case_17.2

May 30, 2020

1 How can we predict the sentiment associated with a customer interaction?

[1]:
```
import numpy as np
import pandas as pd
import os
import nltk
from nltk import word_tokenize
from sklearn.feature_extraction.text import CountVectorizer
from collections import Counter
import matplotlib.pyplot as plt
from wordcloud import WordCloud
from pylab import rcParams
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
rcParams['figure.figsize'] = 30, 60

%matplotlib inline
```

1.1 Introduction (5 mts)

Business Context. You are a data scientist for a large e-commerce firm. You have tens of thousands of customers writing reviews on products each day. Each review contains textual feedback along with a 1-to-5 star rating system (1 being least satisfied and 5 being most satisfied). You also have a customer support team which interacts with customers over call and messaging services. Your company also collects feedback about your customers' experiences with the website interaction after each purchase. Neither this feedback nor the messaging service have a rating number. The firm wants to quantify customer satisfaction coming from these non-rated interactions in order to help with further business decisions (e.g. determine how well your various customer service agents are doing).

Business Problem. Your task is to **build models which can identify the sentiment (positive or negative) of each of these non-rated interactions.**

Analytical Context. The data is a set of reviews in CSV file format. We will combine what we learned about text processing and classification models to develop algorithms capable of classifying interactions by sentiment.

The case is structured as follows: you will 1) read and analyze the input text data and the corresponding response variables (ratings); 2) perform basic pre-processing to prepare the data for modeling; 3) learn and apply various ways of featurizing the reviews text; and finally 4) build machine learning models to classify text as either exhibiting positive or negative sentiment (1 or 0).

1.2 Reading and performing basic analysis of the data (15 mts)

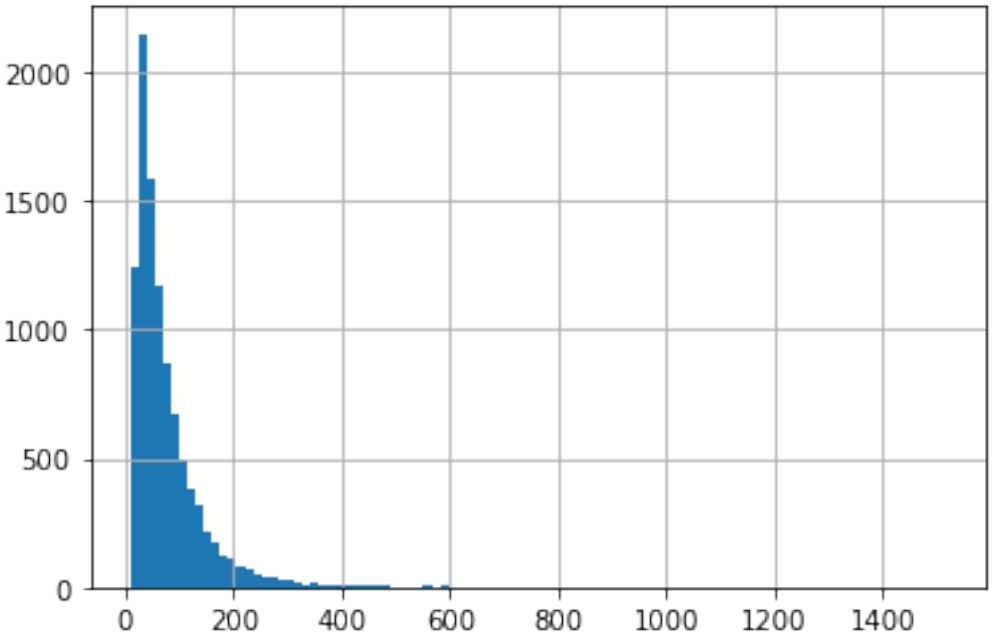
As usual the first step is to read the available data and perform some high-level analysis on it:

[2]:
```
amazon_reviews = pd.read_csv('Reviews.csv')

## Selecting just 10,000 records for faster computation.
##Feel free to comment the following line of code later, to build ML models␣
 ↪using all the data

amazon_reviews = amazon_reviews[:10000]

## Updating the column names to not have any dots (.)
amazon_reviews.columns = [each.replace(".","_") for each in amazon_reviews.
 ↪columns]
```

Let's look at the distribution of number of words per review:

[3]:
```
## Getting the number of words by splitting them by a space
words_per_review = amazon_reviews.Text.apply(lambda x: len(x.split(" ")))
words_per_review.hist(bins = 100)
```

[3]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x106d3e470>
```



[4]:
```
words_per_review.mean()
```

[4]:
```
77.9028
```

Let's also look at the distribution of ratings:

[5]:
```
amazon_reviews.Score.value_counts()
```

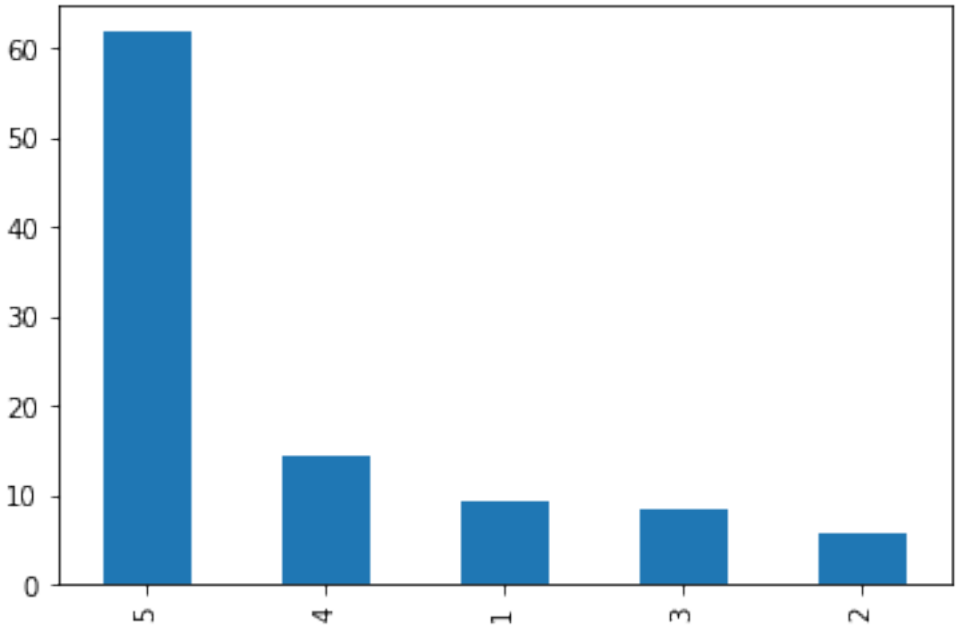
[5]:
```
5    6183
4    1433
1     932
3     862
2     590
Name: Score, dtype: int64
```

[6]:
```
percent_val = 100 * amazon_reviews.Score.value_counts()/amazon_reviews.shape[0]
percent_val
```

[6]:
```
5    61.83
4    14.33
1     9.32
3     8.62
2     5.90
Name: Score, dtype: float64
```

[7]:
```
percent_val.plot.bar()
```

[7]: <matplotlib.axes._subplots.AxesSubplot at 0x106084550>



The distribution is quite skewed, with a giant number of 5s and very few 3s, 2s, and 1s.

1.2.1 Exercise 1: (5 mts)

Create a word cloud for the product reviews.

Answer. One possible solution is shown below:

[8]:
```
word_cloud_text = ''.join(amazon_reviews.Text[:10000])
print(len(word_cloud_text))
wordcloud = WordCloud(max_font_size=100, max_words=100,␣
 ↪background_color="white",\
                          scale = 10,width=800, height=400).
 ↪generate(word_cloud_text)
plt.figure()
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```

```
4157740
```

