TruncatedSVD on Amazon Fine Food Reviews

April 21, 2019

1 Amazon Fine Food Reviews Analysis

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

EDA: https://nycdatascience.com/blog/student-works/amazon-fine-foods-visualization/

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 Score/Rating. A rating of 4 or 5 can be cosnidered as a positive review. A rating of 1 or 2 can be considered as negative one. A review of rating 3 is considered nuetral and such reviews are ignored from our analysis. This is an approximate and proxy way of determining the polarity (positivity/negativity) of a review.

2 [1]. Reading Data

2.1 [1.1] 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 is 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 is above 3, then the recommendation wil be set to "positive". Otherwise, it will be set to "negative".

```
In [1]: %matplotlib inline
        import warnings
        warnings.filterwarnings("ignore")


        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
        from sklearn.model_selection import train_test_split
        from sklearn.decomposition import TruncatedSVD
        from sklearn.cluster import KMeans
        from sklearn.metrics.pairwise import cosine_similarity
        from wordcloud import WordCloud, STOPWORDS

        import re
        # 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

        from gensim.models import Word2Vec
        from gensim.models import KeyedVectors
        import pickle

        from tqdm import tqdm
        import os

In [2]: # using SQLite Table to read data.
        con = sqlite3.connect('../input/database.sqlite')

        # filtering only positive and negative reviews i.e.
        # not taking into consideration those reviews with Score=3
        # SELECT * FROM Reviews WHERE Score != 3 LIMIT 500000, will give top 500000 data points
        # you can change the number to any other number based on your computing power

        # filtered_data = pd.read_sql_query(""" SELECT * FROM Reviews WHERE Score != 3 LIMIT 500
        # for tsne assignment you can take 5k data points

        filtered_data = pd.read_sql_query(""" SELECT * FROM Reviews WHERE Score != 3""", con)

        # Give reviews with Score>3 a positive rating(1), and reviews with a score<3 a negative
        def partition(x):
            if x < 3:
                return 0
            return 1

        #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
        print("Number of data points in our data", filtered_data.shape)
        filtered_data.head(3)

Number of data points in our data (525814, 10)


Out[2]:    Id                         ...
        0   1                         ...                         I have bought several of the V
        1   2                         ...                         Product arrived labeled as Jum
        2   3                         ...                         This is a confection that has

        [3 rows x 10 columns]

In [3]: display = pd.read_sql_query("""
        SELECT UserId, ProductId, ProfileName, Time, Score, Text, COUNT(*)
        FROM Reviews
        GROUP BY UserId
        HAVING COUNT(*)>1
        """, con)

In [4]: print(display.shape)
        display.head()

(80668, 7)


Out[4]:                UserId   ...    COUNT(*)
        0  #oc-R115TNMSPFT9I7   ...           2
        1  #oc-R11D9D7SHXIJB9   ...           3
        2  #oc-R11DNU2NBKQ23Z   ...           2
        3  #oc-R11O5J5ZVQE25C   ...           3
        4  #oc-R12KPBODL2B5ZD   ...           2

        [5 rows x 7 columns]
```

```
In [5]: display[display['UserId']=='AZY10LLTJ71NX']

Out[5]:               UserId   ...    COUNT(*)
        80638  AZY10LLTJ71NX   ...           5

        [1 rows x 7 columns]
```

```
In [6]: display['COUNT(*)'].sum()

Out[6]: 393063
```

3 [2] Exploratory Data Analysis

3.1 [2.1] 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 [7]: display= pd.read_sql_query("""
        SELECT *
        FROM Reviews
        WHERE Score != 3 AND UserId="AR5J8UI46CURR"
        ORDER BY ProductID
        """, con)
        display.head()

Out[7]:       Id                      ...
        0   78445                     ...                           DELICIOUS WAFERS. I FIND T
        1  138317                     ...                           DELICIOUS WAFERS. I FIND T
        2  138277                     ...                           DELICIOUS WAFERS. I FIND T
        3   73791                     ...                           DELICIOUS WAFERS. I FIND T
        4  155049                     ...                           DELICIOUS WAFERS. I FIND T

