Team members:

-Bishal Sainju
-Victor Lee
-Supratik Chanda

Introduction:

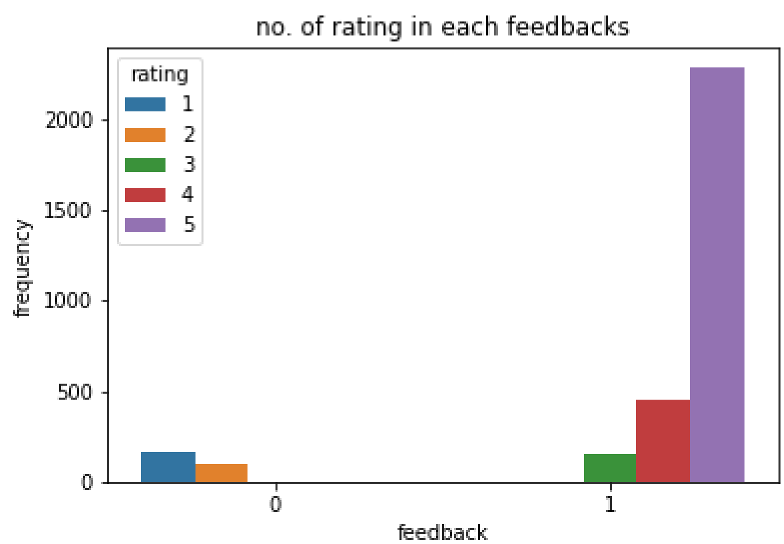
This dataset consists of 3150 Amazon customer reviews(input text), ratings(1-5), date of review, variant and feedback of various amazon Alexa products like Alexa Echo, Echo dots, Alexa Firesticks etc. for learning how to train machine for sentiment analysis(or better awesomeness analysis). We use this data to analyze Amazon's Alexa product, discover insights into consumer reviews and assist with machine learning models.

At first we thought of analyzing the sentiment of the customer reviews, but after analyzing data further, we found out that our data was biased towards positive sentiments, so whatever paraemters we chose, we wouldn't be able to create unbiased model.

Hence we decided to predict if the review is 'awesome' or 'not so awesome'. We basically assumed that if the user rating is 5 stars than, the review is awesome, and 'not so awesome', otherwise.

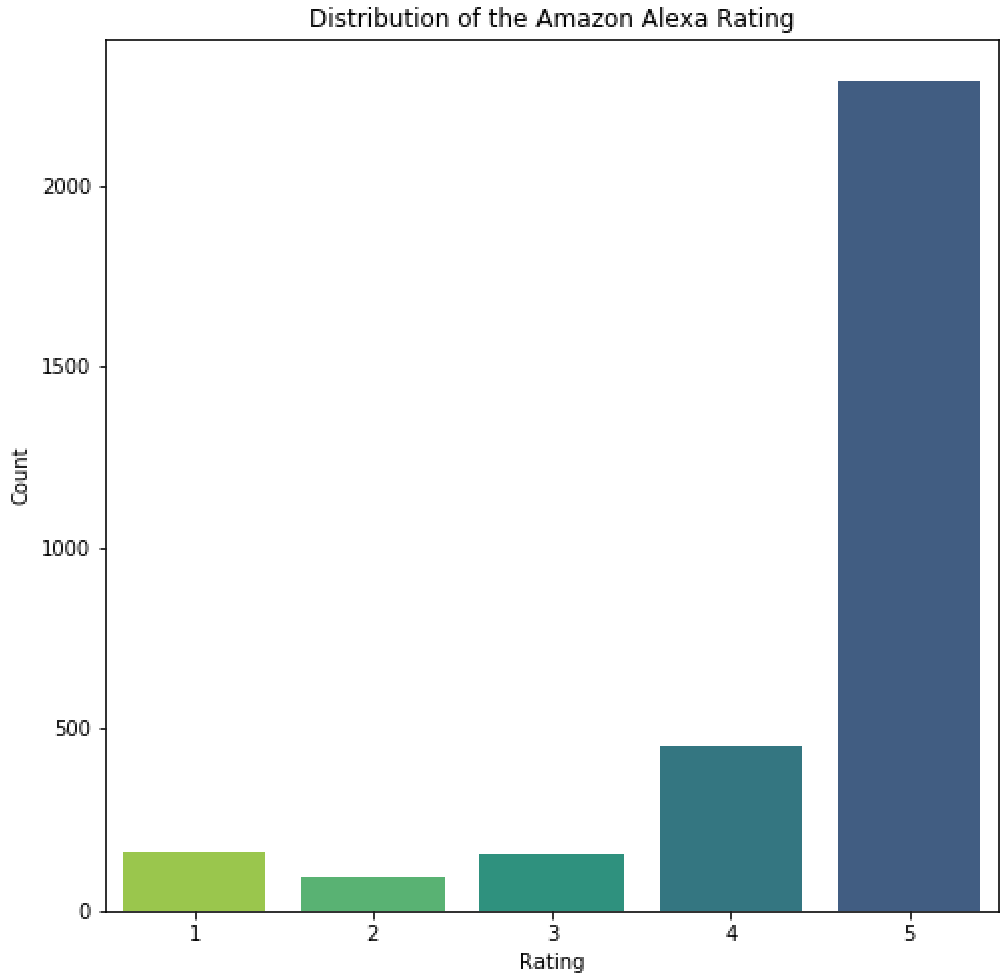
Dataset and Analyses:

1) First of all we checked to see if there was null value or not. There wasn't any, so it wasn't a problem.
2) We then went on to analyze the data. First we created a bar plot of no. of ratings in each feedback. We figured out
that if feedback was 1 it had rating 4 or 5 and if it was 0, it had rating 1, 2, or 3.