The word cloud indicates that many of the reviews talk about food-related things - coffee, flavor, food, taste, drink. We also see some positive words like good, love, best.

1.2.2 Standardizing the ratings for sentiment analysis (5 mts)

For the purposes of sentiment analysis, we will convert all of the ratings into binary values using the follow rule: ratings of 4 or 5 will get mapped to 1, ratings of 1 or 2 will get mapped to 0, and ratings of 3 will get removed.

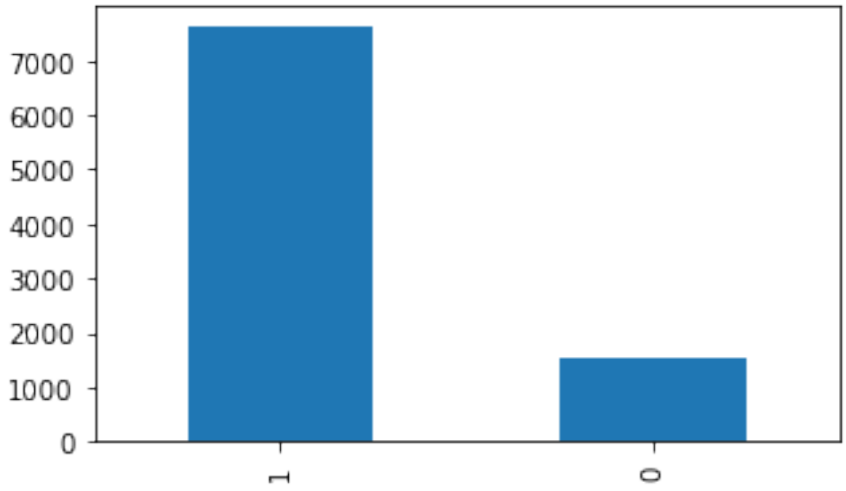
[9]:
```
amazon_reviews['Sentiment_rating'] = np.where(amazon_reviews.Score > 3,1,0)
amazon_reviews['Sentiment_rating'].value_counts()
```

[9]:
```
1    7616
0    2384
Name: Sentiment_rating, dtype: int64
```

[10]:
```
## Removing neutral reviews
amazon_reviews = amazon_reviews[amazon_reviews.Score != 3]
```

[11]:
```
rcParams['figure.figsize'] = 5, 3
amazon_reviews.Sentiment_rating.value_counts().plot.bar()
```

[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11f62b7b8>
```



1.3 Pre-processing (25 mts)

As discussed previously, text preprocessing and normalization is crucial before building a proper NLP model. Some of the important steps are:

1. converting words to lower/upper case
2. removing special characters
3. removing stopwords and high/low-frequency words
4. stemming/lemmatization

You should know most of these already, although number 4 is new. Let's proceed in order. Let's start by converting all of the words into a consistent case format, say lowercase:

[12]:
```
amazon_reviews['reviews_text_new'] = amazon_reviews.Text.apply(lambda x: x.
 ↪lower())
```

[13]:
```
from nltk import word_tokenize

token_lists = [word_tokenize(each) for each in amazon_reviews.Text]
tokens = [item for sublist in token_lists for item in sublist]
print("Number of unique tokens then: ",len(set(tokens)))

token_lists_lower = [word_tokenize(each) for each in amazon_reviews.
 ↪reviews_text_new]
tokens_lower = [item for sublist in token_lists_lower for item in sublist]
print("Number of unique tokens now: ",len(set(tokens_lower)))
```

```
Number of unique tokens then:  27884
Number of unique tokens now:  22852
```

The number of tokens has gone down by ~18% just from normalizing the case.

1.3.1 Exercise 2: (5 mts)

Is removing special characters even a good idea? What are some examples of characters that would likely be safe to remove, and what are some that would not be?

Answer. Removing special characters is a subjective call, especially in cases like this one. People often use special characters to express their emotions and might leave a review like *'This product is the worst!!!'*, while a positive review could be like *'This product is the best. Loved it!'* Here, the presence of exclamation marks clearly indicates something about the underlying sentiment, so removing them may well not be a good idea.

On the other hand, removing non-emotionally charged punctuation such as commas, periods, and semicolons is likely safe.

For the sake of simplicity, we will proceed by removing all of the special characters; however, it pays to keep in mind that this is something to revisit depending on the results we get later. The following gives a list of all the special characters in our dataset:

[14]:
```
### Selecting non alpha numeric charactes that are not spaces
spl_chars = amazon_reviews.reviews_text_new.apply(lambda x: [each for each in␣
 ↪list(x) if not each.isalnum() and each != ' '])

## Getting list of list into a single list
flat_list = [item for sublist in spl_chars for item in sublist]

## Unique special characters
set(flat_list)
```

[14]:
```
{'!',
 '"',
 '#',
 '$',
 '%',
 '&',
 "'",
 '(',
 ')',
 '*',
 '+',
 ',',
 '-',
 '.',
 '/',
 ':',
 ';',
 '<',
 '=',
 '>',
 '?',
```

```
 '@',
 '[',
 ']',
 '^',
 '_',
 '`',
 '{',
 '}',
 '~',
 '§',
 '®'}
```

Let's remove these special characters from the reviews:

[15]:
```
import re
review_backup = amazon_reviews.reviews_text_new.copy()
amazon_reviews.reviews_text_new = amazon_reviews.reviews_text_new.apply(lambda␣
↪x: re.sub('[^A-Za-z0-9 ]+', ' ', x))
```

We can see how our reviews change after removing these:

[16]:
```
print("Old Review:")
review_backup.values[6]
```