        [5 rows x 10 columns]
```

As it can be seen above that same user has multiple reviews with 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 [8]: #Sorting data according to ProductId in ascending order
        sorted_data=filtered_data.sort_values('ProductId', axis=0, ascending=True, inplace=False
```

```
In [9]: #Deduplication of entries
        final=sorted_data.drop_duplicates(subset={"UserId","ProfileName","Time","Text"}, keep='f
        final.shape
```

```
Out[9]: (364173, 10)
```

```
In [10]: #Checking to see how much % of data still remains
         (final['Id'].size*1.0)/(filtered_data['Id'].size*1.0)*100
```

```
Out[10]: 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 [11]: display= pd.read_sql_query("""
         SELECT *
         FROM Reviews
         WHERE Score != 3 AND Id=44737 OR Id=64422
         ORDER BY ProductID
         """, con)

         display.head()
```

```
Out[11]:       Id                         ...                             
         0  64422                         ...                            My son loves spaghetti so
         1  44737                         ...                            It was almost a 'love at f

         [2 rows x 10 columns]
```

```
In [12]: final=final[final.HelpfulnessNumerator<=final.HelpfulnessDenominator]
```

```
In [13]: #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[13]: 1    307061
         0     57110
         Name: Score, dtype: int64
```

4 [3] Preprocessing

4.1 [3.1]. Preprocessing Review Text

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 [14]: # printing some random reviews
         sent_0 = final['Text'].values[0]
         print(sent_0)
         print("="*50)

         sent_1000 = final['Text'].values[1000]
         print(sent_1000)
         print("="*50)

         sent_1500 = final['Text'].values[1500]
         print(sent_1500)
         print("="*50)

         sent_4900 = final['Text'].values[4900]
         print(sent_4900)
         print("="*50)
```

```
this witty little book makes my son laugh at loud. i recite it in the car as we're driving along
==================================================
I was really looking forward to these pods based on the reviews.  Starbucks is good, but I prefe
==================================================
Great ingredients although, chicken should have been 1st rather than chicken broth, the only thi
==================================================
Can't do sugar.  Have tried scores of SF Syrups.  NONE of them can touch the excellence of this
==================================================
```

```
In [15]: # remove urls from text python: https://stackoverflow.com/a/40823105/4084039
         sent_0 = re.sub(r"http\S+", "", sent_0)
         sent_1000 = re.sub(r"http\S+", "", sent_1000)
```

```
        sent_150 = re.sub(r"http\S+", "", sent_1500)
        sent_4900 = re.sub(r"http\S+", "", sent_4900)

        print(sent_0)
```

```
this witty little book makes my son laugh at loud. i recite it in the car as we're driving along
```

```
In [16]: # https://stackoverflow.com/questions/16206380/python-beautifulsoup-how-to-remove-all-t
        from bs4 import BeautifulSoup

        soup = BeautifulSoup(sent_0, 'lxml')
        text = soup.get_text()
        print(text)
        print("="*50)

        soup = BeautifulSoup(sent_1000, 'lxml')
        text = soup.get_text()
        print(text)
        print("="*50)

        soup = BeautifulSoup(sent_1500, 'lxml')
        text = soup.get_text()
        print(text)
        print("="*50)

        soup = BeautifulSoup(sent_4900, 'lxml')
        text = soup.get_text()
        print(text)
```

```
this witty little book makes my son laugh at loud. i recite it in the car as we're driving along
==================================================
I was really looking forward to these pods based on the reviews.  Starbucks is good, but I prefe
==================================================
Great ingredients although, chicken should have been 1st rather than chicken broth, the only thi
==================================================
Can't do sugar.  Have tried scores of SF Syrups.  NONE of them can touch the excellence of this
```

```
In [17]: # https://stackoverflow.com/a/47091490/4084039
        import re

        def decontracted(phrase):
            # specific
            phrase = re.sub(r"won't", "will not", phrase)
            phrase = re.sub(r"can\'t", "can not", phrase)

            # general
            phrase = re.sub(r"n\'t", " not", phrase)
        phrase = re.sub(r"\'re", " are", phrase)
        phrase = re.sub(r"\'s", " is", phrase)
        phrase = re.sub(r"\'d", " would", phrase)
        phrase = re.sub(r"\'ll", " will", phrase)
        phrase = re.sub(r"\'t", " not", phrase)
        phrase = re.sub(r"\'ve", " have", phrase)
        phrase = re.sub(r"\'m", " am", phrase)
        return phrase
```

```
In [18]: sent_1500 = decontracted(sent_1500)
         print(sent_1500)
         print("="*50)
```

