NLP Assignment – 1 | Deception on Amazon

Name: Tanmaiyii Rao

ID : 140361229

Part A - Deception Detection

The main dataset that we are working with is the amazon reviews. The main aim is to classify the reviews to be fake or real by implementing a Support Vector Machine (SVM). Both the review text and the additional features contained in the data set have been used to build and train the classifier on part of the data set. You will then test the accuracy of your classifier on an unseen portion of the corpus.

1. Parsing and Preprocessing

The first task is to define a function to parse the data and preprocess the review texts. For this, the functions parseReview() and preProcess(text) have been defined. In the parseReview as can be seen the Id, Text and Label have been selected from the data. Followed by preprocessing the review text. The text has been tokenised into different words and normalised while preprocessing These functions are called while reading the data from the unicode csv reader.

In [11]:
```
###############
## QUESTION 1 ##
###############

# # Convert line from input file into an id/text/label tuple
def parseReview(reviewLine):
    # Should return a triple of an integer, a string containing the review, and a string indicating the label
    Id = reviewLine[0]
    Text = reviewLine[8]
    Label = reviewLine[1]
    return (Id, Text, Label)

# TEXT PREPROCESSING

def preProcess(text):
# #     # word tokenisation
    text = re.sub(r"(\w)([.,;:!?'\"”\)])", r"\1 \2", text)
    text = re.sub(r"([.,;:!?'\"“\(])(\w)", r"\1 \2", text)
    tokens = re.split(r"\s+",text)
# #     # normalisation
    new_text = re.sub(r"(\S)\1\1+",r"\1\1\1", text)
    tokens = [t.lower() for t in tokens]
    return tokens
```

2. toFeatureVector

The next step is to implement the toFeatureVector function. Given a preprocessed review (that is, a list of tokens), this function will return a dictionary that has its keys as the tokens and values as the weight of those tokens in the preprocessed reviews. The weight has been given as the number of occurences of a token divided by the total token occurences in the preprocessed review. The global featureDict, which is the dictionary that keeps track of all the tokens in the whole review dataset has been incrementally built up.

In [12]:
```
###############
## QUESTION 2 ##
###############
featureDict = {} # A global dictionary of features
featureDict = {}
def toFeatureVector(words):
    v = {}
    for w in words:
        try:
            i = featureDict[w]
        except KeyError:
            i = len(featureDict) + 1
            featureDict[w] = i
        try:
            v[w] += (1.0/len(words))
        except KeyError:
            v[w] = (1.0/len(words))
    return v
```

3. crossValidate function

The crossValidate function has been completed to do a 10-fold cross validation. The precision_recall_fscore_support function has been used to compute the precision, recall, f1 score. The accuracy score has also been added. The f1-score gives you the harmonic mean of precision and recall. The average precision, recall, f-score and accuracy has been calculated and is stored in cv_results.

```
myTestData = []
myTrainData = []

def crossValidate(dataset, folds):
    cv_results = []
    accuracy = []
    shuffle(dataset)
    foldSize = int(len(dataset)/folds)
    for i in range(0,len(dataset),foldSize):
        # insert code here that trains and tests on the 10 folds of data in the dataset
        print ("fold start %d foldSize %d" % (i, foldSize))
        myTestData = dataset[i:i+foldSize]
        myTrainData = dataset[0:i] + dataset[i+foldSize:]
        classifier = trainClassifier(myTrainData)
        y_pred = predictLabels(myTestData, classifier)
#         review,label = zip(*myTestData)
#         y_true = label
        y_true = [x[1] for x in myTestData]
#         y_true = classifier.classify(map(lambda x: x[1], myTestData))
        cv_results.append(precision_recall_fscore_support(y_true, y_pred, average='weighted'))
        accuracy.append(accuracy_score(y_true, y_pred))

#Calclualte avergae of values over the 10-fold runs
    cv_results = np.asarray(cv_results)
    cv_results = [np.mean(cv_results[:,0]), np.mean(cv_results[:,1]), np.mean(cv_results[:,2])]

    accuracy = np.asarray(accuracy)
    accuracy = np.mean(accuracy)
    cv_results.append(accuracy)

    return cv_results
```

After implementing the above functions, the loadData function is called to load the data. Consequently the functions defined above have been called. The corresponding average scores have been calculated and are listed in the table. These accuracy scores give an overview of the performance of the classifier. That is how accurately is it classifying the real and fake labelled reviews.