```
Old Review:
```

[16]:
```
"this saltwater taffy had great flavors and was very soft and chewy.  each candy
was individually wrapped well.  none of the candies were stuck together, which
did happen in the expensive version, fralinger's.  would highly recommend this
candy!  i served it at a beach-themed party and everyone loved it!"
```

[17]:
```
print("New Review:")
amazon_reviews.reviews_text_new[6]
```

```
New Review:
```

[17]:
```
'this saltwater taffy had great flavors and was very soft and chewy   each candy
was individually wrapped well   none of the candies were stuck together  which
did happen in the expensive version  fralinger s   would highly recommend this
candy   i served it at a beach themed party and everyone loved it '
```

The number of unique tokens has dropped further:

[18]:
```
token_lists = [word_tokenize(each) for each in amazon_reviews.Text]
tokens = [item for sublist in token_lists for item in sublist]
print("Number of unique tokens then: ",len(set(tokens)))

token_lists = [word_tokenize(each) for each in amazon_reviews.reviews_text_new]
```

```
tokens = [item for sublist in token_lists for item in sublist]
print("Number of unique tokens now: ",len(set(tokens)))
```

```
Number of unique tokens then:  27884
Number of unique tokens now:  18039
```

1.3.2 Stopwords and high/low frequency words (5 mts)

As discussed before, stopwords naturally occur very frequently in the English language without adding any context specific insights. It makes sense to remove them:

[19]:
```
noise_words = []
stopwords_corpus = nltk.corpus.stopwords
eng_stop_words = stopwords_corpus.words('english')
noise_words.extend(eng_stop_words)
noise_words
```

[19]:
```
['i',
 'me',
 'my',
 'myself',
 'we',
 'our',
 'ours',
 'ourselves',
 'you',
 "you're",
 "you've",
 "you'll",
 "you'd",
 'your',
 'yours',
 'yourself',
 'yourselves',
 'he',
 'him',
 'his',
 'himself',
 'she',
 "she's",
 'her',
 'hers',
 'herself',
 'it',
 "it's",
 'its',
 'itself',
```

```
'they',
'them',
'their',
'theirs',
'themselves',
'what',
'which',
'who',
'whom',
'this',
'that',
"that'll",
'these',
'those',
'am',
'is',
'are',
'was',
'were',
'be',
'been',
'being',
'have',
'has',
'had',
'having',
'do',
'does',
'did',
'doing',
'a',
'an',
'the',
'and',
'but',
'if',
'or',
'because',
'as',
'until',
'while',
'of',
'at',
'by',
'for',
'with',
'about',
```

```
'against',
'between',
'into',
'through',
'during',
'before',
'after',
'above',
'below',
'to',
'from',
'up',
'down',
'in',
'out',
'on',
'off',
'over',
'under',
'again',
'further',
'then',
'once',
'here',
'there',
'when',
'where',
'why',
'how',
'all',
'any',
'both',
'each',
'few',
'more',
'most',
'other',
'some',
'such',
'no',
'nor',
'not',
'only',
'own',
'same',
'so',
'than',
```

```
'too',
'very',
's',
't',
'can',
'will',
'just',
'don',
"don't",
'should',
"should've",
'now',
'd',
'll',
'm',
'o',
're',
've',
'y',
'ain',
'aren',
"aren't",
'couldn',
"couldn't",
'didn',
"didn't",
'doesn',
"doesn't",
'hadn',
"hadn't",
'hasn',
"hasn't",
'haven',
"haven't",
'isn',
"isn't",
'ma',
'mightn',
"mightn't",
'mustn',
"mustn't",
'needn',
"needn't",
'shan',
"shan't",
'shouldn',
"shouldn't",
```

```
 'wasn',
 "wasn't",
 'weren',
 "weren't",
 'won',
 "won't",
 'wouldn',
 "wouldn't"]
```

1.3.3 Exercise 3: (5 mts)

Find the high- and low-frequency words, which we will define as the 1% of words that occur most often in the reviews, as well as define the 1% of words that occur least often in the reviews (after adjusting for case and special characters).

Answer. One possible solution is given below:

[20]:
```
one_percentile = int(len(set(tokens)) * 0.01)
top_1_percentile = Counter(tokens).most_common(one_percentile)
top_1_percentile[:10]
```

[20]:
```
[('the', 28122),
 ('i', 25705),
 ('and', 19980),
 ('a', 18505),
 ('it', 16143),
 ('to', 15137),
 ('of', 12067),
 ('is', 11063),
 ('this', 10530),
 ('br', 9361)]
```

[21]:
```
bottom_1_percentile = Counter(tokens).most_common()[-one_percentile:]
bottom_1_percentile[:10]
```

[21]:
```
[('pruchase', 1),
 ('slick', 1),
 ('cloured', 1),
 ('innocuous', 1),
 ('espensive', 1),
 ('marketer', 1),
 ('strofoam', 1),
 ('destroyers', 1),
 ('ruth', 1),
 ('gleaning', 1)]
```

[22]:
```
noise_words.extend([word for word,val in top_1_percentile])
noise_words.extend([word for word,val in bottom_1_percentile])
```

Stop words and high/low frequency words have now been added to `noise_words`, which will be removed from the reviews prior to training machine learning models.

1.3.4 Stemming & lemmatization (5 mts)

Now we are ready for the last part of our pre-processing - **stemming & lemmatization**.

Different forms of a word often communicate essentially the same meaning. For example, there's probably no difference in intent between a search for `shoe` and a search for `shoes`. The same word may also appear in different tenses; e.g. "run", "ran", and "running". These syntactic differences between word forms are called **inflections**. In general, we probably want to treat inflections identically when featurizing our text.

Sometimes this process is nearly-reversible and quite safe (e.g. replacing verbs with their infinitive, so that "run", "runs", and "running" all become "run"). Other times it is a bit dangerous and context-dependant (e.g. replacing superlatives with their base form, so that "good", "better", and "best" all become "good"). The more aggressive you are, the greater the potential rewards and risks. For a very aggressive example, you might choose to replace "Zeus" and "Jupiter" with "Zeus" only; this might be OK if you are summarizing myths, confusing if you are working on astronomy, and disastrous if you are working on comparative mythology.

We won't get into the details of the differences between stemming, lemmatization and other types of text normalization here, but a careful introduction can be found at: https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html.

[24]:
```
from nltk.stem import PorterStemmer, WordNetLemmatizer, LancasterStemmer
nltk.download('wordnet')
from nltk.corpus import wordnet

porter = PorterStemmer()
lancaster = LancasterStemmer()
lemmatizer = WordNetLemmatizer()
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     /Users/haris.jaliawala/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
```

Stemming algorithms work by cutting off the end or the beginning of the word, taking into account a list of common prefixes and suffixes that can be found.

On the other hand, **lemmatization** takes into consideration the morphological analysis of the words. So lemmatization takes into account the grammar of the word and tries to find the root word instead of just getting to the root word by brute force methods.

[25]:
```
print("Lancaster Stemmer")
print(lancaster.stem("trouble"))
```