```
Great ingredients although, chicken should have been 1st rather than chicken broth, the only thi
==================================================
```

```
In [19]: #remove words with numbers python: https://stackoverflow.com/a/18082370/4084039
         sent_0 = re.sub("\S*\d\S*", "", sent_0).strip()
         print(sent_0)
```

```
this witty little book makes my son laugh at loud. i recite it in the car as we're driving along
```

```
In [20]: #remove spacial character: https://stackoverflow.com/a/5843547/4084039
         sent_1500 = re.sub('[^A-Za-z0-9]+', ' ', sent_1500)
         print(sent_1500)
```

```
Great ingredients although chicken should have been 1st rather than chicken broth the only thing
```

```
In [21]: # https://gist.github.com/sebleier/554280
         # we are removing the words from the stop words list: 'no', 'nor', 'not'
         # <br /><br /> ==> after the above steps, we are getting "br br"
         # we are including them into stop words list
         # instead of <br /> if we have <br/> these tags would have revmoved in the 1st step

         stopwords= set(['br', 'the', 'i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves
                     "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him',
                     'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 't
                     'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "th
                     'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'ha
                     'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as'
                     'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through'
                     'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'ov
                     'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any
                     'most', 'other', 'some', 'such', 'only', 'own', 'same', 'so', 'than', 'too'
                     's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'no
                     've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't",
                    "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'migh
                    "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'w
                    'won', "won't", 'wouldn', "wouldn't"])
```

```
In [22]: # Combining all the above stundents
         from tqdm import tqdm
         preprocessed_reviews = []
         # tqdm is for printing the status bar
         for sentance in tqdm(final['Text'].values):
             sentance = re.sub(r"http\S+", "", sentance)
             sentance = BeautifulSoup(sentance, 'lxml').get_text()
             sentance = decontracted(sentance)
             sentance = re.sub("\S*\d\S*", "", sentance).strip()
             sentance = re.sub('[^A-Za-z]+', ' ', sentance)
             # https://gist.github.com/sebleier/554280
             sentance = ' '.join(e.lower() for e in sentance.split() if e.lower() not in stopwor
             preprocessed_reviews.append(sentance.strip())
```

```
100%|| 364171/364171 [02:20<00:00, 2589.41it/s]
```

```
In [23]: preprocessed_reviews[1500]
```

```
Out[23]: 'great ingredients although chicken rather chicken broth thing not think belongs canola
```

[3.2] Preprocessing Review Summary

```
In [24]: ## Similartly you can do preprocessing for review summary also.
         def concatenateSummaryWithText(str1, str2):
             return str1 + ' ' + str2

         preprocessed_summary = []
         # tqdm is for printing the status bar
         for sentence in tqdm(final['Summary'].values):
             sentence = re.sub(r"http\S+", "", sentence)
             #sentence = BeautifulSoup(sentence, 'lxml').get_text()
             sentence = decontracted(sentence)
             sentence = re.sub("\S*\d\S*", "", sentence).strip()
             sentence = re.sub('[^A-Za-z]+', ' ', sentence)
             # https://gist.github.com/sebleier/554280
             sentence = ' '.join(e.lower() for e in sentence.split() if e.lower() not in stopwor
             preprocessed_summary.append(sentence.strip())

         preprocessed_reviews = list(map(concatenateSummaryWithText, preprocessed_reviews, prepr
         final['CleanedText'] = preprocessed_reviews
         final['CleanedText'] = final['CleanedText'].astype('str')

         del preprocessed_reviews
         del preprocessed_summary
```

```
        del sorted_data
        del filtered_data

100%|| 364171/364171 [00:08<00:00, 43860.74it/s]
```

5 [4] Featurization

5.1 [4.1] BAG OF WORDS

```
In [25]: # #BoW
         # count_vect = CountVectorizer() #in scikit-learn
         # count_vect.fit(preprocessed_reviews)
         # print("some feature names ", count_vect.get_feature_names()[:10])
         # print('='*50)

         # final_counts = count_vect.transform(preprocessed_reviews)
         # print("the type of count vectorizer ",type(final_counts))
         # print("the shape of out text BOW vectorizer ",final_counts.get_shape())
         # print("the number of unique words ", final_counts.get_shape()[1])
```

5.2 [4.2] Bi-Grams and n-Grams.