We can see that the dataset is hugely biased towards feedback 1, or in our analysis, we call it positive sentiment. So, we are pretty confident that whatever model we fit, the model will always be positive biased. Whatever new instances it gets it will try to predict it as a positive sentiment, which is not a correct represantion at all. So, instead of labeling the reviews as positive or negative sentiments, what we have decided to do is, we decided to plot a bar graph of no. of datasets that we have for each ratings.

3) We plotted bar graph of no. of records in each rating.

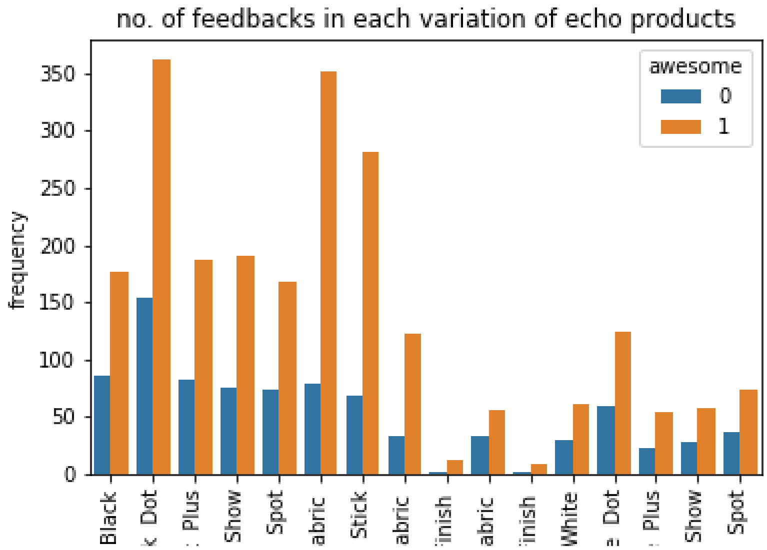


We see that there are many rating 5, so our dataset is always going to be biased towards rating '5'. However, what we can do is the divide the dataset into awesome reviews meaning rating '5' reviews vs not so awesome reviews. This is still biased but we reduce the bias to some extent.

At least we can check to see if our KNN model can perform well or not, for differentiating between 'awesome' and 'not so awesome' reviews or not. We expect that it will have hard time figuring this out but let's see if tuning in our model with appropriate parameters can do any help.

3) We had 2286 awesome reviews and 886 not awesome. So, we made another column 'awesome' with values 1 and 0.

4) Now, we went on to see if any other attributes has any significance in awesomeness of reviews.



It seems like there's not much correlation in the other variables. So, we conclude that, only review text determines awesomeness of the reviews. So, we neglect other attributes for the further analyses.

Machine Learning Preprocessing Steps:

Data Preprocessing:

We used NLTK package from NLP to preprocess the text used by the user in their reviews:
1) We first lowercased every words in the reviews of each customer.
2) Then, we tokenized each words.
3) Then, we removed the stopwords.
4) Then, we stemmed every words for efficiency in further analysis.
5) Then, we joined the words that we processed back for a user.
6) We collect all these preprocessed words for each user in corpus.

Feature Extraction:

We then extracted the feature using tf-idf vectorizer, and also created a tf-idf matrix for 500 relevant features, in the bag of words collection.

Train Model:

We then created KNN training model, and splitted our dataset into train and test sets, and then fed the train set to the model and trained our model on it, and then tested its performance on test set.

Results:

Performance Evaluation:

We then used various metrics to evaluate its performance. We obtained following result:
Confusion Matrix:

```
array([[ 96,  77],
       [118, 339]])
```

Classification Report:

```
              precision    recall  f1-score   support

           0       0.45      0.55      0.50       173
           1       0.81      0.74      0.78       457

   micro avg       0.69      0.69      0.69       630
   macro avg       0.63      0.65      0.64       630
weighted avg       0.71      0.69      0.70       630
```

Best Model:

We then used grid search to figure out the best parameters for our model. And we obtained the following result: {'algorithm': 'auto', 'leaf_size': 30, 'metric': 'jaccard', 'n_neighbors': 10, 'weights': 'distance'}