```
print(lancaster.stem("troubling"))
print(lancaster.stem("troubled"))

# Provide a word to be lemmatized
print("WordNet Lemmatizer")
print(lemmatizer.lemmatize("trouble", wordnet.NOUN))
print(lemmatizer.lemmatize("troubling", wordnet.VERB))
print(lemmatizer.lemmatize("troubled", wordnet.VERB))
```

```
Lancaster Stemmer
troubl
troubl
troubl
WordNet Lemmatizer
trouble
trouble
trouble
```

It can be seen that we get a meaning root word from Lemmatizer while Stemmer just cuts out and extracts the first important part of the word.

1.4 Building our machine learning model (40 mts)

Now we have cleaned-up versions of two very important pieces of data – the actual review text and its corresponding sentiment rating:

[26]:
```
amazon_reviews[['Text','Score','Sentiment_rating']].head(5)
```

[26]:

	Text	Score	Sentiment_rating
0	I have bought several of the Vitality canned d…	5	1
1	Product arrived labeled as Jumbo Salted Peanut…	1	0
2	This is a confection that has been around a fe…	4	1
3	If you are looking for the secret ingredient i…	2	0
4	Great taffy at a great price. There was a wid…	5	1

The independent variables or model features are derived from the review text. Previously, we discussed how we can use n-grams to create features, and specifically how bag-of-words is the simplest interpretation of these n-grams, disregarding order and context entirely and only focusing on frequency/count. Let's use that as a starting point.

1.4.1 Bag-of-words (5 mts)

`CountVectorizer` is a Python class that automatically accounts for certain preprocessing steps like removing stopwords, stemming, creating n-grams, and word tokenization:

[27]:
```
###  Creating a method for stemming
from nltk.stem import PorterStemmer
```

```
stemmer = PorterStemmer()
analyzer = CountVectorizer().build_analyzer()

def stemmed_words(doc):
    return (stemmer.stem(w) for w in analyzer(doc))
```

Let's use this to create a bag of words from the reviews, excluding the noise words we identified earlier:

[28]:
```
### Creating a python object of the class CountVectorizer

bow_counts = CountVectorizer(tokenizer= word_tokenize, stop_words=noise_words,
                             ngram_range=(1,1))

bow_data = bow_counts.fit_transform(amazon_reviews.reviews_text_new)
```

```
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/sklearn/feature_extraction/text.py:300: UserWarning: Your stop_words
may be inconsistent with your preprocessing. Tokenizing the stop words generated
tokens ["'d", "'ll", "'re", "'s", "'ve", 'might', 'must', "n't", 'need', 'sha',
'wo'] not in stop_words.
  'stop_words.' % sorted(inconsistent))
```

Once the bag of words is prepared, the dataset should be divided into training and test sets:

[29]:
```
X_train_bow, X_test_bow, y_train_bow, y_test_bow =  \
                            train_test_split(bow_data,amazon_reviews.
↪Sentiment_rating,test_size = 0.2,random_state = 0)
```

[30]:
```
y_test_bow.value_counts()/y_test_bow.shape[0]
```

[30]:
```
1    0.847921
0    0.152079
Name: Sentiment_rating, dtype: float64
```

The test data contains 84% positive sentiment reviews. So, if we were to naively classify all reviews as positive, then our model would achieve an accuracy of 84%. Therefore, this is the baseline that any model we create must beat.

1.4.2 Applying logistic regression (5 mts)

Let's train the model on our training data and run the resulting model on our test data:

[31]:
```
### Training the model
lr_model_all = LogisticRegression(C = 1)
lr_model_all.fit(X_train_bow,y_train_bow)
```

```
## Predicting the output
test_pred_lr_prob = lr_model_all.predict_proba(X_test_bow)
test_pred_lr_all = lr_model_all.predict(X_test_bow)

print("F1 score: ",f1_score(y_test_bow,test_pred_lr_all))
print("Accuracy: ", accuracy_score(y_test_bow,test_pred_lr_all)* 100)
```

```
F1 score:  0.942550505050505
Accuracy:  90.04376367614879
```

```
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

[32]:
```
prob = [each[1] for each in test_pred_lr_prob]

predictions = pd.DataFrame(list(zip(amazon_reviews[amazon_reviews.index.
↪isin(X_test_bow.indices)].Text.values,
                            amazon_reviews[amazon_reviews.index.
↪isin(X_test_bow.indices)].Score.values,
                            test_pred_lr_all,
                            prob)),
                           columns =␣
↪['Review','Actual_Score','Predicted_Sentiment','Predicted_probability'])
```

[33]:
```
predictions.tail()
```

[33]:

	Review	Actual_Score	Predicted_Sentiment	Predicted_probability
1823	Reveived my item fast! It was exactly what I o…	5	0	0.010782
1824	I drink this tea every day (I'm 7 months pregn…	5	1	0.764695
1825	I was pleasantly surprised by the stronger tha…	5	1	0.999876
1826	First, let me state that I brew most of my tea…	2	0	0.003415
1827	I love this tea! Both the green and white var…	5	1	0.908194

1.4.3 Exercise 4: (5 mts)

Look at the three reviews for which the model predicted the lowest scores, and see if they are correctly predicted. If not, explain why they are wrongly predicted.

Answer. One possible solution is given below:

[34]:
```
predictions.sort_values(['Predicted_probability'], ascending = True)[:3].values
```

[34]:
```
array([['This is by far the best K-cup hot cocoa we have tried.  It has a nice
rich flavor and with the added convenience of a K-cup, it is easy to have a cup
anytime you like. Highly recommend!',
        5, 0, 4.508359868902631e-06],
       ["I've been told that the refrigerator of the future will have
herb/vegetable/salad bins inside to grow vegetables and I thought the Prepara
Herb Savor Pod looked like the perfect precursor. Wrong. The detachable bottom
comes loose continually and leaks water everywhere. Similarly, the plastic plug
at the back does NOT hold. A good idea, a nice-looking product, but poorly
executed. Refrain from purchasing -- plastic bags are MUCH, much better.",
        1, 0, 4.988728366547799e-06],
       ["This has been my favorite brand of coconut water of the ones I've
tried. They use real fruit juice for flavoring, as well as it tastes good.<br
/><br />We tried the smaller size, and they just aren't large enough to get
enough water at the gym, and are not resealable.<br /><br />Things I like about
this:<br /><br />* Perfect size for an hour long gym routine<br />* Resealable
cap(smaller one lacks this)<br />* Tasty (Like a pina colada)<br /><br
/>Negative<br /><br />* Box makes it hard to get the last drops- usually end up
squishing it up to get the last bit<br /><br />So great coconut water, perfect
size, and cheaper than buying locally at sprouts with the Amazon's 'subscribe
and save'!",
        5, 0, 4.750717739460743e-05]], dtype=object)
```

The first and third reviews are incorrectly predicted. It is unclear why the first review got a bad score (there are not any obvious negative words in there), but the third review has a number of negative words like "aren't", "not", "lacks", "negative", despite the fact that in context these words don't mean much at all. Clearly, featurizing our text as 1-grams cannot capture the overall context of the reviews.

1.4.4 Exercise 5: (10 mts)

Modify the set of features in the model to include bigrams, trigrams, and 4-grams. Don't remove the noise words defined earlier before featurizing. (Hint: set `ngram_range=(1,4)`.)

Answer. One possible solution is shown below:

[35]:
```
### Changes with respect to the previous code
### 1. Increasing the n-grams from just having 1-gram to (1-gram, 2-gram,3-gram␣
 ↪and 4-gram)
### 2. Including the stopwords in the bag of words features

bow_counts = CountVectorizer(tokenizer= word_tokenize,
                             ngram_range=(1,4))
```

```
bow_data = bow_counts.fit_transform(amazon_reviews.reviews_text_new)
```

[36]:
```
# Notice the increase in features with inclusion of stopwords
bow_data
```

[36]:
```
<9138x1261413 sparse matrix of type '<class 'numpy.int64'>'
        with 2530528 stored elements in Compressed Sparse Row format>
```

[37]:
```
X_train_bow, X_test_bow, y_train_bow, y_test_bow =  \
                            train_test_split(bow_data,amazon_reviews.
 ↪Sentiment_rating,test_size = 0.2,random_state = 0)
```

[38]:
```
### Changes to the logistic regression
### Changing from the default regularization penalty of l2 to l1
### Changing the cost parameter C to be 0.9

lr_model_all_new = LogisticRegression(C = 0.9, penalty= 'l1')
```

[39]:
```
# Training the model
lr_model_all_new.fit(X_train_bow,y_train_bow)

# Predicting the results
test_pred_lr_prob = lr_model_all_new.predict_proba(X_test_bow)
test_pred_lr_all = lr_model_all_new.predict(X_test_bow)

print("F1 score: ",f1_score(y_test_bow,test_pred_lr_all))
print("Accuracy: ", accuracy_score(y_test_bow,test_pred_lr_all)* 100)
```

```
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)

F1 score:  0.952411370169275
Accuracy:  91.84901531728666
```

The accuracy has jumped from 90% to 91.8%. This is an example of what simple hyperparameter tuning and input feature modification can do to the overall performance. We can even get interpretable features from this in terms of what contributed the most to positive and negative sentiment:

[40]:
```
lr_weights = pd.DataFrame(list(zip(bow_counts.get_feature_names(),
                                   lr_model_all_new.coef_[0])),
                          columns= ['words','weights'])

lr_weights.sort_values(['weights'],ascending = False)[:15]
```

[40]:

	words	weights
157445	be disappointed	2.671491
75472	amazing	2.329963
810997	perfect	2.288485
358331	excellent	2.260199
819823	pleased	2.205520
930961	smooth	2.185045
809921	peppermint	1.938622
717690	not bad	1.869363
305794	delicious	1.709353
1023733	the best	1.674294
474680	have not	1.615059
642547	loves	1.613441
434105	glad	1.595783
641186	love this	1.591686
867538	refreshing	1.556071

[41]:
```
lr_weights.sort_values(['weights'],ascending = False)[-15:]
```

[41]:

	words	weights
456146	grounds	-1.930999
874505	return	-1.934368
522624	i tried to	-2.007605
724475	not very	-2.023503
37882	a refund	-2.193945
1241695	worst	-2.198617
981035	swiss miss	-2.265235
316985	disappointed	-2.273972
312428	died	-2.277751
144964	awful	-2.323514
1260317	yuck	-2.854556
1010045	than this	-2.871502
719525	not for	-2.897461
724762	not worth	-3.002539
317336	disappointing	-3.441590

1.4.5 Exercise 6: (10 mts)

Perform random forests classification on our feature set just as we did above with logistic regression.

Answer. One possible solution is given below:

[42]:
```
rf_model_all = RandomForestClassifier()

### Fitting the model by providing trainining data and training reponse variable
rf_model_all.fit(X_train_bow,y_train_bow)
```

```
### After the model is fit, output is predcited for the cross validation data
test_pred_lr_prob = rf_model_all.predict_proba(X_test_bow)
test_pred_lr_all = rf_model_all.predict(X_test_bow)
```

```
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/sklearn/ensemble/forest.py:245: FutureWarning: The default value of
n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

[43]:
```
print("F1 score: ",f1_score(y_test_bow,test_pred_lr_all))
print("Accuracy: ", accuracy_score(y_test_bow,test_pred_lr_all)* 100)
```

```
F1 score:  0.9321266968325792
Accuracy:  87.69146608315098
```

This is not quite as good as logistic regression. We can get the n-grams which were most important for the predictions as follows:

[44]:
```
feature_importances = pd.DataFrame(rf_model_all.feature_importances_,
                                   index = bow_counts.get_feature_names(),
                                    columns=['importance'])
```

[45]:
```
feature_importances.sort_values(['importance'], ascending=False)[:10]
```

[45]:

	importance
disappointed	0.004863
waste	0.004682
disappointing	0.002836
return	0.002682
not worth	0.002547
terrible	0.002246
awful	0.001877
weak and	0.001819
never buy	0.001732
returned	0.001604

1.5 TF-IDF model (25 mts)

Of course, bag-of-words are not the only way to featurize text. Another method, which we briefly touched upon before, is the **Term Frequency-Inverse Document Frequency (TF-IDF)** method. This evaluates how important a word is to a document within a large collection of documents (i.e. corpus). The importance increases proportionally based on the number of times a word appears in the document but is offset by the frequency of the word in the corpus.

The TF-IDF weight is the product of two terms. The first computes the normalized Term Frequency (TF); i.e. the number of times a word appears in a document divided by the total number of words in that document. The second term is the Inverse Document Frequency (IDF), computed as the

logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears:

Let's re-featurize our original set of reviews based on TF-IDF and split the resulting features into train and test sets:

[47]:
```
### Creating a python object of the class CountVectorizer
tfidf_counts = TfidfVectorizer(tokenizer= word_tokenize,stop_words=noise_words,
                               ngram_range=(1,1))
tfidf_data = tfidf_counts.fit_transform(amazon_reviews.reviews_text_new)
```

```
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/sklearn/feature_extraction/text.py:300: UserWarning: Your stop_words
may be inconsistent with your preprocessing. Tokenizing the stop words generated
tokens ["'d", "'ll", "'re", "'s", "'ve", 'might', 'must', "n't", 'need', 'sha',
'wo'] not in stop_words.
  'stop_words.' % sorted(inconsistent))
```

[48]:
```
X_train_tfidf, X_test_tfidf, y_train_tfidf, y_test_tfidf =  \
                            train_test_split(tfidf_data,amazon_reviews.
↪Sentiment_rating,test_size = 0.2,random_state = 0)
```

1.5.1 Applying logistic regression to TF-IDF features (5 mts)

Let's apply logistic regression to the features created from TF-IDF:

[49]:
```
### Setting up the model class
lr_model_tf_idf = LogisticRegression()

## Training the model
lr_model_tf_idf.fit(X_train_tfidf,y_train_tfidf)

## Prediciting the results
test_pred_lr_prob = lr_model_tf_idf.predict_proba(X_test_tfidf)
test_pred_lr_all = lr_model_tf_idf.predict(X_test_tfidf)

## Evaluating the model
print("F1 score: ",f1_score(y_test_bow,test_pred_lr_all))
print("Accuracy: ", accuracy_score(y_test_bow,test_pred_lr_all)* 100)
```

```
F1 score:  0.9335347432024169
Accuracy:  87.96498905908096
```

```
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

Here we have attained an accuracy of 88% with TF-IDF as compared to 90% with 1-grams.

1.5.2 Exercise 7: (10 mts)

Try increasing the accuracy of the model by setting `ngram_range=(1,4)` and not removing the noise words beforehand.

Answer. One possible solution is given below:

[50]:
```
### Creating a python object of the class CountVectorizer
### Changes: Removing stop words and including 1-4 grams in the tf-idf data

tfidf_counts = TfidfVectorizer(tokenizer= word_tokenize,
                               ngram_range=(1,4))
tfidf_data = tfidf_counts.fit_transform(amazon_reviews.reviews_text_new)
```

[51]:
```
X_train_tfidf, X_test_tfidf, y_train_tfidf, y_test_tfidf =  \
                            train_test_split(tfidf_data,amazon_reviews.
 ↪Sentiment_rating,test_size = 0.2,random_state = 0)
```

[52]:
```
### Setting up the model class
lr_model_tf_idf_new = LogisticRegression(C = 1e2, penalty= 'l1')

## Training the model
lr_model_tf_idf_new.fit(X_train_tfidf,y_train_tfidf)

## Prediciting the results
test_pred_lr_prob = lr_model_tf_idf_new.predict_proba(X_test_tfidf)
test_pred_lr_all = lr_model_tf_idf_new.predict(X_test_tfidf)

## Evaluating the model`
print("F1 score: ",f1_score(y_test_bow,test_pred_lr_all))
print("Accuracy: ", accuracy_score(y_test_bow,test_pred_lr_all)* 100)
```

```
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)

F1 score:  0.9561347743165926
Accuracy:  92.45076586433261
```

This is a higher accuracy than using the bag-of-words featurization from earlier. We can also easily get the set of top 10 features that contributed the most to positive or negative sentiment:

[53]:
```
lr_weights = pd.DataFrame(list(zip(tfidf_counts.get_feature_names(),
                                   lr_model_tf_idf_new.coef_[0])),
                          columns= ['words','weights'])

lr_weights.sort_values(['weights'],ascending = False)[:10]
```

[53]:
```
                   words    weights
448809             great  93.241390
157445   be disappointed  88.296242
810997           perfect  83.623632
1023733        the best  79.431283
358331         excellent  78.007845
305794         delicious  72.766660
75472            amazing  69.382075
819823           pleased  67.082665
717690           not bad  64.098780
1236019          without  60.654889
```

[54]:
```
lr_weights.sort_values(['weights'],ascending = False)[-10:]
```

[54]:
```
                     words    weights
1241695              worst -81.408043
1010045          than this -81.890072
1037349  the great reviews -82.267350
719525             not for -82.560666
317336       disappointing -83.023326
722361       not recommend -84.135218
1260317              yuck -93.438275
522624          i tried to -93.620064
724762           not worth -97.169905
316985        disappointed -99.813757
```

1.6 Word embeddings model (20 mts)

The final type of featurization we will cover are **word embeddings**. This is a type of word representation that allows words with similar meaning to have a similar representation. It is this approach to representing words and documents that may be considered one of the key breakthroughs of deep learning on challenging natural language processing problems. It is capable of capturing the context of a word in a document, its semantic and syntactic similarity, and its relation with other words. Word embeddings are in fact a class of techniques where individual words are represented as real-valued vectors in a predefined vector space.

In the following image, each of the words have been represented in 2-dimensions for simplicity. It can be clearly seen that words with similar context are grouped together – bathroom, kitchen, bathtub are grouped together, while microwave, refrigerator, oven form another group, etc.

[56]:
```
from IPython.display import Image
from IPython.core.display import HTML
```

There are different methods to learn word embeddings - Word2Vec, GloVe, FastText. **Word2Vec** uses a shallow Neural Network and is of two types; *CBOW* and *Skip Gram*. **GloVe** is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representa-

tions showcase interesting linear substructures of the word vector space. **fastText** is a library for learning of word embeddings and text classification created by Facebook's AI Research lab.

1.6.1 Why use word embeddings over bag-of-words and TF-IDF? (5 mts)

Each word is represented by a real-valued vector, which generally has tens or hundreds of dimensions. This is in contrast to he thousands or millions of dimensions required for sparse word representations. Thus, word embeddings can drastically reduce the number of dimensions required for representing a text document:

[57]:
```
import gensim
```

[58]:
```
### Loading a pre-trained glove word embedding that is trained on twitter␣
 ↪dataset
### This word embedding is 200 dimensional in lenght

model = gensim.models.KeyedVectors.load_word2vec_format(
    os.path.join(os.getcwd(), 'glove.twitter.27B.200d_out.txt'), binary=False,
    unicode_errors='ignore')
```

We had approximately 18,000 distinct tokens for 1-gram features in the bag-of-words representation, yet will only have 200 dimensions in this word embedding. This is a huge difference!

Moreover, word embeddings capture the context and semantics of the sentences since each word vector representation is itself based on its contextual meaning.

Below is the vector representation for "food" and "great":

[59]:
```
print("The embedding for food is",len(model['food']),"dimensional")

model['food']
```

```
The embedding for food is 200 dimensional
```

[59]:
```
array([-6.9175e-01, -1.4259e-01,  3.8653e-01, -2.3141e-01, -2.0408e-01,
       -2.1565e-01,  7.7839e-01,  2.2689e-03, -7.2446e-02, -6.0134e-01,
       -4.2400e-01, -5.7140e-01, -8.4249e-01,  1.5947e-01, -1.2899e-01,
        5.9032e-01, -1.3632e-01, -6.6478e-01, -1.9557e-01, -8.2453e-01,
       -1.3177e-01,  1.3514e-01, -7.3214e-01,  4.8200e-01,  4.3505e-01,
        1.6676e+00, -1.8275e-01, -1.0007e-01,  3.7003e-01,  1.0411e-01,
       -8.8115e-01, -9.7733e-04, -2.9459e-01, -7.3869e-02, -4.0103e-01,
       -4.6626e-01,  2.3253e-01,  2.7776e-01,  4.0754e-01, -4.5051e-02,
       -1.9468e-01, -2.9230e-01, -3.4642e-01, -4.9286e-01,  1.0467e-01,
        7.2143e-01,  5.9596e-01,  5.3495e-01,  3.8788e-02, -1.4406e-01,
       -5.2248e-02, -6.8292e-01, -1.0080e-01, -1.2961e-01, -2.6006e-02,
        1.4836e-01,  3.2417e-02,  1.3997e-01,  8.3943e-03, -2.3139e-01,
       -1.8000e-01, -3.1689e-01,  2.3606e-01,  1.8237e-01,  4.3933e-01,
       -3.2313e-01, -2.1512e-03, -4.4172e-01,  4.1011e-01,  1.7174e-01,
```

```
       -8.6405e-01, -3.9674e-01,  4.4175e-01,  5.9300e-01,  1.8982e-01,
       -2.9646e-02, -3.4041e-01, -3.3708e-02,  7.3449e-01,  4.5300e-01,
       -2.7855e-02, -1.8993e-02,  3.8107e-01, -5.6606e-02,  1.4864e-02,
        3.1518e-01, -3.2304e-01, -2.7439e-01,  6.1900e-02,  3.2886e-01,
        1.5138e-01,  5.3268e-01, -1.6616e-01, -2.3076e-01, -9.6515e-02,
        4.5991e-01, -5.1475e-01,  1.0297e-01, -4.0225e-02,  5.6679e-01,
        3.1027e-01,  1.5679e-01, -2.5897e-01,  4.6312e-01,  2.2561e-01,
       -3.9300e-01, -3.9593e-01,  4.4001e-01,  3.7176e-01,  1.4747e-02,
       -1.9193e-01, -2.2478e-01, -1.2665e-01, -3.4982e-01,  5.0847e-01,
        3.1720e-01,  1.2942e-01, -6.2695e-01,  5.8675e-01,  4.1040e-02,
        1.8835e-01, -2.2626e-01, -1.1744e-01,  5.1429e-03,  7.2058e-02,
       -4.9525e-01,  4.4159e-01,  8.6225e-01,  7.6765e-02, -9.7908e-02,
        6.8383e-02,  3.0596e-01,  3.7980e-01,  1.1563e-01, -6.1020e-01,
       -6.8107e-01,  3.2723e-02,  2.5346e-01,  3.5334e-01,  2.5407e-01,
       -4.6516e-01,  4.8858e-01,  3.9032e-01, -8.1296e-01, -6.9780e-01,
       -1.2542e-01,  7.9234e-02,  1.2918e-01, -1.1048e-01,  8.9312e-03,
        3.6999e-01,  3.0116e-01, -4.6578e+00, -4.4493e-03,  2.0313e-02,
       -5.0215e-02, -2.0646e-01, -3.7321e-02, -5.1779e-02,  6.6986e-02,
       -5.8853e-01,  7.1753e-01,  4.2784e-02,  1.6667e-03, -2.6193e-01,
        5.8214e-01, -1.0513e+00, -3.0341e-02,  7.3892e-01, -1.8003e-01,
       -1.1104e-01,  3.0846e-01,  4.4027e-01, -8.4080e-02, -2.6251e-01,
       -3.8733e-01, -2.6630e-01,  1.9655e-01,  5.3812e-02, -2.4456e-01,
       -7.8868e-01, -7.1843e-01,  7.0593e-02, -1.9051e-01,  2.5553e-01,
       -1.3786e-01,  1.2942e-01,  4.5864e-01,  5.5462e-01,  8.2104e-01,
       -2.5049e-01, -3.3623e-01,  1.8491e-01, -4.8235e-01,  3.1425e-01,
        2.4499e-01, -2.4404e-01,  8.0309e-02,  3.4060e-01,  7.0451e-01],
      dtype=float32)
```

[60]:
```
print("The embedding for great is",len(model['great']),"dimensional")

model['great']
```

```
The embedding for great is 200 dimensional
```

[60]:
```
array([ 1.0751e-01,  1.5958e-01,  1.3332e-01,  1.6642e-01, -3.2737e-02,
        1.7592e-01,  7.2395e-01,  1.1713e-01, -3.5036e-01, -4.2937e-01,
       -4.0925e-01, -2.5761e-01, -1.0264e+00, -1.0014e-01,  5.5390e-02,
        2.0413e-01,  1.2807e-01, -2.6337e-02, -6.9719e-02, -3.6193e-02,
       -1.9917e-01,  3.9437e-02, -9.2358e-02,  2.6981e-01, -2.0951e-01,
        1.5455e+00, -2.8123e-01,  3.2046e-01,  4.5545e-01, -3.8841e-02,
       -1.7369e-01, -2.3251e-01, -5.9551e-02,  2.3250e-01,  4.4214e-01,
        3.3666e-01,  3.9352e-02, -1.2462e-01, -2.9317e-01, -4.8857e-02,
        6.9021e-01,  7.1279e-02,  1.0252e-01,  1.6122e-01, -2.3536e-01,
        6.2724e-02,  2.0222e-01,  5.0234e-02, -1.1611e-01,  2.8909e-02,
       -1.1109e-01, -5.0241e-02, -5.9063e-01, -8.8747e-02,  5.1444e-01,
       -1.3715e-01,  1.7194e-01, -8.3657e-02,  9.6333e-02, -9.7063e-02,
        3.4003e-03, -7.0180e-02, -5.9588e-01, -2.8264e-01,  1.2529e-01,
```

```
 2.4359e-01, -4.9082e-01, -4.2533e-02,  2.2158e-01, -2.1491e-01,
-4.2101e-02,  2.3359e-01,  3.1978e-01,  3.5063e-01,  6.1748e-01,
-1.0197e-01,  5.3357e-01, -3.6005e-01, -1.7212e-02,  1.6645e-01,
 8.9432e-01,  2.7322e-02,  3.0683e-01,  1.9715e-02,  6.0516e-01,
 4.1085e-01,  5.5945e-01, -8.4501e-02,  3.5933e-01,  1.0216e-01,
 2.6675e-01, -6.0445e-01, -1.0513e-01, -1.9248e-01,  2.9150e-01,
-1.0537e-01,  5.2671e-01,  2.3763e-01, -1.3640e-01, -6.1029e-02,
 1.0081e-01,  7.4541e-02, -1.4899e-01, -2.2301e-01, -1.3653e-02,
 4.0192e-02,  5.5821e-03, -2.9936e-02,  2.7338e-02,  5.9412e-01,
-1.0302e-01,  9.0319e-02,  3.1055e-01,  6.3336e-01,  2.9762e-01,
-8.4671e-02, -1.2552e-01, -6.3930e-01,  3.8613e-01,  6.6371e-01,
 5.1345e-01,  2.0719e-01,  2.1100e-01,  1.4579e-01, -7.3321e-02,
-7.0593e-01, -6.2578e-02, -2.5470e-01,  1.1986e-01,  1.6102e-01,
 3.2958e-02, -2.4159e-01, -2.5708e-01,  3.2051e-01, -1.1569e-01,
 6.7540e-03, -1.1688e-01, -3.6158e-02, -6.5320e-01,  4.9560e-01,
-3.9429e-02, -1.8395e-01,  2.3295e-01,  5.4128e-01,  2.4568e-02,
-1.9862e-01,  2.1041e-01,  9.3798e-02,  8.3096e-03, -6.1551e-02,
 2.3262e-01, -4.2756e-02, -5.3511e+00,  3.0604e-01,  3.3578e-01,
-3.6771e-01,  5.6225e-01, -8.2341e-02,  2.9809e-01,  2.5189e-01,
-4.6203e-01,  1.0452e-01, -3.9540e-01,  3.6961e-01,  1.3093e-01,
 1.6653e-01, -3.1915e-01,  1.6974e-01,  4.2575e-01,  3.6420e-01,
 3.7175e-01, -1.9450e-01,  6.2702e-02,  4.9775e-01,  3.1842e-02,
-6.4072e-02,  7.6183e-02, -5.9534e-01,  3.1731e-01, -2.8254e-01,
 1.5987e-01, -9.2750e-02, -4.1426e-02,  7.5799e-02,  9.5740e-03,
-2.1532e-01, -3.1419e-01, -1.5144e-01, -4.6584e-01, -1.1069e-01,
-4.0130e-01,  3.9266e-02,  8.1880e-01, -4.2955e-02,  2.1698e-01,
-6.0347e-02,  3.3431e-01, -9.9549e-02, -1.8156e-01, -8.5143e-02],
dtype=float32)
```

To find the vector for an entire review, we get the vector for each word in the review separately and take a simple average.

1.6.2 Exercise 8: (5 mts)

Calculate the vector for every single review in the dataset.

Answer. One possible solution is shown below:

[61]:
```
review_embeddings = []

for each_review in amazon_reviews.reviews_text_new:

    ## Review_average
    Review_average = np.zeros(model.vector_size)
    count_val = 0

    for each_word in word_tokenize(each_review):
```

```
# Uncomment the following sentence to ignore stopwords while calculating word␣
↪embeddings
#         if(each_word.lower() in noise_words):
#             print(each_word.lower())
#             continue

        if(each_word.lower() in model):
            Review_average =+ model[each_word.lower()]
            count_val += 1

    review_embeddings.append(list(Review_average/count_val))
```

Let's convert the list of vector representations for each review into a DataFrame and split it into train and test sets:

[62]:
```
embedding_data = pd.DataFrame(review_embeddings)
embedding_data = embedding_data.fillna(0)
```

[63]:
```
X_train_embed, X_test_embed, y_train_embed, y_test_embed =  \
                            train_test_split(embedding_data,amazon_reviews.
↪Sentiment_rating,test_size = 0.2,random_state = 0)
```

Let's now apply logistic regression to our word embeddings representation:

[64]:
```
lr_model_all.fit(X_train_embed,y_train_embed)
test_pred_lr_prob = lr_model_all.predict_proba(X_test_embed)
test_pred_lr_all = lr_model_all.predict(X_test_embed)

print("F1 score: ",f1_score(y_test_embed,test_pred_lr_all))
print("Accuracy: ", accuracy_score(y_test_embed,test_pred_lr_all)* 100)
```

```
F1 score:  0.9177027827116636
Accuracy:  84.79212253829321

/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

Unfortunately, this is not as good as either the bag-of-words or TF-IDF representations. Furthermore, although word embeddings was really effective at reducing the overall number of dimensions, it suffers from the problem of interpretability. This means that it is very hard for us to even diagnose what is causing its sub-par performance.

In our case, creating features using TF-IDF got us an accuracy of 92% with very interpretable features. This is a good combination and so we deem this the best model for us here.

1.7 Conclusions (5 mts)

In this case, we cleaned up and featurized an Amazon reviews dataset and built some classification models on these featurizations to predict sentiment. We saw that bag-of-words and TF-IDF both gave interpretable features, while word embeddings did not really. Through increase the set of n-grams we used from 1-grams to up to 4-grams, we were able to get our logistic regression model accuracy up to 92%.

1.8 Takeaways (5 mts)

Building machine learning models on text is a very involved discipline. Some important things of note are as follows:

1. Although there are different types of pre-processing involved in textual data, not everything has to be applied in each case. For instance, when dealing with text messages special characters might represent important information and need not be removed. Furthermore, upper case may mean someone is angry and represents shouting, so case may not need to be normalized either.
2. Hyperparameter tuning in machine learning models is a very important step. We can't go ahead training a model with default parameters. Different sets of parameters have to be tried to see what contributes to the best model.
3. Every NLP classification task is different, but the process to be followed is similar to what we did in this case: wrangle the data -> create features from text -> train ML models.