```
In [26]: # #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))
         # # please do read the CountVectorizer documentation http://scikit-learn.org/stable/mod

         # # you can choose these numebrs min_df=10, max_features=5000, of your choice
         # count_vect = CountVectorizer(ngram_range=(1,2), min_df=10, max_features=5000)
         # final_bigram_counts = count_vect.fit_transform(preprocessed_reviews)
         # print("the type of count vectorizer ",type(final_bigram_counts))
         # print("the shape of out text BOW vectorizer ",final_bigram_counts.get_shape())
         # print("the number of unique words including both unigrams and bigrams ", final_bigram
```

5.3 [4.3] TF-IDF

```
In [32]: #Using only 250k points
         min_final = final.sample(n=250000)

         x_train = min_final['CleanedText']

         tfidf_model = TfidfVectorizer(min_df=30)
         x_train_tfidf = tfidf_model.fit_transform(x_train)
         print("Shape of Data after fitting : {}".format(x_train_tfidf.shape))
         print("Some sample features(unique words in the corpus)",tfidf_model.get_feature_names(
Shape of Data after fitting : (250000, 11531)
Some sample features(unique words in the corpus) ['aa', 'aback', 'abandon', 'abandoned', 'abdomi
```

5.4 [4.4] Word2Vec

```
In [ ]: # # Train your own Word2Vec model using your own text corpus
        # i=0
        # list_of_sentance=[]
        # for sentance in preprocessed_reviews:
        #     list_of_sentance.append(sentance.split())
```

```
In [ ]: # # Using Google News Word2Vectors

        # # 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.


        # # http://kavita-ganesan.com/gensim-word2vec-tutorial-starter-code/#.W17SRFAzZPY
        # # you can comment this whole cell
        # # or change these varible according to your need

        # is_your_ram_gt_16g=False
        # want_to_use_google_w2v = False
        # want_to_train_w2v = True

        # if want_to_train_w2v:
        #     # min_count = 5 considers only words that occured atleast 5 times
        #     w2v_model=Word2Vec(list_of_sentance,min_count=5,size=50, workers=4)
        #     print(w2v_model.wv.most_similar('great'))
        #     print('='*50)
        #     print(w2v_model.wv.most_similar('worst'))

        # elif want_to_use_google_w2v and is_your_ram_gt_16g:
        #     if os.path.isfile('GoogleNews-vectors-negative300.bin'):
        #         w2v_model=KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bi
        #         print(w2v_model.wv.most_similar('great'))
        #         print(w2v_model.wv.most_similar('worst'))
        #     else:
        #         print("you don't have gogole's word2vec file, keep want_to_train_w2v = True, t
```

```
In [ ]: # w2v_words = list(w2v_model.wv.vocab)
```

```
        # print("number of words that occured minimum 5 times ",len(w2v_words))
        # print("sample words ", w2v_words[0:50])
```

5.5 [4.4.1] Converting text into vectors using Avg W2V, TFIDF-W2V

[4.4.1.1] Avg 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 tqdm(list_of_sentance): # for each review/sentence
        #     sent_vec = np.zeros(50) # as word vectors are of zero length 50, you might need to
        #     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
        #         if word in w2v_words:
        #             vec = w2v_model.wv[word]
        #             sent_vec += vec
        #             cnt_words += 1
        #     if cnt_words != 0:
        #         sent_vec /= cnt_words
        #     sent_vectors.append(sent_vec)
        # print(len(sent_vectors))
        # print(len(sent_vectors[0]))
```

[4.4.1.2] TFIDF weighted W2v

```
In [ ]: # # S = ["abc def pqr", "def def def abc", "pqr pqr def"]
        # model = TfidfVectorizer()
        # tf_idf_matrix = model.fit_transform(preprocessed_reviews)
        # # we are converting a dictionary with word as a key, and the idf as a value
        # dictionary = dict(zip(model.get_feature_names(), list(model.idf_)))

In [ ]: # # TF-IDF weighted Word2Vec
        # tfidf_feat = model.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 tqdm(list_of_sentance): # 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
        #         if word in w2v_words and word in tfidf_feat:
        #             vec = w2v_model.wv[word]
        # #             tf_idf = tf_idf_matrix[row, tfidf_feat.index(word)]
        #             # to reduce the computation we are
        #             # dictionary[word] = idf value of word in whole courpus
        #             # sent.count(word) = tf valeus of word in this review
        #             tf_idf = dictionary[word]*(sent.count(word)/len(sent))
#             sent_vec += (vec * tf_idf)
#             weight_sum += tf_idf
#     if weight_sum != 0:
#         sent_vec /= weight_sum
#     tfidf_sent_vectors.append(sent_vec)
#     row += 1
```