We obtained the f1-score of 0.9031 for this.
Cross validation, confusion matrix and scores for one of the folds of 10-fold cross validation is shown below:

```
              precision    recall  f1-score   support

           0       0.94      0.84      0.88        87
           1       0.94      0.98      0.96       229

   micro avg       0.94      0.94      0.94       316
   macro avg       0.94      0.91      0.92       316
weighted avg       0.94      0.94      0.94       316

[[ 73  14]
 [  5 224]]
acc:0.939873417721519, prec:0.9411764705882353, recall:0.9781659388646288, f1:0.9593147751605996
```

Conclusion:

1) So, we have got a pretty decent model than what we were expecting.
2) We have only used tf-idf vectorizer only, we could try our model using other feature extractor line, count vectorizer, binary vectorizer and so on.
3) Our model chose jaccard metric to be the best estimator, maybe because jaccard works better for sparse datasets.
4) We have just tried our model on 2 values of k-neighbors [5, 10] due to time constrain, we could try it on various ranges of K as well.
5) We could add more data preprocessing to further refine our texts(like lemmatization, dictionary comparision, POS removal)
6) Similarly, we have used 500 best features, we could further use PCA or we could play with this parameter as well. We haven't had enough time for doing this. If allowed more time, we can work on this.
7) We tried creating pipeline for trying out different things that we mentioned we couldn't do, but it took a long

In [33]:
```
import matplotlib.pyplot as plt
import re
import nltk
import pandas as pd
import numpy as np
import seaborn as sns
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix ,accuracy_score, f1_score, precision_score, recall_score

from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, chi2, f_regression, f_classif, mutual_info_classif, \
    mutual_info_regression, RFE
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
```

In [34]:
```
dataset= pd.read_csv('amazon_alexa.tsv',delimiter='\t',quoting=3)
dataset.head()
```

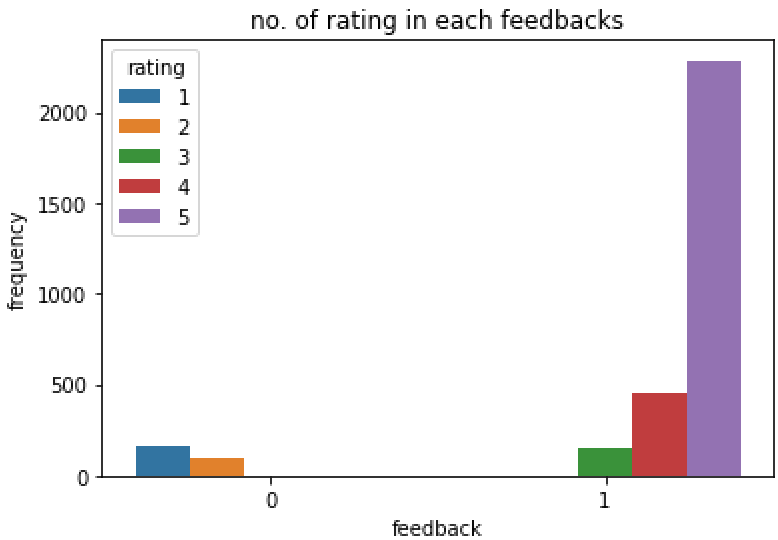
Out[34]:

	rating	date	variation	verified_reviews	feedback
0	5	31-Jul-18	Charcoal Fabric	Love my Echo!	1
1	5	31-Jul-18	Charcoal Fabric	Loved it!	1
2	4	31-Jul-18	Walnut Finish	"Sometimes while playing a game, you can answe...	1
3	5	31-Jul-18	Charcoal Fabric	"I have had a lot of fun with this thing. My 4...	1
4	5	31-Jul-18	Charcoal Fabric	Music	1