Average Precision	Average Recall	Average F-Score	Average Accuracy	Features
0.6557	0.6546	0.6543	0.6546	34913

4. Improving Pre-processing for better accuracy

After training the classifier, different ways have been implemented to improve the preprocessing and see the effect on the classifier performance. I have implemented the CountVectorizer with Stop words and the regex tokenizer to tokenise punctuations. Also implemented the Snowball Stemmer and Lemmatizer. Finally, also implemented the effect of using bigrams and trigrams instead of unigrams.

- **CountVectorizer for removing stop words** –_ The preprocess(text) function has been modified to remove the stop words.

```
def preProcess(text):
    vectorizer = CountVectorizer(stop_words = 'english')
    analyze = vectorizer.build_analyzer()
    tokens=analyze(text)
    return tokens
```

Regexp Tokenizer for regular expressions –_ The preprocess(text) function has been modified to split using regular expressions

-

```
def preProcess(text):
    tokenizer = RegexpTokenizer('\w+|\$[\d\.]+|\S+')
    tokens = tokenizer.tokenize(text)
    return tokens
```

- **Snowball Stemmer for stemming –** The preprocess(text) function has been modified to stem the words.

```
# To stem the data
import nltk.stem
sbs = nltk.stem.SnowballStemmer('english')

def preProcess(text):
    vectorizer = CountVectorizer(stop_words = 'english')
    analyze = vectorizer.build_analyzer()
    tokens=analyze(text)
    tokens = [sbs.stem(t) for t in tokens]
# Should return a list of tokens
    return tokens
```

- **WordNet Lemmatizer for lemmatizing –** The preprocess(text) function has been modified to lemmatize the words.

```
# # To lemmatize the data
from nltk.stem import WordNetLemmatizer
wnl = WordNetLemmatizer()

def preProcess(text):
    vectorizer = CountVectorizer(stop_words = 'english')
    analyze = vectorizer.build_analyzer()
    tokens=analyze(text)
    tokens = [wnl.lemmatize(t) for t in tokens]
# #Should return a list of tokens
    return tokens
```

- **Bigrams –** The preprocess(text) and the toFeatureVector functions have been modified to process word bigrams. Bigrams are collocations of word pairs.

```
import nltk
from nltk import bigrams
def preProcess(text):
    #normalisation and tokenising
    no_symbols = re.sub(r'[^\w]', ' ', text.lower())
    tokens = no_symbols.split()
    bitokens = list(bigrams(tokens))
    return bitokens
```

```
def toFeatureVector(words):
    featureVector = {}
    # return a dictionary 'featureVect' where the keys are the tokens in 'words' and the values
    for w in words:
        newW = '{}-{}'.format(w[0],w[1])
        #print(newW)
        #print(w)
        #try:
        featureVector[newW] = featureVector.get(newW,0) + 1
        #except KeyError:
        #    featureVector[w] = 1.0
        #try:
        featureDict[newW] = featureDict.get(newW,0) + 1
        #except KeyError:
        #    featureDict[w] = 1.0
    return featureVector
```

- **Trigrams –** The preprocess(text) and the toFeatureVector functions have been modified to process word trigrams. Trigrams are collocations of word triples as the name suggests which are similar to bigrams but contain three words.

```
import nltk
from nltk import trigrams
def preProcess(text):
    #normalisation and tokenising
    no_symbols = re.sub(r'[^\w]', ' ', text.lower())
    tokens = no_symbols.split()
    tritokens = list(trigrams(tokens))
    return tritokens
```

```
def toFeatureVector(words):
    featureVector = {}
    # return a dictionary 'featureVect' where the keys are the tokens in 'words' and the va
    for w in words:
        newW = '{}-{}-{}'.format(w[0],w[1],w[2])
        #print(newW)
        #print(w)
        #try:
        featureVector[newW] = featureVector.get(newW,0) + 1
        #except KeyError:
        #    featureVector[w] = 1.0
        #try:
        featureDict[newW] = featureDict.get(newW,0) + 1
        #except KeyError:
        #    featureDict[w] = 1.0
    return featureVector
```

Method	Average Precision	Average Recall	Average F-Score	Average Accuracy	Features
Stop-Word removal	0.6543	0.6528	0.6525	0.6528	34567
RegExp	0.6527	0.6517	0.6512	0.6517	54974
Stemmer	0.6553	0.6545	0.6542	0.6545	23790
Lemmatizer	0.6520	0.6514	0.6510	0.6514	30641
Bigrams	0.6207	0.6206	0.6205	0.6206	447746
Trigrams	0.6020	0.6015	0.6015	0.6015	1410287

As it can be seen, removing stop words, lemmatizing and stemming does not change the accuracy much. Although there is variation in the number of features. Using Bigrams and trigrams makes the accuracy worse as can be seen in the table.