6 [5] Assignment 11: Truncated SVD

```
<li><strong>Apply Truncated-SVD on only this feature set:</strong>
    <ul>
        <li><font color='red'>SET 2:</font>Review text, preprocessed one converted into vectors
<br>
<li><strong>Procedure:</strong>
    <ul>
<li>Take top 2000 or 3000 features from tf-idf vectorizers using idf_ score.</li>
<li>You need to calculate the co-occurrence matrix with the selected features (Note: X.X^T
```

doesn't give the co-occurrence matrix, it returns the covariance matrix, check these bolgs blog-1, blog-2 for more information)

```
        <li>You should choose the n_components in truncated svd, with maximum explained
```

variance. Please search on how to choose that and implement them. (hint: plot of cumulative explained variance ratio)

```
        <li>After you are done with the truncated svd, you can apply K-Means clustering and choo
```

the best number of clusters based on elbow method.

```
        <li> Print out wordclouds for each cluster, similar to that in previous assignment. </li
        <li>You need to write a function that takes a word and returns the most similar words us
```

cosine similarity between the vectors(vector: a row in the matrix after truncatedSVD)

```
    </ul>
</li>
<br>
```

6.1 Truncated-SVD

6.1.1 [5.1] Taking top features from TFIDF, SET 2

In [33]:
```
# Get the feature names from TFIDF Model
tfidf_features = tfidf_model.get_feature_names()

# Fetch the idf scores from the model for each feature
idf_scores = tfidf_model.idf_
```

```
# Sort the array of (features, idf_ score) based on idf_score in descending manner
value_zips = sorted(zip(tfidf_features, idf_scores), key=lambda tup: tup[1], reverse=Tr

# Take top 2500 features from the above zip
top_threshold_value = 2500
top_features = value_zips[:top_threshold_value]
top_ft_values = list(map(lambda x: x[0], top_features))
print(" # Printing the top {} features from the model.\n\n".format(top_threshold_value)
print(top_ft_values)
```

```
 # Printing the top 2500 features from the model.


[illegible]
```

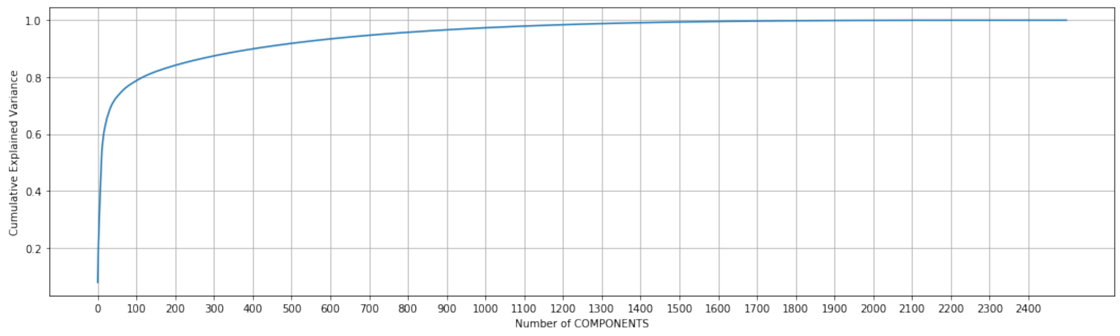
6.1.2 [5.2] Calulation of Co-occurrence matrix