In [35]:
```
dataset.isnull().sum()
```

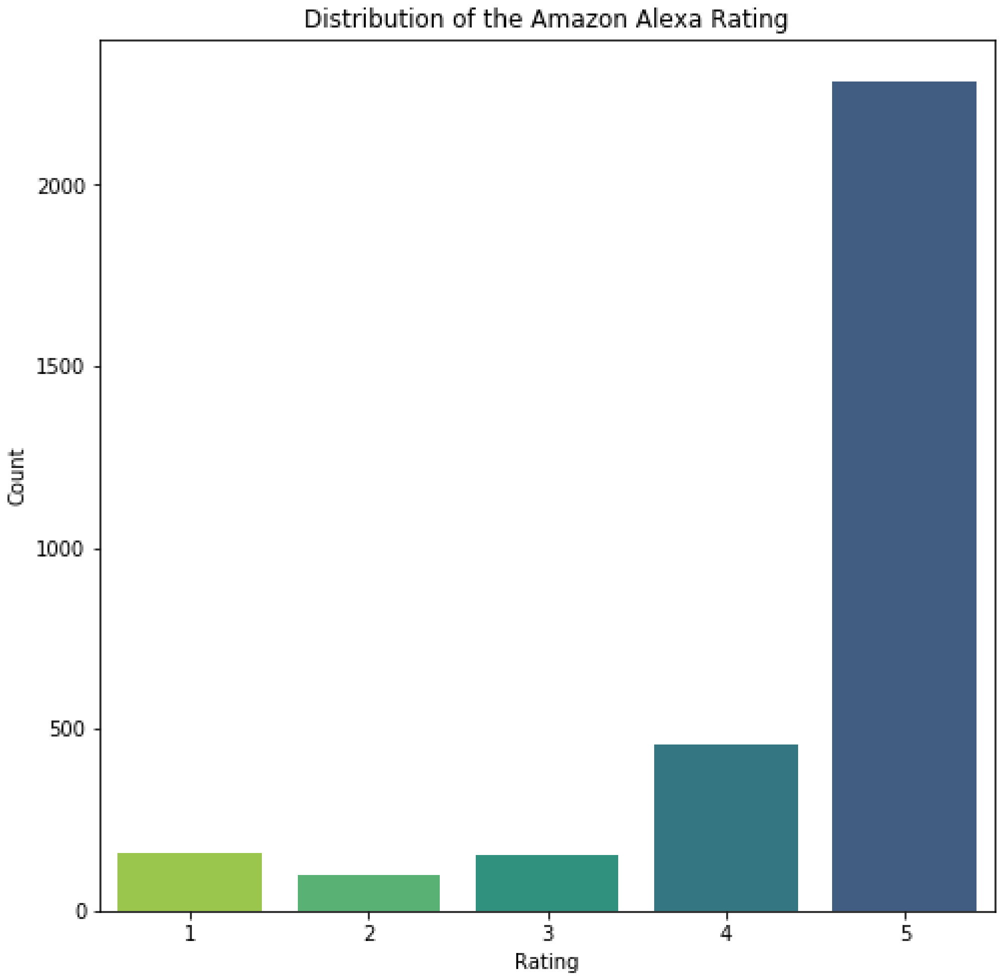
Out[35]:
```
rating              0
date                0
variation           0
verified_reviews    0
feedback            0
dtype: int64
```

In [36]:
```
temp = (dataset.groupby(['feedback']))['rating'].value_counts()\
.reset_index(name = "frequency")
sns.barplot(x = "feedback", y = "frequency", hue = "rating", data = temp
)\
.set_title("no. of rating in each feedbacks")
plt.savefig('1.png')
```



We can see that the dataset is hugely biased towards feedback 1, or in our analysis, we call it positive sentiment. So, we are pretty confident that whatever model we fit, the model will always be positive biased. Whatever new instances it gets it will try to predict it as a positive sentiment, which is not a correct represantion at all.
So, instead of labeling the reviews as positive or negative sentiments, what we have decided to do is, lets plot a bar graph of no. of datasets that we have for each ratings.

In [37]:
```
plt.figure(figsize=(8,8))
ax=sns.countplot(dataset['rating'],palette=sns.color_palette(palette="viridis_r"))
ax.set_title("Distribution of the Amazon Alexa Rating")
ax.set_xlabel("Rating")
ax.set_ylabel("Count")
plt.savefig('2.png')
```



In [38]:
```
dataset.groupby(['rating'])['rating'].count()
```

Out[38]:
```
rating
1     161
2      96
3     152
4     455
5    2286
Name: rating, dtype: int64
```

We see that there are many rating 5, so our dataset is always going to be biased towards rating '5'. However, what we can do is the divide the dataset into awesome reviews meaning rating '5' reviews vs not so awesome reviews. This is still biased but we reduce the bias to some extent.
At least we can check to see if our KNN model can perform well or not, for differentiating between 'awesome' and 'not so awesome' reviews or not. We expect that it will have hard time figuring this out but let's see if tuning in our model with appropriate parameters can do any help.

In [39]:
```
dataset.groupby('rating').count()
dataset['awesome'] = 0
dataset.loc[dataset['rating'] ==5, 'awesome'] = 1

y = dataset['awesome'].values
dataset.groupby(['awesome'])['awesome'].count()
```

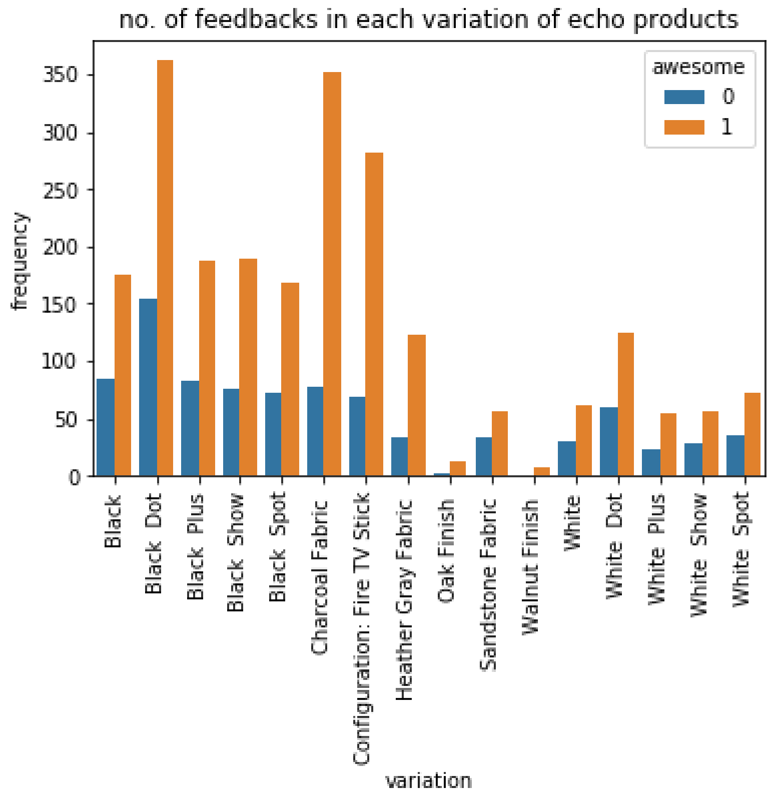
Out[39]:
```
awesome
0     864
1    2286
Name: awesome, dtype: int64
```