5. Adding additional features to improve performance

Three additional features have been added namely Rating, Product_category and Verified_purchase. All three of these datatypes have been added individually to the default features (ID, Text) with the lable. The performance is then tested. The additional features have been added to the parseReview and have been updated in the dictionary in the loadData function itself using the update method of the dictionary. Then the three additional features have added to the standard features of ID and review Text. The following code has been implemented.

- **Verified Purchase + Rating + Product Category**

```
def loadData(path, Text=None):
    with open(path, 'rb') as f:
        reader = unicodecsv.reader(f, encoding='utf-8', delimiter='\t')
        next(reader)
        for line in reader: #Add the new features
            (Id, Text, Label,Category) = parseReview(line)
            rawData.append((Id, Text, Label, Category))
            preprocessedData.append((Id, preProcess(Text), Label, Category))

def splitData(percentage):
    dataSamples = len(rawData)
    halfOfData = int(len(rawData)/2)
    trainingSamples = int((percentage*dataSamples)/2)
    for (_, Text, Label, Category) in rawData[:trainingSamples] + rawData[halfOfData:halfOfData+trainingSamples]:
        # Update the dictionary
        d = (toFeatureVector(preProcess(Text)))
        d.update({"Product_category": Category})
        trainData.append((d,Label))
    for (_, Text, Label,Category) in rawData[trainingSamples:halfOfData] + rawData[halfOfData+trainingSamples:]:
        testData.append((toFeatureVector(preProcess(Text)),Label))
```

```
################
## QUESTION 5##
################
def parseReview(reviewLine):
    # Should return a triple of an integer, a string containing the review, and a string indi
#     L = []
    Id = (reviewLine[0])
    Text = (reviewLine[8])
    Label = (reviewLine[1])
    Rating = (reviewLine[2])
    Verified = (reviewLine[3])
    Category = (reviewLine[4])
    return (Id, Text, Label, Rating, Verified, Category)

# TEXT PREPROCESSING

def preProcess(text):
    #normalisation and tokenising
    no_symbols = re.sub(r'[^\w]', ' ', text.lower())
    tokens = no_symbols.split()
    return tokens
```

- **Rating** - Adding rating only in the loadData function and ParseReview

```
def loadData(path, Text=None):
    with open(path, 'rb') as f:
        reader = unicodecsv.reader(f, encoding='utf-8', delimiter='\t')
        next(reader)
        for line in reader: #Add the new features
            (Id, Text, Label,Rating) = parseReview(line)
            rawData.append((Id, Text, Label, Rating))
            preprocessedData.append((Id, preProcess(Text), Label, Rating))

def splitData(percentage):
    dataSamples = len(rawData)
    halfOfData = int(len(rawData)/2)
    trainingSamples = int((percentage*dataSamples)/2)
    for (_, Text, Label, Rating) in rawData[:trainingSamples] + rawData[halfOfData:halfOfData+trainingSamples]:
        # Update the dictionary
        d = (toFeatureVector(preProcess(Text)))
        d.update({"Rating": Rating})
        trainData.append((d,Label))
    for (_, Text, Label,Rating) in rawData[trainingSamples:halfOfData] + rawData[halfOfData+trainingSamples:]:
        testData.append((toFeatureVector(preProcess(Text)),Label))
```

```
################
## QUESTION 5##
################
def parseReview(reviewLine):
    # Should return a triple of an integer, a string containing the review, and a string indicating the label
#     L = []
    Id = (reviewLine[0])
    Text = (reviewLine[8])
    Label = (reviewLine[1])
    Rating = (reviewLine[2])
    Verified = (reviewLine[3])
    Category = (reviewLine[4])
    return (Id, Text, Label, Rating)
```

Verified Purchase_ - Adding verified_purchase only in the loadData function and ParseReview

```
# load data from a file and append it to the rawData
def loadData(path, Text=None):
    with open(path, 'rb') as f:
        reader = unicodecsv.reader(f, encoding='utf-8', delimiter='\t')
        next(reader)
        for line in reader: #Add the new features
            (Id, Text, Label,Verified) = parseReview(line)
            rawData.append((Id, Text, Label, Verified))
            preprocessedData.append((Id, preProcess(Text), Label, Verified))

def splitData(percentage):
    dataSamples = len(rawData)
    halfOfData = int(len(rawData)/2)
    trainingSamples = int((percentage*dataSamples)/2)
    for (_, Text, Label, Verified) in rawData[:trainingSamples] + rawData[halfOfData:halfOfData+trainingSamples]:
        # Update the dictionary
        d = (toFeatureVector(preProcess(Text)))
        d.update({"Verified_purchase": Verified})
        trainData.append((d,Label))
    for (_, Text, Label,Verified) in rawData[trainingSamples:halfOfData] + rawData[halfOfData+trainingSamples:]:
        testData.append((toFeatureVector(preProcess(Text)),Label))
```