In [34]:
```
# Initializing the co-occurrence matrix with 0
co_occ_matrix = np.zeros((top_threshold_value, top_threshold_value))

# Building the word vectors
li_word_vectors = []
for review in tqdm(x_train, ascii=True, desc="Building out Word Vectors"):
    words = review.split(" ")
    li_word_vectors.append(words)

# Building Co-Occurrence Matrix
for sentences in tqdm(li_word_vectors, ascii=True, desc="Building Co-occurrence Matrix"
    for i in range(0, top_threshold_value):
        feature_i = top_ft_values[i]
        if feature_i in sentences:
            for j in range(i+1, top_threshold_value):
                feature_j = top_ft_values[j]

                if i != j:
                    count = co_occ_matrix[i][j]
                    if feature_j in sentences:
                        indices = [i for i, x in enumerate(sentences) if x == feature_i
                        for indc in indices:
                            start = int(indc-5)
                            end = int(indc+5)
                            if start < 0:
                                start = 0
                            context_window_li = sentences[start: end]
                            if feature_j in context_window_li:
                                count += context_window_li.count(feature_j)
                    co_occ_matrix[i][j] = int(count)
                    co_occ_matrix[j][i] = int(count)

Building out Word Vectors: 100%|##########| 250000/250000 [00:03<00:00, 82716.48it/s]
Building Co-occurrence Matrix: 100%|##########| 250000/250000 [12:35<00:00, 331.08it/s]
```

6.1.3 [5.3] Finding optimal value for number of components (n) to be retained.

```
In [35]: svd = TruncatedSVD(n_components=top_threshold_value-1, n_iter = 10, random_state=42)
         svd.fit(co_occ_matrix)

         plt.figure(figsize=(18, 5))
         plt.plot(np.cumsum(svd.explained_variance_ratio_))
         plt.grid()
         plt.xticks(np.arange(0, top_threshold_value, 100))
         plt.xlabel('Number of COMPONENTS')
         plt.ylabel('Cumulative Explained Variance')
```

```
Out[35]: Text(0, 0.5, 'Cumulative Explained Variance')
```



As we can observe, around 95% variance is explained if we select 900 number of components.

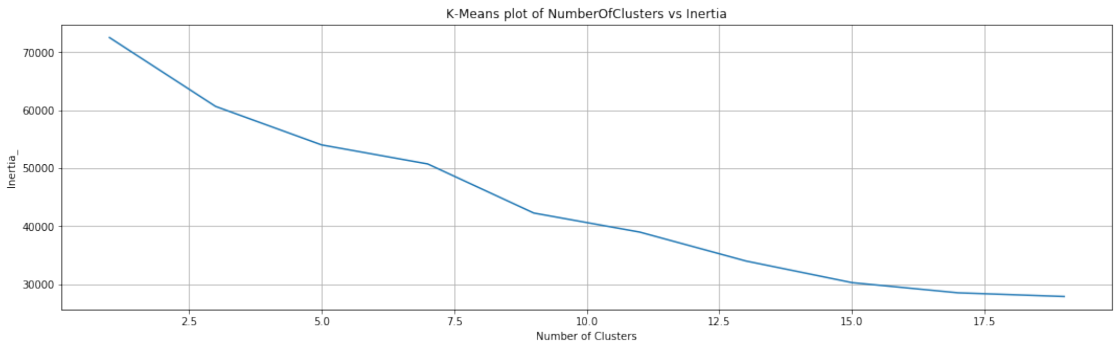
6.1.4 [5.4] Applying k-means clustering

```
In [61]: svd_clf = TruncatedSVD(n_components=900, random_state=42)
         svd_opt = svd_clf.fit_transform(co_occ_matrix)
```

```
In [62]: n_clusters = np.arange(1, 20, 2)
         save_interia_li = []

         for n_cl in tqdm(n_clusters, ascii=True, desc="K-Means Clustering"):
             clf = KMeans(n_clusters=n_cl, n_jobs=-1, random_state=42)
             clf.fit(svd_opt)
             save_interia_li.append(clf.inertia_)

         plt.close()
```

```
        plt.figure(figsize=(18, 5))
        plt.plot(n_clusters, save_interia_li)
        plt.title('K-Means plot of NumberOfClusters vs Inertia')
        plt.xlabel("Number of Clusters")
        plt.ylabel("Inertia_")
        plt.grid()
        plt.show()
```

```
K-Means Clustering: 100%|##########| 10/10 [00:23<00:00,  3.11s/it]
```



Using the elbow method, we select 9 optimal clusters to be built.

```
In [63]: optimal_k = 9
         k_clf = KMeans(n_clusters=optimal_k, n_jobs=-1)
         k_clf.fit(svd_opt)
```

```
Out[63]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
             n_clusters=9, n_init=10, n_jobs=-1, precompute_distances='auto',
             random_state=None, tol=0.0001, verbose=0)
```