In [40]:
```
dataset.shape
```

Out[40]:
```
(3150, 6)
```

Let's look at other features other than the review comments.
Let's look at variations. (Does any variation have more positive feedback over others?)

In [61]:
```
temp = (dataset.groupby(['variation']))['awesome'].value_counts()\
.reset_index(name = "frequency")
sns.barplot(x = "variation", y = "frequency", hue = "awesome", data = te
mp)\
.set_title("no. of feedbacks in each variation of echo products")
plt.xticks(rotation = 90)
plt.savefig('3.png')
plt.show()
```



We don't think variation has much information in prediction whether the review is awesome or not.
We can drop date column and variations column, since they don't provide much information for further analysis.
Now, let's consider only the text used verified_reviews columns, and based on the text used in that we can predict whether those feedbacks were awesome or not.
So, for that we did some data preprocessing so that, we can get bag of words collections, which we can use to for tf-idf matrix for each record.

In [42]:
```
corpus = []
for i in range(0, 3150):
    # column : "verified_reviews", row ith
    review = re.sub('[^a-zA-Z]', ' ', dataset['verified_reviews'][i])
    # convert all cases to lower cases
    review = review.lower()
    # split to array(default delimiter is " ")
    review = review.split()
    # creating PorterStemmer object to
    # take main stem of each word
    ps = PorterStemmer()
    # loop for stemming each word
    # in string array at ith row
    review = [ps.stem(word) for word in review if not word in set(stopwo
rds.words('english'))]
    # rejoin all string array elements
    # to create back into a string
    review = ' '.join(review)
    # append each string to create
    # array of clean text
    corpus.append(review)
```

In [43]:
```
corpus[2]
```

Out[43]:
```
'sometim play game answer question correctli alexa say got wrong answer
like abl turn light away home'
```

In [44]:
```
dataset['verified_reviews'][2]
```

Out[44]:
```
'"Sometimes while playing a game, you can answer a question correctly b
ut Alexa says you got it wrong and answers the same as you.  I like bei
ng able to turn lights on and off while away from home."'
```

Let's use tf-idf vectorizer to creat a bag of word collection. And vectorize each record into tf-idf values

In [45]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
# from sklearn.feature_extraction.text import CountVectorizer

vectorizer = TfidfVectorizer(max_features=500)
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names())

