunc(w).split():
                    if(cleaned_words.isalpha()):
                        filtered_sentence.append(cleaned_words.lower())
                    else:
                        continue
            list_of_sent.append(filtered_sentence)
```

```
In [ ]: print(final['Text'].values[0])
        print("*****************************************************************")
        print(list_of_sent[0])
```

```
In [ ]: w2v_model=gensim.models.Word2Vec(list_of_sent,min_count=5,size=50, workers=4)
```

```
In [ ]: words = list(w2v_model.wv.vocab)
        print(len(words))
```

```
In [ ]: w2v_model.wv.most_similar('tasty')
```

```
In [ ]: w2v_model.wv.most_similar('like')
```

```
In [ ]: count_vect_feat = count_vect.get_feature_names() # list of words in the BoW
        count_vect_feat.index('like')
        print(count_vect_feat[64055])
```

8 [7.2.7] Avg W2V, TFIDF-W2V

In [ ]:
```
# average Word2Vec
# compute average word2vec for each review.
sent_vectors = []; # the avg-w2v for each sentence/review is stored in this list
for sent in list_of_sent: # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length
    cnt_words =0; # num of words with a valid vector in the sentence/review
    for word in sent: # for each word in a review/sentence
        try:
            vec = w2v_model.wv[word]
            sent_vec += vec
            cnt_words += 1
        except:
            pass
    sent_vec /= cnt_words
    sent_vectors.append(sent_vec)
print(len(sent_vectors))
print(len(sent_vectors[0]))
```

In [ ]:
```
# TF-IDF weighted Word2Vec
tfidf_feat = tf_idf_vect.get_feature_names() # tfidf words/col-names
# final_tf_idf is the sparse matrix with row= sentence, col=word and cell_val = tfidf

tfidf_sent_vectors = []; # the tfidf-w2v for each sentence/review is stored in this li
row=0;
for sent in list_of_sent: # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length
    weight_sum =0; # num of words with a valid vector in the sentence/review
    for word in sent: # for each word in a review/sentence
        try:
            vec = w2v_model.wv[word]
            # obtain the tf_idfidf of a word in a sentence/review
            tfidf = final_tf_idf[row, tfidf_feat.index(word)]
            sent_vec += (vec * tf_idf)
            weight_sum += tf_idf
        except:
            pass
    sent_vec /= weight_sum
    tfidf_sent_vectors.append(sent_vec)
    row += 1
```

9 Assignment