In [11]:
```
################
## QUESTION 5##
################
def parseReview(reviewLine):
    # Should return a triple of an integer, a string containing the review, and a string indicating the label
#     L = []
    Id = (reviewLine[0])
    Text = (reviewLine[8])
    Label = (reviewLine[1])
    Rating = (reviewLine[2])
    Verified = (reviewLine[3])
    Category = (reviewLine[4])
    return (Id, Text, Label, Verified)
```

- **Product Category** - Adding product category only in the loadData function and ParseReview

```
def loadData(path, Text=None):
    with open(path, 'rb') as f:
        reader = unicodecsv.reader(f, encoding='utf-8', delimiter='\t')
        next(reader)
        for line in reader: #Add the new features
            (Id, Text, Label,Category) = parseReview(line)
            rawData.append((Id, Text, Label, Category))
            preprocessedData.append((Id, preProcess(Text), Label, Category))

def splitData(percentage):
    dataSamples = len(rawData)
    halfOfData = int(len(rawData)/2)
    trainingSamples = int((percentage*dataSamples)/2)
    for (_, Text, Label, Category) in rawData[:trainingSamples] + rawData[halfOfData:halfOfData+trainingSamples]:
        # Update the dictionary
        d = (toFeatureVector(preProcess(Text)))
        d.update({"Product_category": Category})
        trainData.append((d,Label))
    for (_, Text, Label,Category) in rawData[trainingSamples:halfOfData] + rawData[halfOfData+trainingSamples:]:
        testData.append((toFeatureVector(preProcess(Text)),Label))
```

```
###############
## QUESTION 5##
###############
def parseReview(reviewLine):
    # Should return a triple of an integer, a string containing the review, and a string indicating the label
#     L = []
    Id = (reviewLine[0])
    Text = (reviewLine[8])
    Label = (reviewLine[1])
    Rating = (reviewLine[2])
    Verified = (reviewLine[3])
    Category = (reviewLine[4])
    return (Id, Text, Label, Category)
```

Method	Average Precision	Average Recall	Average F-Score	Average Accuracy	Features
Rating	0.6587	0.6582	0.6580	0.6581	34913
Verified Purchase	0.7843	0.7826	0.7822	0.7826	34913
Product Category	0.6578	0.6574	0.6573	0.6574	34913
Verified Purchase/Rating/Product Category	0.7886	0.7873	0.7872	0.7874	34913

As can be seen in the table, Rating and Product Category do not really have an effect on the classifier's performance. Verified purchase is the only feature that improves the performance by more than 13%. Originally the performance was around 65% and now it has gone up to 78%. When all the three features are added at once, you can see that the performance slightly improves by 0.0048%. This validates that Verified purchase alone brings up the accuracy. This sheds light on the fact that if purchases are not verified, more the likelihood of the review being fake.

Part B - Data Exploration

In this section, we make an analysis of the data.

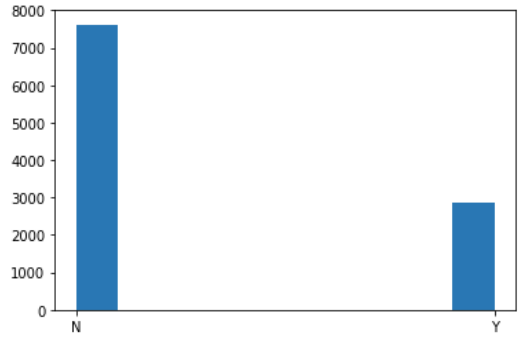
1. Analysis of the review class (fake or real)

After analysing the data, you observe that there are equal number of fake reviews and real reviews. 10,500 each and 21000 reviews in total. The graphs have been plotted using matplotlib and the code has been given in the source files.