print(X.shape)
print(X)
```

```
['abil', 'abl', 'absolut', 'access', 'account', 'activ', 'actual', 'a
d', 'adapt', 'add', 'addit', 'advertis', 'alarm', 'alexa', 'allow', 'al
most', 'alon', 'along', 'alreadi', 'also', 'although', 'alway', 'amaz',
'amazon', 'annoy', 'anoth', 'answer', 'anyth', 'app', 'appl', 'around',
'ask', 'assist', 'audibl', 'audio', 'avail', 'away', 'awesom', 'back',
'bad', 'base', 'basic', 'bass', 'batteri', 'bed', 'bedroom', 'bedsid',
'begin', 'best', 'better', 'big', 'birthday', 'bit', 'blue', 'bluetoot
h', 'book', 'bose', 'bought', 'brand', 'brief', 'built', 'bulb', 'butto
n', 'buy', 'cabl', 'call', 'came', 'camera', 'cannot', 'capabl', 'cel
l', 'chang', 'channel', 'chat', 'check', 'clear', 'clock', 'color', 'co
me', 'command', 'commun', 'complaint', 'complet', 'comput', 'connect',
'consid', 'constantli', 'contact', 'continu', 'control', 'conveni', 'co
ok', 'cool', 'cord', 'cost', 'could', 'coupl', 'creat', 'current', 'cus
tom', 'daili', 'daughter', 'day', 'deal', 'decid', 'definit', 'deliv',
'design', 'devic', 'differ', 'direct', 'disappoint', 'discov', 'disli
k', 'display', 'done', 'door', 'dot', 'download', 'drop', 'eas', 'eas
i', 'easier', 'easili', 'echo', 'els', 'enabl', 'end', 'enjoy', 'enoug
h', 'entertain', 'especi', 'etc', 'even', 'ever', 'everi', 'everyon',
'everyth', 'exactli', 'excel', 'except', 'excit', 'expect', 'experi',
'explor', 'extern', 'extra', 'extrem', 'face', 'fact', 'famili', 'fan',
'fantast', 'far', 'fast', 'favorit', 'featur', 'feel', 'figur', 'fina
l', 'find', 'fine', 'fire', 'firestick', 'first', 'fix', 'flash', 'forw
ard', 'found', 'free', 'friend', 'friendli', 'frustrat', 'full', 'fun',
'function', 'futur', 'game', 'gave', 'gen', 'gener', 'get', 'gift', 'gi
ve', 'glad', 'go', 'goe', 'good', 'googl', 'got', 'great', 'group', 'ha
lf', 'handi', 'happen', 'happi', 'hard', 'hear', 'heard', 'help', 'hig
h', 'highli', 'home', 'hook', 'hope', 'hour', 'hous', 'household', 'how
ev', 'hub', 'hue', 'huge', 'husband', 'immedi', 'impress', 'improv', 'i
nclud', 'inform', 'instal', 'instead', 'instruct', 'integr', 'interac
t', 'intercom', 'internet', 'issu', 'item', 'job', 'joke', 'keep', 'ki
d', 'kind', 'kitchen', 'know', 'lamp', 'later', 'learn', 'least', 'les
s', 'let', 'life', 'light', 'like', 'limit', 'link', 'list', 'listen',
'littl', 'live', 'lock', 'lol', 'long', 'longer', 'look', 'lot', 'lou
d', 'louder', 'love', 'low', 'lyric', 'made', 'mainli', 'make', 'mani',
'may', 'mini', 'minut', 'miss', 'mom', 'money', 'month', 'morn', 'mostl
i', 'mother', 'move', 'movi', 'much', 'multipl', 'music', 'must', 'nam
e', 'nd', 'need', 'netflix', 'never', 'new', 'news', 'next', 'nice', 'n
ight', 'nightstand', 'noth', 'number', 'offer', 'offic', 'often', 'ok',
'old', 'one', 'open', 'oper', 'option', 'order', 'origin', 'outlet', 'o
veral', 'packag', 'paid', 'pandora', 'part', 'pay', 'peopl', 'perfect',
'perfectli', 'perform', 'person', 'philip', 'phone', 'pick', 'pictur',
'piec', 'plan', 'play', 'playlist', 'pleas', 'plu', 'plug', 'power', 'p
retti', 'price', 'prime', 'probabl', 'problem', 'product', 'program',
'provid', 'purchas', 'put', 'qualiti', 'question', 'quick', 'quit', 'ra
dio', 'rang', 'rd', 'read', 'readi', 'real', 'realiz', 'realli', 'reaso
n', 'receiv', 'recip', 'recommend', 'refurbish', 'regret', 'regular',
'remind', 'remot', 'repeat', 'replac', 'request', 'respond', 'respons',
'return', 'review', 'right', 'ring', 'room', 'run', 'said', 'sale', 'sa
tisfi', 'save', 'say', 'schedul', 'screen', 'search', 'second', 'secu
r', 'see', 'seem', 'servic', 'set', 'setup', 'sever', 'shop', 'show',
'simpl', 'sinc', 'sit', 'size', 'skill', 'sleep', 'small', 'smaller',
'smart', 'someon', 'someth', 'sometim', 'son', 'song', 'soon', 'sound',
'space', 'speak', 'speaker', 'specif', 'spot', 'spotifi', 'st', 'stan
d', 'star', 'start', 'station', 'step', 'stick', 'still', 'stop', 'stre
am', 'stuff', 'suggest', 'super', 'support', 'suppos', 'sure', 'surpri
s', 'switch', 'sync', 'system', 'take', 'talk', 'tech', 'technolog', 't
ell', 'terribl', 'thank', 'thermostat', 'thing', 'think', 'third', 'tho
```

```
ugh', 'thought', 'three', 'throughout', 'time', 'timer', 'told', 'too
k', 'tooth', 'top', 'total', 'touch', 'tri', 'troubl', 'turn', 'tv', 't
wo', 'type', 'understand', 'unit', 'unless', 'unplug', 'updat', 'upgra
d', 'us', 'use', 'user', 'valu', 'via', 'video', 'view', 'voic', 'volu
m', 'wait', 'wake', 'walk', 'want', 'watch', 'way', 'weather', 'week',
'well', 'went', 'white', 'whole', 'wife', 'wifi', 'wireless', 'wish',
'without', 'wonder', 'word', 'work', 'worth', 'would', 'wrong', 'year',
'yet', 'youtub']
(3150, 500)
  (0, 257)	0.6475941971007908
  (0, 124)	0.7619854039818493
  (1, 257)	1.0
  (2, 399)	0.2721173567605258
  (2, 322)	0.17643204062984438
  (2, 177)	0.2818717213501252
  (2, 26)	0.5048203096711343
  (2, 339)	0.23634090310138897
  (2, 13)	0.14538413718642387
  (2, 374)	0.22565613134049495
  (2, 189)	0.21196419792333504
  (2, 496)	0.3185785821805567
  (2, 242)	0.14849721472811817
  (2, 1)	0.2236674149521693
  (2, 455)	0.22036447659800928
  (2, 241)	0.2108160937466327
  (2, 36)	0.28842350145045
  (2, 202)	0.19473749963229675
  (3, 322)	0.42039849081244574
  (3, 177)	0.3358189528253245
  (3, 242)	0.17691799272595543
  (3, 241)	0.25116403838461243
  (3, 254)	0.27322831841266265
  (3, 174)	0.24598283812508004
  (3, 438)	0.21397611953050744
  :	:
  (3148, 438)	0.1299769406847663
  (3148, 403)	0.11383775213436528
  (3148, 278)	0.10675237977967211
  (3148, 25)	0.1714942356922214
  (3148, 406)	0.12852633927799648
  (3148, 466)	0.0981110902018455
  (3148, 190)	0.09394800979762047
  (3148, 262)	0.15723226087769335
  (3148, 96)	0.2127373418492159
  (3148, 391)	0.18393308494165517
  (3148, 79)	0.1733764655496375
  (3148, 425)	0.18144430512432136
  (3148, 338)	0.29588225358044606
  (3148, 441)	0.18591710525849284
  (3148, 185)	0.15882105608947902
  (3148, 170)	0.20289327760520562
  (3148, 173)	0.1960234682397002
  (3148, 117)	0.2537161551800617
  (3148, 203)	0.42003583483813506
  (3148, 429)	0.20511792460126888
  (3148, 231)	0.22424714306846652
  (3148, 81)	0.2156745439932865
```