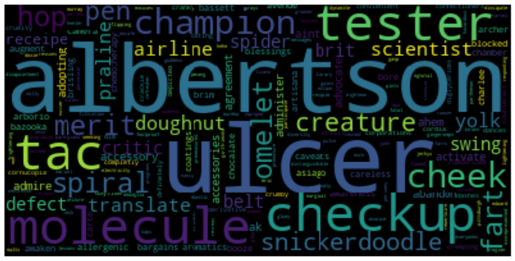
6.1.5 [5.5] Wordclouds of clusters obtained in the above section

```
In [64]: cluster_labels = k_clf.labels_
         ft_indexes = [i for i in range(top_threshold_value)]
         dict_label_value = dict()
         for (key, val) in zip(cluster_labels, ft_indexes):
             dict_label_value.setdefault(key, [])
             dict_label_value[key].append(val)

         cluster_arr = []
         sorted_labels = sorted(list(set(cluster_labels)))
         for i in sorted_labels:
             temp = []
             for ft_ind in sorted(dict_label_value[i]):
                 temp.append(top_ft_values[ft_ind])
             cluster_arr.append(temp)
```

```
In [65]: #Reference - https://blog.goodaudience.com/how-to-generate-a-word-cloud-of-any-shape-in
         def generate_wordcloud(words):
             word_cloud = WordCloud(background_color='black').generate(words)
             return word_cloud


         print("Plotting {} clusters".format(optimal_k))
         n_plots = int(list(map(sum, [divmod(optimal_k, 2)]))[0])+1
         plt.close()
         plt.figure(figsize=(12,19))
         for index, ft in enumerate(cluster_arr):
             plt.subplot(n_plots,2,index+1)
             plt.title("# Cluster: {}".format(str(int(index)+1)))
             wc = generate_wordcloud(' '.join(ft))
             plt.axis('off')
             plt.tight_layout(pad=0)
             plt.imshow(wc)
         plt.show()
```

```
Plotting 9 clusters
```

Cluster: 1



Cluster: 2

Cluster: 3

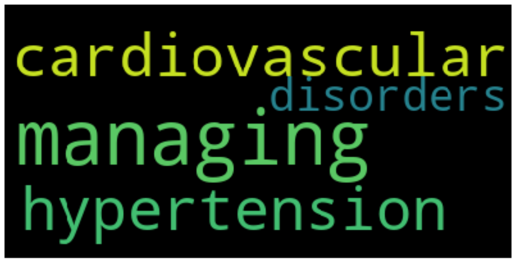
Cluster: 4

Cluster: 5

Cluster: 6

Cluster: 7

Cluster: 8



Cluster: 9

6.1.6 [5.6] Function that returns most similar words for a given word.

```
In [67]: def compute_most_similar_words(word, n):
             similarity_matrix = cosine_similarity(svd_opt)
             vector = similarity_matrix[top_ft_values.index(word)]
             sorted_ind = vector.argsort()[::-1][1:n]
             similar_words = []
             for ind in sorted_ind:
                 similar_words.append(top_ft_values[ind])
             return similar_words

         word_inp = top_ft_values[500]
         no_of_similar_words_to_compute = 10
         print("Printing Similar Words of {}".format(word_inp))

         sm_words = compute_most_similar_words(word_inp, no_of_similar_words_to_compute)
         print("\nTOP {} similar words to {} are -> {}".format(no_of_similar_words_to_compute, w
```

```
Printing Similar Words of incomplete

TOP 10 similar words to incomplete are -> ['fixes', 'whisky', 'cartoon', 'examined', 'blessings'
```

7 [6] Conclusions

- About 250k reviews from the amazon fine food reviews dataset were taken for the analysis.
- Then these reviews were converted to a matrix of TFIDF features using TFIDFVectorizer in sklearn. However, a condition was applied while vectorizing that min_df to be 30 which in turn ignored some words who appeared less than 30 times. Just to make the vectorized vocabulary rich.
- After vectorization, 2500 top features were taken from the TFIDF Vectorized model using idf_ score.
- Then, co-occurence matrix was built on those 2500 top features.
- Using the cummulative plot for explained variance, we observed that **around 95% of variance was explained by around *900 number of components* using TruncatedSVD**.
- KMeans clustering algorithm was applied on the truncated matrix and using the elbow method, **optimal number of clusters selected were 9**.
- Then, Wordclouds were built on those 9 clusters.
- Lastly, a function which would return us similar words to a word. In this, cosine similarity was applied to the truncated matrix to find out the similar words.