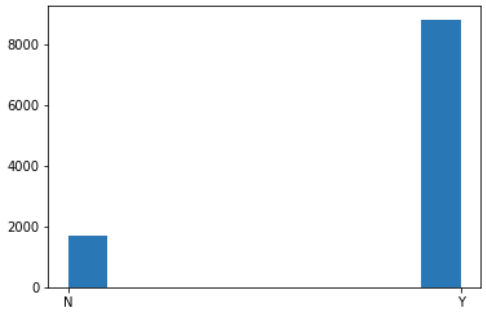
Verified Purchase

The different features of the real and fake labels have been compared. Below is the comparison of the fake label and real label with verified purchase. As can be seen, the fake labels are mostly not verified. Whereas, the real labels are mostly all verified. The graphs below depict the relation between the two. The first graph is for fake label and the second is for real.

Fake Label



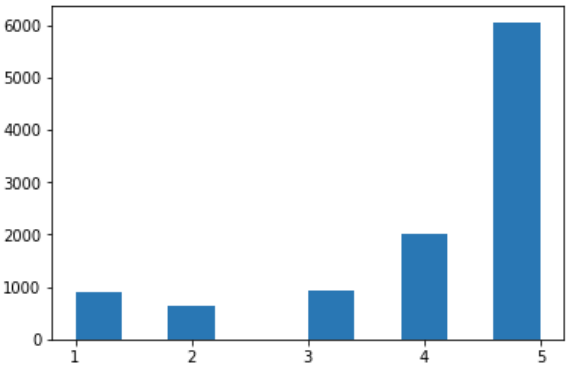
Real Label



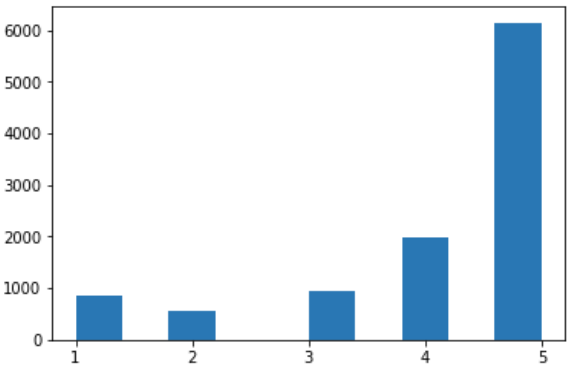
Ratings

Below, are the comparisons for ratings function. The first graph is for fake label and the second is for real. It can be seen that both fake and real classes have similar type of ratings.

Fake Label



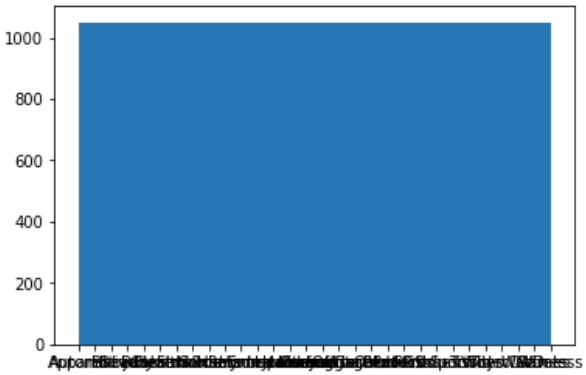
Real Label

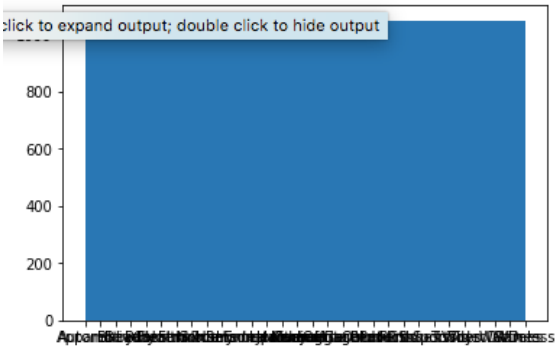


Product Category

Here, we see the different product categories the classes belong to. The first graph is for fake label and the second is for real. It is apparent from both the graphs that instances whether real or fake are equally distributed throughout for the whole data.

Fake Label **Real Label**





As can be seen from the graphs, this confirms part A results that verified purchase is a very useful feature in determining the label of the class to be real or fake.

2. Linguistic traits of the fake and real reviews

In this part, some of the linguistic and stylistic traits of the reviews are examined and the two classes are compared. The classes have been separated into fake and real classes and various aspects have been analysed. The average length of the text, the average word length, stop words, capital words, punctuation, Flesch Kinkaid reading level have been analysed. Also, counted the number of time the product title appears in the review text.

Average length of the review text for Fake Labels and Real Labels

The average length of the review text for real and fake labels has been calculated. The following code has been used to implement this after loading the data. The code is provided in the source files. The average text length of fake reviews is 61. The average text length of real reviews is 81.

[13]:
```
#how long are the reviews for each class?
###On average, how long are the reviews for each class?
def AvLength(text):
    textlength = []
    for i in text[1]:
        len(i)
        textlength.append(len(i))

    return np.mean(textlength)

print ("The average length in fake reviews is: %d " % AvLength(dfFakeReviews))

print ("The average length in real reviews is: %d" % AvLength(dfRealReviews))
```

```
The average length in fake reviews is: 61
The average length in real reviews is: 81
```

Average word Length of the reviews for Fake Labels and Real Labels

After loading the data, the following code below has been implemented to calculate the average word length. The code has been provided in the source files. The average word length in real label is 4.076 and the average word length in fake label is 4.057.

In [7]:
```
#Average word length per class
#divide fake or real
fake_words = []
real_words = []

for i in range(len(preData)):
    if (rawData[i][0] == 'fake'):
        fake_words.append(preData[i][1])
    else:
        real_words.append(preData[i][1])
```

In [8]:
```
## Counting word real and word fake
word_len_fake = []
word_len_real = []

for i in range(len(real_words)):
    for j in range(len(real_words[i])):
        word_len_real.append(len(real_words[i][j]))

for i in range(len(fake_words)):
    for j in range(len(fake_words[i])):
        word_len_fake.append(len(fake_words[i][j]))
```

In [9]:
```
## Average word length per class
print(np.average(word_len_real))
print(np.average(word_len_fake))
```

```
4.0762300770383675
4.057513688906916
```

Other Complex measures – Flesch – Kinkaid Index

Flesch - Kinkaid Readability test has also been used to address more complex measures like reading level in the real and fake labels. The Flesch - Kinkaid reading ease to calculate the average level of reading of the text. It can be seen that the there is a high level of reading ease. This implies that the text is easy to read up to 79% for both fake and real reviews.

[16]:
```
#More complex measures - Flesch Kinkaid Readability Test
str_fake = []
str_real= []

fki_fake = []
fki_real = []

for i in range(len(rawData)):
    if (rawData[i][0] == 'fake'):
        str_fake.append(rawData[i][1])
    else:
        str_real.append(rawData[i][1])

for t in range(len(str_fake)):
    fki_fake.append(textstat.flesch_reading_ease(str_fake[t]))

for t in range(len(str_real)):
    fki_real.append(textstat.flesch_reading_ease(str_real[t]))
```

[17]:
```
fake_average = sum(fki_fake)/len(fki_fake)
real_average = sum(fki_real)/len(fki_real)

print("The fake average is: ", fake_average)
print("The real average is: ", real_average)
```

```
The fake average is:  79.75902761904763
The real average is:  79.029706666667
```

Stop Words in both classes

The data has been analysed to count the number of stop words in each label. The following code has been implemented. It can be seen that fake labels contain about 51% of stop words in the review text whereas real label data has around 50% stop words as seen in the code below. This shows that there is not much difference in the number of stop words in each class.

```
stopWords = set(stopwords.words('english'))
fake_stopwords = []
real_stopwords = []
real_total =[]
fake_total = []

#Total words in real label
for i in range(len(real_words)):
    for w in real_words[i]:
        real_total.append(w)

#Total words in fake label
for i in range(len(fake_words)):
    for w in fake_words[i]:
        fake_total.append(w)

#stopwords in real label
for i in range(len(real_words)):
    for w in real_words[i]:
        if w in stopWords:
            real_stopwords.append(w)

#stopwords in fake label
for i in range(len(fake_words)):
    for w in fake_words[i]:
        if w in stopWords:
            fake_stopwords.append(w)

print("total words in fake labels: " , len(fake_total))
print("stop words in fake labels" , len(fake_stopwords))

print("total words in real labels: " ,len(real_total))
print("stop words in real labels" , len(real_stopwords))

stopword_per_fake = len(fake_stopwords)/len(fake_total)*100

stopword_per_real = len(real_stopwords)/len(real_total)*100
```

```
print("% of stopwords fake:" , stopword_per_fake)
print("% of stopwords real:" , stopword_per_real)
```

```
total words in fake labels:  641030
stop words in fake labels 330168
total words in real labels:  857365
stop words in real labels 428835
% of stopwords fake: 51.50585776016723
% of stopwords real: 50.01778705685444
```

The use punctuation in real and fake classes

Now, the use of punctuation has been analysed in both fake and real labels. The following code has been implemented. The real labels have 163501 characters of punctuation whereas the fake labels have 106917 characters of punctuation. The real reviews use more punctuation than the fake reviews. This justifies the validity of the purchase as in customers who genuinely purchased the item, would spend more time on how they phrase the reviews.

In [22]:
```
###The use of punctuation

import string

count = lambda l1, l2 : len(list(filter(lambda c: c in l2, l1)))
sum_fake_punct = 0
sum_real_punct = 0

for i in range(len(str_fake)):
    for w in str_fake[i]:
        punct_fake = count(w, string.punctuation)
        sum_fake_punct += punct_fake

for i in range(len(str_real)):
    for w in str_real[i]:
        punct_real = count(w, string.punctuation)
        sum_real_punct += punct_real
```

In [23]:
```
print("punctuation in fake label:" , sum_fake_punct)
print("punctuation in real label: ", sum_real_punct)
```

```
punctuation in fake label: 106917
punctuation in real label:  163501
```

Caps-Lock in fake and real classes

Also, the number of words starting with capital letters have been analysed in both fake and real labels. The following code has been implemented. The real labels have 53874 Caps-Lock words whereas the fake labels have 99039 Caps-Lock words. This shows that the fake labels have almost twice the number of words starting with Caps-Lock than real labels.

In [21]:
```
# no. of capital words in fake and real classes
fakeClassUpper = []
realClassUpper = []

def labelClass(text):

    for line in text:
        if line[0] == "fake":
            fakeClassUpper.append(line)
        else:
            realClassUpper.append(line)
    return
```

In [22]:
```
labelClass(rawData)
```

In [23]:
```
# Counting the no. of Capital words in the fake reviews
labelClass(rawData)
count=0
for record in fakeClassUpper:
    for word in record[1].split(" "):
        if word.isupper():
            count += 1

print("Capital Words in real class: " + str(count))
```

```
Capital Words in real class: 53874
```

In [24]:
```
# Counting the no. of capital words in real reviews

labelClass(rawData)
count=0
for record in realClassUpper:
    for word in record[1].split(" "):
        if word.isupper():
            count += 1

print("Capital Words in real class: " + str(count))
```

```
Capital Words in real class: 99039
```

Appearance of Product title in reviews

Lastly, the number of times the product title appears in the review text for both the labels has also been tested. The following code has been implemented. As can be seen, the number of times the product title appears in the review text is 1065 for fake labels where it is 924 for real labels. This shows that the number exceeds over a 100 for fake labels.

[6]:
```
# The number of times the product title appears in Reviews
tokenData=[]
for (Label,Text, prod_title) in rawData:
    tokenData.append((Label,Text, preProcess(prod_title)))
title_fake = 0
title_real= 0
for i in range(len(tokenData)):
    if (rawData[i][0] == 'fake'):
        if tokenData[i][2][0] in tokenData[i][1]:
            title_fake+=1
    else:
        if tokenData[i][2][0] in tokenData[i][1]:
            title_real+=1
```

[7]:
```
print("Fake: " + str(title_fake))
print("Real: "+ str(title_real))
```

```
Fake: 1065
Real: 924
```

3. Sentiment Analysis

In this section, the sentiment of the reviews has been looked at. To build the sentiment classifier, the deception classifier has been retained although modified. The review rating has been set as the sentiment gold standard. Reviews that got 1-2 stars are classified as negative reviews and the reviews that got 4-5 stars are positive. The rating has been used as the feature label. Previously in the Deception Classifier, the real or fake labels have been used. Whereas here, positive or negative rating has been set as the label. The rating is the 3rd column in the dataset. That is line[2] when setting it in the defined python function ParseReview. Accordingly, the load data function has been modified to append the new positive and negative lists. neg_data and pos_data have been initialised as empty lists when loading the dataset and calling the loadData function. Another aspect that was considered is the size of the positive and negative reviews. There are 16183 positive reviews whereas only 2949 negative reviews. As such, when testing the data, before splitting the data into train and test, the rawData function has been set to only take 2500 each of positive and negative reviews so that there is a balanced dataset. It can be seen that for the sentiment classifier the accuracy improves, using only the standard features that is only ID and the review text.

The following code has been implemented. The loadData function and ParseReview function have been modified. Also when splitting the data, the rawData function has been set to be a balanced dataset.

[n [5]:
```
# load data from a file and append it to the rawData
def loadData(path, Text=None):
    with open(path, 'rb') as f:
        reader = unicodecsv.reader(f, encoding='utf-8', delimiter='\t')
        next(reader)
        for line in reader:
            (Id, Text, Label) = parseReview(line)
            if Label == "negative":
                neg_data.append((Id, Text, Label))
            elif Label == "positive":
                pos_data.append((Id, Text, Label))
#             rawData is now a combination of pos_data and neg_data
#             rawData.append((Id, Text, Label))
            preprocessedData.append((Id, preProcess(Text), Label))
```

```
# Positive and negative reviews
Pos = 0
Neg = 0
three = 0
# N, P, T = 0, 0, 0


# Convert line from input file into an id/text/label tuple
def parseReview(reviewLine):
    # Use rating as a label and selecting positive and negative reviews
    Id = reviewLine[0]
    Text = reviewLine[8]
    Label = reviewLine[2]
    if int(reviewLine[2]) < 3:
        Label = "negative"
        global Neg
        Neg += 1
    elif int(reviewLine[2]) >= 4:
        Label = "positive"
        global Pos
        Pos += 1
    else:
        Label = "none"
        global three
        three += 1


    return (Id, Text, Label)

# TEXT PREPROCESSING
```

```
# selecting rawData to be of a balanced dataset
rawData = pos_data[:2500] + neg_data[:2500]
```

Additionally, after additional features have been added to the default features of the Review Text and ID. The accuracies have been listed in the table below. The code has been modified in the following way , and each additional feature has been replaced. For example first product title was added, and then replaced by verified purchase in the LoadData and ParseReview function.

```
# load data from a file and append it to the rawData
def loadData(path, Text=None):
    with open(path, 'rb') as f:
        reader = unicodecsv.reader(f, encoding='utf-8', delimiter='\t')
        next(reader)
        for line in reader:
            (Id, Text, Label, Prod_Title) = parseReview(line)
            if Label == "negative":
                neg_data.append((Id, Text, Label, Prod_Title))
            elif Label == "positive":
                pos_data.append((Id, Text, Label,Prod_Title))
#             rawData is now a combination of pos_data and neg_data›
#             rawData.append((Id, Text, Label))
            preprocessedData.append((Id, preProcess(Text), Label, Prod_Title))
```

```
def splitData(percentage):
    dataSamples = len(rawData)
    halfOfData = int(len(rawData)/2)
    trainingSamples = int((percentage*dataSamples)/2)
    for (_, Text, Label, Verified) in rawData[:trainingSamples] + rawData[halfOfData:halfOfData+trainingSamples]:
        d = (toFeatureVector(preProcess(Text)))
        d.update({"Product_title": Prod_Title})
        trainData.append((d,Label))
#         trainData.append((toFeatureVector(preProcess(Text)),Label))
    for (_, Text, Label, Verified) in rawData[trainingSamples:halfOfData] + rawData[halfOfData+trainingSamples:]:
        testData.append((toFeatureVector(preProcess(Text)),Label))
```

```
# Positive and negative reviews
Pos = 0
Neg = 0
three = 0
# N, P, T = 0, 0, 0

# DOC_ID    LABEL    RATING    VERIFIED_PURCHASE    PRODUCT_CATEGORY    PRODUCT_ID    PRODUCT_TITI
# Convert line from input file into an id/text/label tuple
def parseReview(reviewLine):
    # Use rating as a label and selecting positive and negative reviews
    Id = reviewLine[0]
    Text = reviewLine[8]
    Prod_Title = reviewLine[6]
    Review_Title = reviewLine[7]
    Verified = reviewLine[3]
    Category = reviewLine[4]
    Label = reviewLine[2]
    if int(reviewLine[2]) < 3:
        Label = "negative"
        global Neg
        Neg += 1
    elif int(reviewLine[2]) >= 4:
        Label = "positive"
        global Pos
        Pos += 1
    else:
        Label = "none"
        global three
        three += 1


    return (Id, Text, Label, Prod_Title)
```

Method/Features	Average Precision	Average Recall	Average F-Score	Average Accuracy	No. of Features
Text, ID (default)	0.7906	0.7890	0.7890	0.7890	14888
Text, ID, Product Title	0.7683	0.7665	0.7665	0.7665	14888
Text, ID, Verified Purchase	0.7911	0.79025	0.7902	0.79025	14888
Text, ID, Review Title	0.7851	0.783	0.7823	0.783	14888
Text, ID, Product_Category	0.7789	0.7780	0.7780	0.778	14888

As can be seen in the table, the sentiment classifier performance is higher than the deception classifier when using the default features alone. The deception classifier's average accuracy with just ID and Text was approximately 65% whereas here it is 79% approximately. You can also notice that adding additional features does not improve the performance much. One can interpolate from this that there is difference in the way in the functioning when choosing ratings as the label instead of real or fake.

References

- http://www.nltk.org/book/
- http://scikit-learn.org/stable/
- https://stackoverflow.com