```
  (3148, 272)	0.2223649132110504
  (3148, 34)	0.21417656306141739
  (3149, 187)	1.0
```

In [46]:
```
###Creating the bag of words model
X = X.toarray()
# y = dataset['awesome'].values

X.shape
```

Out[46]:
```
(3150, 500)
```

Now, we built a simple knn model with default parameters to check if everything is going well.

In [47]:
```
##Splitting the dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.
2, stratify=y, random_state = 42)
```

In [48]:
```
# classifier = KNeighborsClassifier(metric='jaccard')
classifier = KNeighborsClassifier()
classifier.fit(X_train, y_train)
```

Out[48]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowsk
i',
           metric_params=None, n_jobs=None, n_neighbors=5, p=2,
           weights='uniform')
```

In [49]:
```
y_test
```

Out[49]:
```
array([1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1,
0,
          1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1,
1,
          1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1,
1,
          0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1,
1,
          0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1,
1,
          1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1,
0,
          1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1,
1,
          1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1,
1,
          0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1,
1,
          1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1,
0,
          1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0,
1,
          1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1,
1,
          1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1,
1,
          1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1,
1,
          1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0,
1,
          0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1,
1,
          1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
0,
          1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1,
1,
          1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1,
1,
          1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1,
1,
          1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1,
1,
          1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1,
1,
          1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
1,
          1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1,
1,
          1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0,
1,
          0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0,
0,
          0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1,
1,
          1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
0,
          1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0])
```

In [50]:
```
###predicting the test set results
y_pred = classifier.predict(X_test)
y_pred
```

Out[50]:
```
array([1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1,
0,
       1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0,
0,
       0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1,
0,
       0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1,
1,
       0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1,
1,
       1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1,
0,
       1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0,
1,
       1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0,
1,
       0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1,
0,
       1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0,
1,
       1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0,
1,
       0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,
1,
       0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1,
1,
       1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
1,
       0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1,
0,
       1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0,
1,
       1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0,
1,
       1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1,
1,
       1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
1,
       0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
0,
       1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1,
0,
       0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1,
1,
       1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1,
1,
       1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0,
1,
       0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1,
0,
       0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0,
1,
       0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1,
1,
       1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0])
```

In [51]:
```
##Making the Confusion matrix
cm = confusion_matrix(y_test, y_pred)
cm
```

Out[51]:
```
array([[ 96,  77],
       [118, 339]])
```

In [52]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.45      0.55      0.50       173
           1       0.81      0.74      0.78       457

   micro avg       0.69      0.69      0.69       630
   macro avg       0.63      0.65      0.64       630
weighted avg       0.71      0.69      0.70       630
```

It seems like everything is going well. We can see that we have obtained f1-score of around .70ish
Now, let's find the best estimates and best parameters, by feeding this into gridsearchCV.

In [53]:
```
params = dict(n_neighbors=[5, 10],weights=['uniform', 'distance'],algori
thm=['auto'],\
              leaf_size=[30], metric=['euclidean', 'jaccard'])

grid=GridSearchCV(estimator=KNeighborsClassifier(),param_grid=params,cv=
10,n_jobs=-1,scoring='f1')
grid.fit(X, y)
print(grid.best_params_)
print(grid.best_score_)
```

```
{'algorithm': 'auto', 'leaf_size': 30, 'metric': 'jaccard', 'n_neighbor
s': 10, 'weights': 'distance'}
0.9031151225997964
```

Let's build classification report and confusion matrix to get some insight into our model.

In [54]:
```
def scores(y_true, y_pred):
    print (classification_report(y_true, y_pred))
    print (confusion_matrix(y_true, y_pred))
    print ('acc:{}, prec:{}, recall:{}, f1:{}\n\n'.format(accuracy_score
(y_true, y_pred), \
                                                          precision_score(y_
true, y_pred),
                                                          recall_score(y_tru
e, y_pred),
                                                          f1_score(y_true, y
_pred)))
    return f1_score(y_true, y_pred)
```

In [55]:
```
from sklearn.model_selection import cross_val_score
from sklearn.metrics import make_scorer
knn_final = KNeighborsClassifier(n_neighbors=10, weights='distance', met
ric='jaccard', algorithm='auto', leaf_size=30)
cv_scores = cross_val_score(knn_final, X, y, cv=10, scoring=make_scorer(
scores))
print(np.mean(cv_scores).round(4))
```

```
              precision    recall  f1-score   support

           0       0.94      0.84      0.88        87
           1       0.94      0.98      0.96       229

   micro avg       0.94      0.94      0.94       316
   macro avg       0.94      0.91      0.92       316
weighted avg       0.94      0.94      0.94       316

[[ 73  14]
 [  5 224]]
acc:0.939873417721519, prec:0.9411764705882353, recall:0.97816593886462
88, f1:0.9593147751605996


              precision    recall  f1-score   support

           0       0.67      0.21      0.32        87
           1       0.76      0.96      0.85       229

   micro avg       0.75      0.75      0.75       316
   macro avg       0.71      0.58      0.58       316
weighted avg       0.74      0.75      0.70       316

[[ 18  69]
 [  9 220]]
acc:0.7531645569620253, prec:0.7612456747404844, recall:0.9606986899563
319, f1:0.8494208494208493


              precision    recall  f1-score   support

           0       0.96      0.80      0.88        87
           1       0.93      0.99      0.96       229

   micro avg       0.94      0.94      0.94       316
   macro avg       0.94      0.90      0.92       316
weighted avg       0.94      0.94      0.93       316

[[ 70  17]
 [  3 226]]
acc:0.9367088607594937, prec:0.9300411522633745, recall:0.9868995633187
773, f1:0.9576271186440679


              precision    recall  f1-score   support

           0       0.83      0.22      0.35        87
           1       0.77      0.98      0.86       229

   micro avg       0.77      0.77      0.77       316
   macro avg       0.80      0.60      0.60       316
weighted avg       0.78      0.77      0.72       316

[[ 19  68]
 [  4 225]]
acc:0.7721518987341772, prec:0.7679180887372014, recall:0.9825327510917
```

```
03, f1:0.8620689655172412

              precision    recall  f1-score   support

           0       0.65      0.23      0.34        86
           1       0.77      0.95      0.85       229

   micro avg       0.76      0.76      0.76       315
   macro avg       0.71      0.59      0.60       315
weighted avg       0.73      0.76      0.71       315

[[ 20  66]
 [ 11 218]]
acc:0.7555555555555555, prec:0.7676056338028169, recall:0.9519650655021
834, f1:0.8499025341130605


              precision    recall  f1-score   support

           0       0.75      0.21      0.33        86
           1       0.77      0.97      0.86       229

   micro avg       0.77      0.77      0.77       315
   macro avg       0.76      0.59      0.59       315
weighted avg       0.76      0.77      0.71       315

[[ 18  68]
 [  6 223]]
acc:0.765079365079365, prec:0.7663230240549829, recall:0.97379912663755
46, f1:0.8576923076923078


              precision    recall  f1-score   support

           0       0.55      0.21      0.30        86
           1       0.76      0.93      0.84       228

   micro avg       0.74      0.74      0.74       314
   macro avg       0.65      0.57      0.57       314
weighted avg       0.70      0.74      0.69       314

[[ 18  68]
 [ 15 213]]
acc:0.7356687898089171, prec:0.7580071174377224, recall:0.9342105263157
895, f1:0.8369351669941061


              precision    recall  f1-score   support

           0       0.87      0.56      0.68        86
           1       0.85      0.97      0.91       228

   micro avg       0.86      0.86      0.86       314
   macro avg       0.86      0.76      0.79       314
weighted avg       0.86      0.86      0.85       314
```

```
[[ 48  38]
 [  7 221]]
acc:0.856687898089172, prec:0.8532818532818532, recall:0.96929824561403
51, f1:0.9075975359342916


              precision    recall  f1-score   support

           0       1.00      0.87      0.93        86
           1       0.95      1.00      0.98       228

   micro avg       0.96      0.96      0.96       314
   macro avg       0.98      0.94      0.95       314
weighted avg       0.97      0.96      0.96       314

[[ 75  11]
 [  0 228]]
acc:0.964968152866242, prec:0.9539748953974896, recall:1.0, f1:0.976445
3961456103


              precision    recall  f1-score   support

           0       1.00      0.86      0.92        86
           1       0.95      1.00      0.97       228

   micro avg       0.96      0.96      0.96       314
   macro avg       0.97      0.93      0.95       314
weighted avg       0.96      0.96      0.96       314

[[ 74  12]
 [  0 228]]
acc:0.9617834394904459, prec:0.95, recall:1.0, f1:0.9743589743589743


0.9031
```

In [ ]: