# 664 NLP Assignment 3¶

# Wanyue Xiao

# Due: April 12, 2020

The purpose of this assignment is to explore the sentiment of the comments at the sentence level. Specifically, this includes how to process the words and how to conduct the sentiment analysis using classifiers. Ultimately, two types of sentences, which are positive sentences and negative sentences, will be extracted and saved as files.

```python
import nltk
import sklearn
from nltk import *
from nltk.corpus import treebank
from nltk.corpus import sentence_polarity
import random
import re
import math
from IPython import display
from pprint import pprint
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score, cross_validate, KFold
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
sns.set(style='darkgrid', context='talk', palette='Dark2')
```

```python
f = open('review_Content.txt')
review_text = f.read()
```

By using the len function, we find that the review_text contains 88129772 words

```python
len(review_text)
```

```
88129772
```

## Sentence Parsing

Now, since we want to extract sentences from the review text, we need to parse those texts to sentences by using RegExr. Before that, we should replace '\n' with blank space and then use RegExr to parse sentence.

```python
review_text = review_text.replace('\n',' ')
sentence=re.compile("[A-Z].*?[\.\!\?]+", re.MULTILINE | re.DOTALL )
parsed_sentence = sentence.findall(review_text)
```

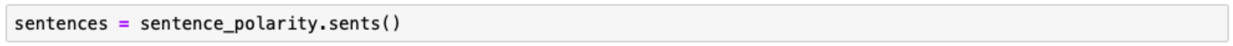
For the target documents, it contains 1202086 sentences.

```python
len(parsed_sentence)
```

```
1202086
```

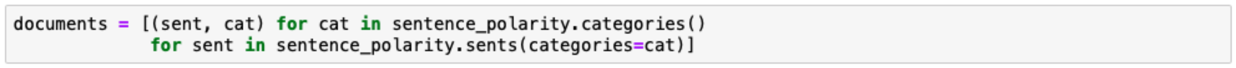
## Feature Selection

We can use the sentence_polarity package's dataset to build models (or classifier). First of all, we need to get the sentences dataset.



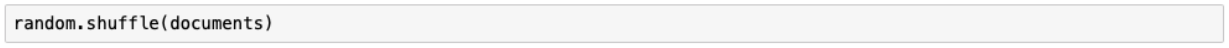
```
sentences = sentence_polarity.sents()
```

Then, given that we have to use this dataset to train classifier, both the data of 'sentence' and 'tag' should be extracted from the dataset.



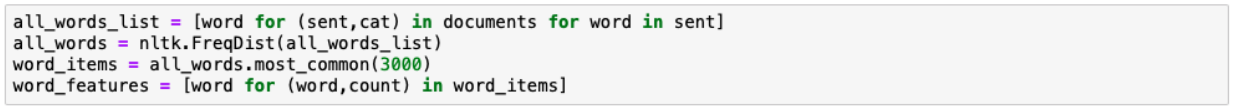
```
documents = [(sent, cat) for cat in sentence_polarity.categories()
             for sent in sentence_polarity.sents(categories=cat)]
```

Since the documents are in order by label, we mix them up for later separation into training and test sets.



```
random.shuffle(documents)
```

Here we put all the tokens inside the all_words_list. Then we can use the FreqDist function to calculate the word count of each token that is inside the all_words_list. After several experiments of feature words selection, we decide to use the top 3000 words (which is the optimal number) and store it inside the word_features object.



```
all_words_list = [word for (sent,cat) in documents for word in sent]
all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(3000)
word_features = [word for (word,count) in word_items]
```

Here we can build a data frame to go through those top features.
The top 20 tokens do not provide much meaningful information since most of them are punctuations and stop words. We can find that '.' has appeared 14010 times because most of the sentences end up with stop sign. Besides, most of them are determinator or prepositions. Hence, we need to eliminate those words or punctuations.

```
word_features_df = pd.DataFrame.from_records(word_items)
word_features_df.columns = ['Feature', 'Count']
word_features_df.head(20)
```

|  | Feature | Count |
|---|---|---|
| 0 | . | 14010 |
| 1 | the | 10098 |
| 2 | , | 10037 |
| 3 | a | 7282 |
| 4 | and | 6196 |
| 5 | of | 6062 |
| 6 | to | 4233 |
| 7 | is | 3368 |
| 8 | in | 2629 |
| 9 | that | 2470 |
| 10 | it | 2283 |
| 11 | as | 1801 |
| 12 | but | 1637 |
| 13 | with | 1561 |
| 14 | film | 1446 |
| 15 | this | 1440 |
| 16 | for | 1436 |
| 17 | its | 1336 |
| 18 | an | 1323 |
| 19 | movie | 1268 |

Introducing the stop word from nltk package.

```
stopwords = nltk.corpus.stopwords.words('english')
```

After removing stop words and punctuations from the all_words_list object, we can then, again, select the top 3000 words and store it in the word_features_noStop object.

```
all_words_list_noStop = [word for word in all_words_list if word.isalpha()]
all_words_list_noStop = [w for w in all_words_list_noStop if w not in stopwords]
all_words_noStop = nltk.FreqDist(all_words_list_noStop)
word_items_noStop = all_words_noStop.most_common(3000)
word_features_noStop = [word for (word,count) in word_items_noStop]
```

Then, we can go through those top features. Now, we can summarise that the token with the highest frequency is film (which is 1446). The second common token is movie, which is 1268. Besides, words like 'story', 'commedy', and 'characters' are also upon the list. Hence, we can speculate that the dataset records reviews of movie or film.

```
word_items_noStop_df = pd.DataFrame.from_records(word_items_noStop)
word_items_noStop_df.columns = ['Feature', 'Count']
word_items_noStop_df.head(20)
```

| | Feature | Count |
|---|---|---|
| 0 | film | 1446 |
| 1 | movie | 1268 |
| 2 | one | 727 |
| 3 | like | 720 |
| 4 | story | 476 |
| 5 | much | 386 |
| 6 | even | 382 |
| 7 | good | 377 |
| 8 | comedy | 353 |
| 9 | time | 339 |
| 10 | characters | 313 |
| 11 | little | 302 |
| 12 | way | 296 |
| 13 | funny | 283 |
| 14 | make | 278 |
| 15 | enough | 267 |
| 16 | never | 262 |
| 17 | makes | 252 |
| 18 | may | 245 |
| 19 | us | 241 |

## Data Cleaning

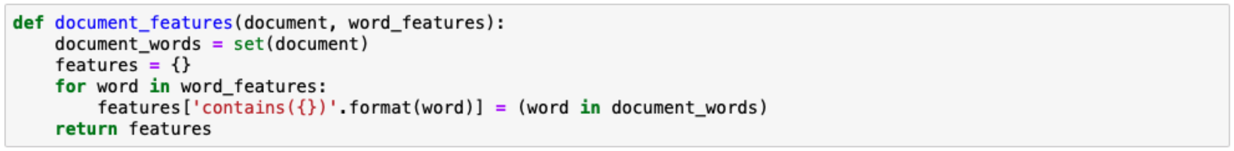
Considering all the features we selected are lower-case, we need to convert the review_text to lower_case for the good of classification's performance.

```
parsed_sentence_lower = [each.lower() for each in parsed_sentence]
```

## Classification Model

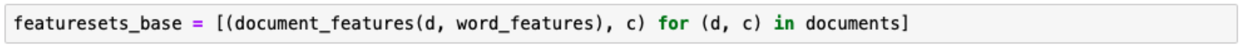
## Method 1: Baseline Model

The first one is a baseline model which only uses the word_features (top 3000 corpus with stopwords and punctuations) we obtained above. Then we can create a document_features function to "dummy" the words contained inside the documents object.



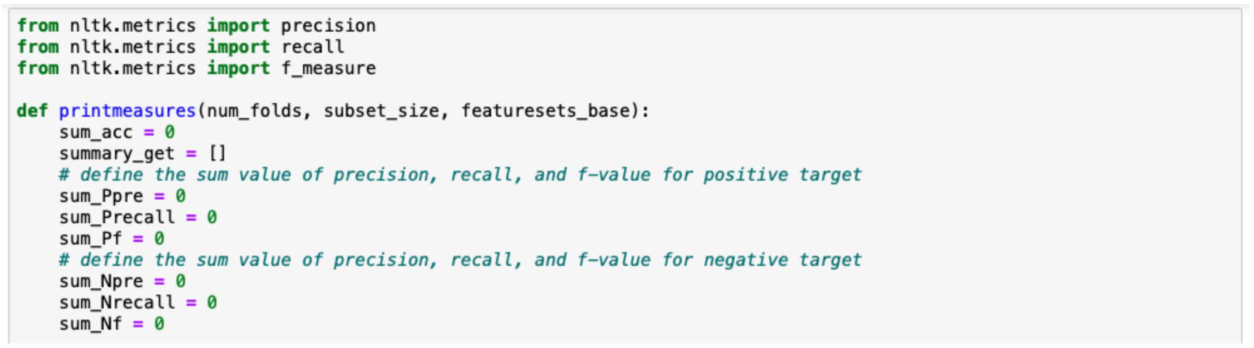
```python
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features
```

The feature below uses the document_features function created above to process the documents with word_features. Now we can get featureset which contains processed features and target tag.



```python
featuresets_base = [(document_features(d, word_features), c) for (d, c) in documents]
```

For model performance evaluation, it is a commonplace to use cross-validation to assess models. The naive Bayes model from nltk, however, does not support the cross-validation function or compatible with cross-validation function from sklearn package. Then, we build a cross-validation function by using nltk.metrics function. This function will return a list which records average value of 'Accuracy', 'Pos Precision', 'Pos Recall', 'Pos F-Measure', 'Neg Precision', 'Neg Recall', and 'Neg F-Measure'.



```python
from nltk.metrics import precision
from nltk.metrics import recall
from nltk.metrics import f_measure

def printmeasures(num_folds, subset_size, featuresets_base):
    sum_acc = 0
    summary_get = []
    # define the sum value of precision, recall, and f-value for positive target
    sum_Ppre = 0
    sum_Precall = 0
    sum_Pf = 0
    # define the sum value of precision, recall, and f-value for negative target
    sum_Npre = 0
    sum_Nrecall = 0
    sum_Nf = 0
    for i in range(num_folds):
        testing_set = featuresets_base[i*subset_size:(i+1)*subset_size]
        training_set = featuresets_base[:i*subset_size] + featuresets_base[(i+1)*subset_size:]
        classifier = nltk.NaiveBayesClassifier.train(training_set)
        acc = nltk.classify.accuracy(classifier,testing_set)
        print('Iteration',i,'\'s Accuracy:', acc)
        sum_acc = sum_acc + acc

        reflist = []
        testlist = []
        for (features, label) in testing_set:
            reflist.append(label)
            testlist.append(classifier.classify(features))

        refpos = set()
        refneg = set()
        testpos = set()
        testneg = set()

        for i, label in enumerate(reflist):
            if label == 'pos': refpos.add(i)
            if label == 'neg': refneg.add(i)
        for i, label in enumerate(testlist):
            if label == 'pos': testpos.add(i)
            if label == 'neg': testneg.add(i)

        # calculate the precision value
        pre = precision(refpos, testpos)
        if pre is None:
            pre = 0
        sum_Ppre = sum_Ppre + pre
        # calculate the recall value
        rec = recall(refpos, testpos)
        if rec is None:
            rec = 0
        sum_Precall = sum_Precall + rec
        # calculate the f-measure value
        f_mea = f_measure(refpos, testpos)
        if f_mea is None:
            f_mea = 0
        sum_Pf = sum_Pf + f_mea

        # calculate the precision value
        Npre = precision(refneg, testneg)
        if Npre is None:
            Npre = 0
        sum_Npre = sum_Npre + Npre
        # calculate the recall value
        Nrecall = recall(refneg, testneg)
        if Nrecall is None:
            Nrecall = 0
        sum_Nrecall = sum_Nrecall + Nrecall
        # calculate the f-measure value
        Nf = f_measure(refneg, testneg)
        if Nf is None:
            Nf = 0
        sum_Nf = sum_Nf + Nf

    print("Performance Summary:")
    summary_get.append([round(sum_acc/num_folds,4)])
    summary_get.append([round(sum_Ppre/num_folds,4)])
    summary_get.append([round(sum_Precall/num_folds,4)])
    summary_get.append([round(sum_Pf/num_folds,4)])
    summary_get.append([round(sum_Npre/num_folds,4)])
    summary_get.append([round(sum_Nrecall/num_folds,4)])
    summary_get.append([round(sum_Nf/num_folds,4)])
    print("Accuracy:", sum_acc/num_folds)
    print("Pos_Precision:", sum_Ppre/num_folds)
    print("Pos_Recall:", sum_Precall/num_folds)
    print("Pos_F-Value:", sum_Pf/num_folds)
    print("Neg_Precision:", sum_Npre/num_folds)
    print("Neg_Recall:", sum_Nrecall/num_folds)
    print("Neg_F-Value:",  sum_Nf/num_folds)
    summary_get
    return(classifier)
```

Calling the printmeasures() function to display the result. We can see that the average accuracy rate is 0.7624 which seems mediocre. The recall in positive target is 0.7507. Similarly, the recall value in negative target group is 0.7740 which seems fine. This indicates that the ability to predict negative sentence is better than that to predict positive sentences.

```
printmeasures(5, 2000, featuresets_base)
```

```
Iteration 0 's Accuracy: 0.765
Iteration 1 's Accuracy: 0.769
Iteration 2 's Accuracy: 0.7625
Iteration 3 's Accuracy: 0.7575
Iteration 4 's Accuracy: 0.758
Performance Summary:
Accuracy: 0.7624
Pos_Precision: 0.7677326832934973
Pos_Recall: 0.7506854930267624
Pos_F-Value: 0.759034348440264
Neg_Precision: 0.757375900465416
Neg_Recall: 0.7739486223187952
Neg_F-Value: 0.7654989007232766
```

### Stopwords and Punctuations Remove

Now, by using the baseline model, we decide to remove stowords and to see what will happen.

```
featuresets_noStop = [(document_features(d, word_features_noStop), c) for (d, c) in documents]
```

Compared with the average accuracy of baseline model with stopwords and punctuations, that of baseline model without stopwords and punctuations (which is 0.7549) decreases by approximately 0.1%. In this case, the recall value in negative group is still higher than recall value in positive group. The recall values in negative group decrease by nearly 0.3%. The recall in positive group reduces approximately by 1%. One can speculate that the removing stopwords and punctuations might negatively influence the model's ability to predict sentences' category.

```
printmeasures(5, 2000, featuresets_noStop)
```

```
Iteration 0 's Accuracy: 0.76
Iteration 1 's Accuracy: 0.756
Iteration 2 's Accuracy: 0.7565
Iteration 3 's Accuracy: 0.7435
Iteration 4 's Accuracy: 0.7585
Performance Summary:
Accuracy: 0.7548999999999999
Pos_Precision: 0.7618974031862784
Pos_Recall: 0.7391270412255008
Pos_F-Value: 0.750332685149241
Neg_Precision: 0.748192380291801
Neg_Recall: 0.7704064193274383
Neg_F-Value: 0.7591312252425461
```

## Method 2: Subjectivity Model with Stopwords

Now, let's try another model which introduces the subjectivity words as features.

```
SLpath = 'subjclueslen1-HLTEMNLP05.tff'
```

We have to create a function to extract contents from the SLpath document.

```
def readSubjectivity(path):
    flexicon = open(path, 'r')
    # initialize an empty dictionary
    sldict = { }
    for line in flexicon:
        fields = line.split()   # default is to split on whitespace
        # split each field on the '=' and keep the second part as the value
        strength = fields[0].split("=")[1]
        word = fields[2].split("=")[1]
        posTag = fields[3].split("=")[1]
        stemmed = fields[4].split("=")[1]
        polarity = fields[5].split("=")[1]
        if (stemmed == 'y'):
            isStemmed = True
        else:
            isStemmed = False
        # put a dictionary entry with the word as the keyword
        #     and a list of the other values
        sldict[word] = [strength, posTag, isStemmed, polarity]
    return sldict
```

```
SL = readSubjectivity(SLpath)
```

Here we can see that the number of subjectivity words inside the document is 6885.

```
len(SL.keys())
```

```
6885
```

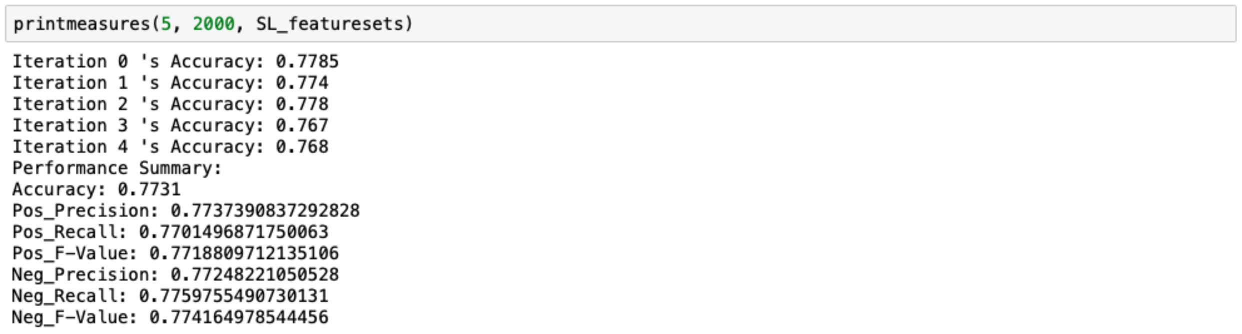
After gathering the subjectivity words, we should create a feature generation function. The SL_features function will 'dummy' each word inside the sentence. Then, it will compare each word to words inside the subjectivity document. Once the word matches the document, the corresponding variable will increase by 1. Finally, we can get two count variables which records the count value of positive words and negative words.

```
def SL_features(document, word_features, SL):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    # count variables for the 4 classes of subjectivity
    weakPos = 0
    strongPos = 0
    weakNeg = 0
    strongNeg = 0
    for word in document_words:
        if word in SL:
            strength, posTag, isStemmed, polarity = SL[word]
            if strength == 'weaksubj' and polarity == 'positive':
                weakPos += 1
            if strength == 'strongsubj' and polarity == 'positive':
                strongPos += 1
            if strength == 'weaksubj' and polarity == 'negative':
                weakNeg += 1
            if strength == 'strongsubj' and polarity == 'negative':
                strongNeg += 1
            features['positivecount'] = weakPos + (2 * strongPos)
            features['negativecount'] = weakNeg + (2 * strongNeg)
    return features
```

Using the SL_features function to fit the document.

```
SL_featuresets = [(SL_features(d, word_features, SL), c) for (d, c) in documents]
```

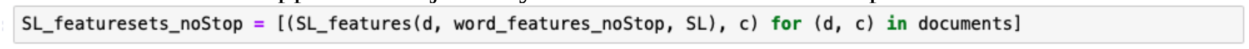
Again, calling the printmeasure function to conduct the cross-validation method. The average accuracy is 0.7731, which is higher than that of baseline model with stop words and punctuations. The average recall of SL_features model in the positive and negative target groups is 0.770 and 0.775 respectively, which are also higher than those of baseline model in the positive and negative group. Therefore, it seems that this model is better than baseline model.



```
printmeasures(5, 2000, SL_featuresets)

Iteration 0 's Accuracy: 0.7785
Iteration 1 's Accuracy: 0.774
Iteration 2 's Accuracy: 0.778
Iteration 3 's Accuracy: 0.767
Iteration 4 's Accuracy: 0.768
Performance Summary:
Accuracy: 0.7731
Pos_Precision: 0.7737390837292828
Pos_Recall: 0.7701496871750063
Pos_F-Value: 0.7718809712135106
Neg_Precision: 0.77248221050528
Neg_Recall: 0.7759755490730131
Neg_F-Value: 0.774164978544456
```

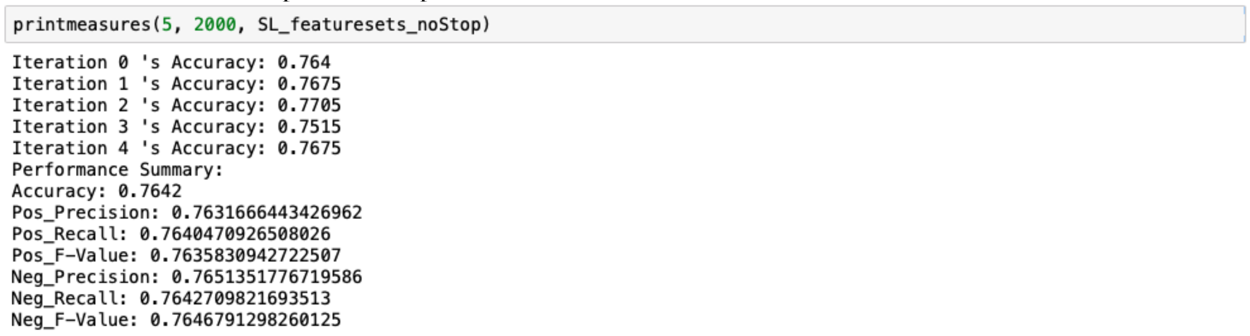
**Stopwords and Punctuations Remove**

Let's see what will happen to Subjectivity Model that is without Stopwords.



```
SL_featuresets_noStop = [(SL_features(d, word_features_noStop, SL), c) for (d, c) in documents]
```

Compared with the average accuracy of SL model with stopwords and punctuations, that of SL model without stopwords and punctuations decrease by nearly 0.9% which is a minor change. The recall values in positive and negative group are the same which is 0.764. The change between recalls in this model, for both groups, and recall of SL with stopwords is nearly 1%. This indicates that, for SL model, removing stopswords and punctuations might negatively influence the model's prediction performance.

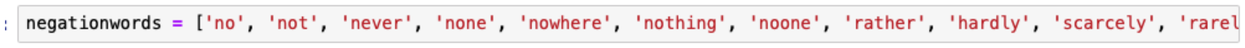


```
printmeasures(5, 2000, SL_featuresets_noStop)

Iteration 0 's Accuracy: 0.764
Iteration 1 's Accuracy: 0.7675
Iteration 2 's Accuracy: 0.7705
Iteration 3 's Accuracy: 0.7515
Iteration 4 's Accuracy: 0.7675
Performance Summary:
Accuracy: 0.7642
Pos_Precision: 0.7631666443426962
Pos_Recall: 0.7640470926508026
Pos_F-Value: 0.7635830942722507
Neg_Precision: 0.7651351776719586
Neg_Recall: 0.7642709821693513
Neg_F-Value: 0.7646791298260125
```

## Method 3: Representing Negation

Examples of negation words include not, no, never, cannot, shouldn't, wouldn't, etc. Negation handling is an automatic way of determining the scope of negation and inverting the polarities of opinionated words that are actually affected by a negation.

The negationwords are 'no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone', 'rather', 'hardly', 'scarcely', 'rarely', 'seldom', 'neither', 'nor'.



```
negationwords = ['no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone', 'rather', 'hardly', 'scarcely', 'rarel
```

Here we can create a NOT_features function. Start the feature set with all 3000-word features and 6000 Not word features set to false. If a negation occurs, add the following word as a Not word feature (if it’s in the top 6000 feature words), and otherwise add it as a regular feature word.

```
def NOT_features(document, word_features, negationwords):
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = False
        features['contains(NOT{})'.format(word)] = False
    # go through document words in order
    for i in range(0, len(document)):
        word = document[i]
        if ((i + 1) < len(document)) and ((word in negationwords) or (word.endswith("n't"))):
            i += 1
            features['contains(NOT{})'.format(document[i])] = (document[i] in word_features)
        else:
            features['contains({})'.format(word)] = (word in word_features)
    return features
```

Create feature sets as before, using the NOT_features extraction funtion, train the classifier and test the accuracy.

```
NOT_featuresets = [(NOT_features(d, word_features, negationwords), c) for (d, c) in documents]
```

Calling the printmeasures() function to summarise the model performance. The average accuray is 0.7779, which is highest among those three models. The average recalls of negative target groups are also the highest up to now. Therefore, Negation Model is being recognized as the best in this report.

```
printmeasures(5, 2000, NOT_featuresets)
```

```
Iteration 0 's Accuracy: 0.7775
Iteration 1 's Accuracy: 0.782
Iteration 2 's Accuracy: 0.773
Iteration 3 's Accuracy: 0.777
Iteration 4 's Accuracy: 0.78
Performance Summary:
Accuracy: 0.7779
Pos_Precision: 0.779500651678768
Pos_Recall: 0.774906504864586
Pos_F-Value: 0.7771677384441918
Neg_Precision: 0.7763384415658502
Neg_Recall: 0.7807945789240769
Neg_F-Value: 0.7785325204112141
```

After deciding to select the negation model with stopwords and punctuations as the best model to predict the reviews, we have to get a classifier based on this model.

## Stopwords and Punctuations Remove

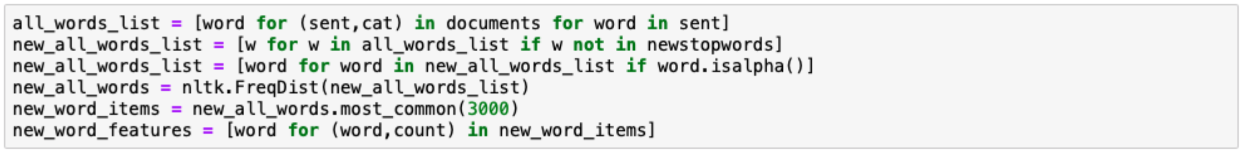
Representing Negation without Stopwords and punctuations.
Using a stopword list to prune the word features. We’ll start with the NLTK stop word list, but we’ll remove some of the negation words, or parts of words, that our negation filter uses. There are several words that are contained in both the negation and stopwords object. Hence, we need to find out those words and eliminate them from the stopwords object.

```
stopwords = nltk.corpus.stopwords.words('english')
newstopwords = [word for word in stopwords if word not in negationwords]
```

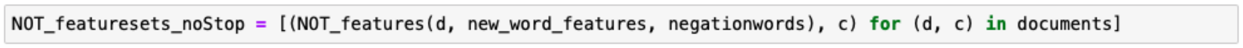
```
len(newstopwords)
```

```
176
```

Then, executing the same procedures to find out the top 3000 features.

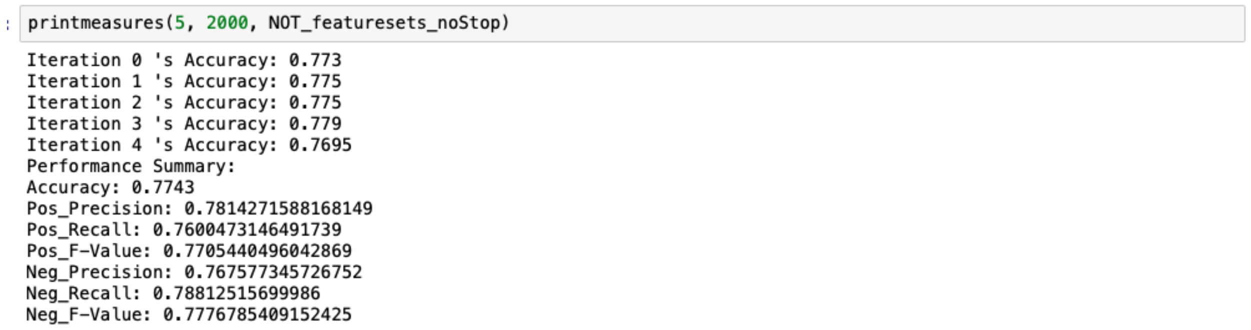


```
all_words_list = [word for (sent,cat) in documents for word in sent]
new_all_words_list = [w for w in all_words_list if w not in newstopwords]
new_all_words_list = [word for word in new_all_words_list if word.isalpha()]
new_all_words = nltk.FreqDist(new_all_words_list)
new_word_items = new_all_words.most_common(3000)
new_word_features = [word for (word,count) in new_word_items]
```



```
NOT_featuresets_noStop = [(NOT_features(d, new_word_features, negationwords), c) for (d, c) in documents]
```

Compared with the average accuracy of negation model with stopwords and punctuations, that of negation model without stopwords and punctuations decrease by nearly 0.03%. The recall values in positive group is 0.76 which is also lower than that of negation model with stopwords. The recalls negative group groups, however, is higher than that of negation model showed above. One can make a conclusion that removing punctuations and stopwords might not be a good idea for this dataset even though the change in accuracy between those two negation models is small.



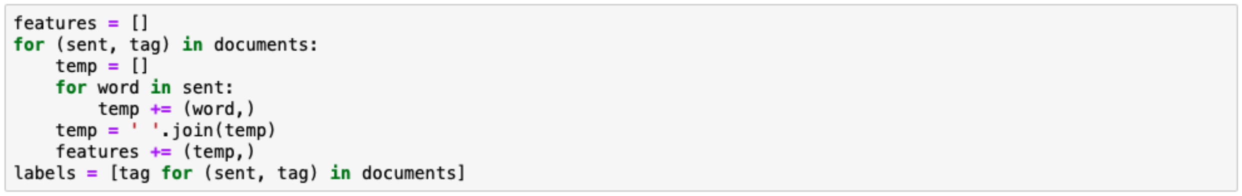
```
printmeasures(5, 2000, NOT_featuresets_noStop)
```

```
Iteration 0 's Accuracy: 0.773
Iteration 1 's Accuracy: 0.775
Iteration 2 's Accuracy: 0.775
Iteration 3 's Accuracy: 0.779
Iteration 4 's Accuracy: 0.7695
Performance Summary:
Accuracy: 0.7743
Pos_Precision: 0.7814271588168149
Pos_Recall: 0.7600473146491739
Pos_F-Value: 0.7705440496042869
Neg_Precision: 0.767577345726752
Neg_Recall: 0.78812515699986
Neg_F-Value: 0.7776785409152425
```

## Additional Experiment Part

Even though we did some experiments which covers topics we learned in the class, we still wish to find that if other classifiers have a better performance on this dataset nor not. In this case, we will use the Multinomial Naïve Bayes Model and Random Forest to train the dataset.

Firstly, we can get two variables by splitting the documents object into two.
The sents contain the sentences (which are from the sentence_polarity package) while the targets object contains labels (labels that decides the category of sentences).



```
features = []
for (sent, tag) in documents:
    temp = []
    for word in sent:
        temp += (word,)
    temp = ' '.join(temp)
    features += (temp,)
labels = [tag for (sent, tag) in documents]
```

We can generate document term matrix by using scikit-learn's CountVectorizer.
Inside the CountVectorizer function, we can set several parameters. The lowercase parameter is used to turn the text to lower case. Here we assign the values of stopwords to the stop_words parameter. Besides, we can create a tokenizer to remove unwanted elements from out data like symbols and numbers.

Here, given that the final model we selected above is Negation Model and the features set we selected above is the word_features (the top 3000 features which contains stop words and

punctuations). Hence, in order to get the same feature set, we only use max_features this parameter to select the top 3000 features.
Then we can use the fit_transform() function to label encode the target variable. This is done to transform Categorical data of string type in the data set into numerical values which the model can understand.

```
from sklearn.feature_extraction.text import CountVectorizer
from nltk.tokenize import RegexpTokenizer
cv = CountVectorizer(max_features=3000)
```

```
text_counts = cv.fit_transform(features)
```

**Split train and test set**
The Corpus will be split into two data sets, Training and Test. The training data set will be used to fit the model and the predictions will be performed on the test data set. The Training Data will have 70% of the corpus and Test data will have the remaining 30% as we have set the parameter *test_size=0.3*.
Let's split dataset by using function train_test_split(). This function needs 3 parameters, which are features, target, and test_set size. Additionally, we can use random_state to select records randomly.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(text_counts, labels, test_size=0.3, random_state=1)
```

Import scikit-learn metrics module for accuracy calculation and generate a model by using Multinomial Naive Bayes. The accuracy of Multinomial Naive Bayes is 0.760 which is higher than that of Baseline Model and SL Model but is lower than that of Negation Model. The accuracy rate, however, is quite high compared with other instances above.

```
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics

clf = MultinomialNB().fit(X_train, y_train)
predicted= clf.predict(X_test)
print("MultinomialNB Accuracy:",'\n',metrics.accuracy_score(y_test, predicted))
print("MultinomialNB Matrix:",'\n',confusion_matrix(y_test,predicted))
print("MultinomialNB Report:",'\n',classification_report(y_test,predicted))
```

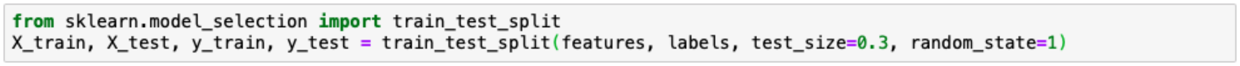
```
MultinomialNB Accuracy:
 0.7608627696155048
MultinomialNB Matrix:
 [[1242  342]
 [ 423 1192]]
MultinomialNB Report:
               precision    recall  f1-score   support

         neg       0.75      0.78      0.76      1584
         pos       0.78      0.74      0.76      1615

    accuracy                           0.76      3199
   macro avg       0.76      0.76      0.76      3199
weighted avg       0.76      0.76      0.76      3199
```

Then, let's build the **Text Classification Model by using TF-IDF instead of Bag-of-word method**.

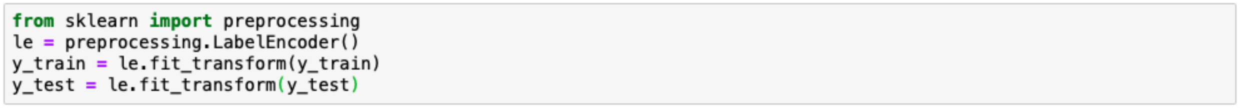
- In Term Frequency (TF), you just count the number of words occurred in each document. The main issue with this Term Frequency is that it will give more weight to longer documents. Term frequency is basically the output of the BoW model.

- IDF (Inverse Document Frequency) measures the amount of information a given word provides across the document. IDF is the logarithmically scaled inverse ratio of the number of documents that contain the word and the total number of documents.
- TF-IDF (Term Frequency-Inverse Document Frequency) normalizes the document term matrix. It is the product of TF and IDF. Word with high tf-idf in a document, it is most of the times occurred in given documents and must be absent in the other documents. So the words must be a signature word.

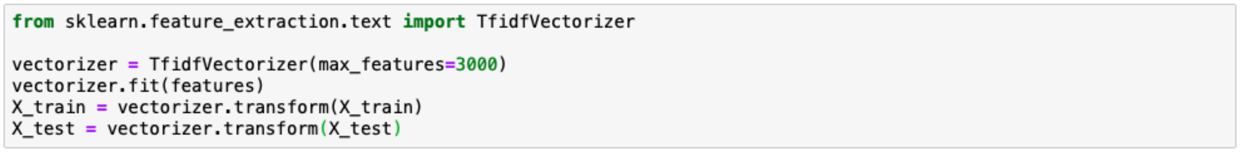
Similarly, we can use the TfidVectorizer to highlight words that are more interesting, e.g. frequent in a document but not across documents. Also, setting the max_features as 3000.



```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.3, random_state=1)
```

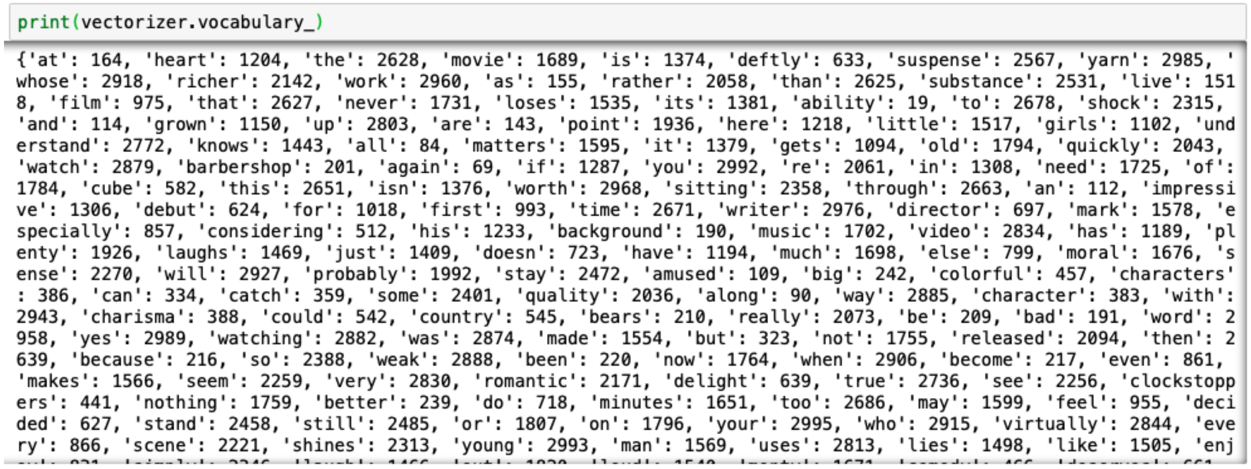


```
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
y_train = le.fit_transform(y_train)
y_test = le.fit_transform(y_test)
```



```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(max_features=3000)
vectorizer.fit(features)
X_train = vectorizer.transform(X_train)
X_test = vectorizer.transform(X_test)
```

We can use vocabulary_ to go through the features that selected by using TF-IDF method.



```
print(vectorizer.vocabulary_)
```

```
{'at': 164, 'heart': 1204, 'the': 2628, 'movie': 1689, 'is': 1374, 'deftly': 633, 'suspense': 2567, 'yarn': 2985, '
whose': 2918, 'richer': 2142, 'work': 2960, 'as': 155, 'rather': 2058, 'than': 2625, 'substance': 2531, 'live': 151
8, 'film': 975, 'that': 2627, 'never': 1731, 'loses': 1535, 'its': 1381, 'ability': 19, 'to': 2678, 'shock': 2315,
'and': 114, 'grown': 1150, 'up': 2803, 'are': 143, 'point': 1936, 'here': 1218, 'little': 1517, 'girls': 1102, 'und
erstand': 2772, 'knows': 1443, 'all': 84, 'matters': 1595, 'it': 1379, 'gets': 1094, 'old': 1794, 'quickly': 2043,
'watch': 2879, 'barbershop': 201, 'again': 69, 'if': 1287, 'you': 2992, 're': 2061, 'in': 1308, 'need': 1725, 'of':
1784, 'cube': 582, 'this': 2651, 'isn': 1376, 'worth': 2968, 'sitting': 2358, 'through': 2663, 'an': 112, 'impressi
ve': 1306, 'debut': 624, 'for': 1018, 'first': 993, 'time': 2671, 'writer': 2976, 'director': 697, 'mark': 1578, 'e
specially': 857, 'considering': 512, 'his': 1233, 'background': 190, 'music': 1702, 'video': 2834, 'has': 1189, 'pl
enty': 1926, 'laughs': 1469, 'just': 1409, 'doesn': 723, 'have': 1194, 'much': 1698, 'else': 799, 'moral': 1676, 's
ense': 2270, 'will': 2927, 'probably': 1992, 'stay': 2472, 'amused': 109, 'big': 242, 'colorful': 457, 'characters'
: 386, 'can': 334, 'catch': 359, 'some': 2401, 'quality': 2036, 'along': 90, 'way': 2885, 'character': 383, 'with':
2943, 'charisma': 388, 'could': 542, 'country': 545, 'bears': 210, 'really': 2073, 'be': 209, 'bad': 191, 'word': 2
958, 'yes': 2989, 'watching': 2882, 'was': 2874, 'made': 1554, 'but': 323, 'not': 1755, 'released': 2094, 'then': 2
639, 'because': 216, 'so': 2388, 'weak': 2888, 'been': 220, 'now': 1764, 'when': 2906, 'become': 217, 'even': 861,
'makes': 1566, 'seem': 2259, 'very': 2830, 'romantic': 2171, 'delight': 639, 'true': 2736, 'see': 2256, 'clockstopp
ers': 441, 'nothing': 1759, 'better': 239, 'do': 718, 'minutes': 1651, 'too': 2686, 'may': 1599, 'feel': 955, 'deci
ded': 627, 'stand': 2458, 'still': 2485, 'or': 1807, 'on': 1796, 'your': 2995, 'who': 2915, 'virtually': 2844, 'eve
ry': 866, 'scene': 2221, 'shines': 2313, 'young': 2993, 'man': 1569, 'uses': 2813, 'lies': 1498, 'like': 1505, 'enj
[illegible]
```

The classification rate of MultinomialNB using TF-IDF features is 0.7684, which is higher than the model with BoW method. Hence, we can speculate that TF-IDF is better than BoW method.

```
clf = MultinomialNB().fit(X_train, y_train)
predicted= clf.predict(X_test)
print("MultinomialNB Accuracy:",'\n',metrics.accuracy_score(y_test, predicted))
print("MultinomialNB Matrix:",'\n',confusion_matrix(y_test,predicted))
print("MultinomialNB Report:",'\n',classification_report(y_test,predicted))
```

```
MultinomialNB Accuracy:
 0.7683651140981557
MultinomialNB Matrix:
 [[1233  327]
 [ 414 1225]]
MultinomialNB Report:
               precision    recall  f1-score   support

           0       0.75      0.79      0.77      1560
           1       0.79      0.75      0.77      1639

    accuracy                           0.77      3199
   macro avg       0.77      0.77      0.77      3199
weighted avg       0.77      0.77      0.77      3199
```

We need to improve the accuracy by using some other preprocessing or feature engineering. Let's suggest in comment box some approach for accuracy improvement.

## Random Forest with the Top 3000

In the code below, we define that the max_features should be 3000, which means that it only uses the 2500 most frequently occurring words to create a bag of words feature vector. Words that occur less frequently are not very useful for classification.
Similarly, max_df specifies that only use those words that occur in a maximum of 80% of the documents. Words that occur in all documents are too common and are not very useful for classification. Similarly, min-df is set to 7 which shows that include words that occur in at least 7 documents.

```
from sklearn.ensemble import RandomForestClassifier
text_classifier = RandomForestClassifier(n_estimators=400, random_state=1)
text_classifier.fit(X_train, y_train)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=400,
                       n_jobs=None, oob_score=False, random_state=1, verbose=0,
                       warm_start=False)
```

```
predictions = text_classifier.predict(X_test)
```

The accuracy rate is 0.6990 which is the lowest up to now. Then we have to change the model types.

```
print("Random Forest Accuracy:",'\n',accuracy_score(y_test, predictions))
print("Random Forest Matrix:",'\n',confusion_matrix(y_test,predictions))
print("Random Forest Report:",'\n',classification_report(y_test,predictions))
```

```
Random Forest Accuracy:
 0.6989684276336355
Random Forest Matrix:
 [[1115  445]
 [ 518 1121]]
Random Forest Report:
               precision    recall  f1-score   support

         neg       0.68      0.71      0.70      1560
         pos       0.72      0.68      0.70      1639

    accuracy                           0.70      3199
   macro avg       0.70      0.70      0.70      3199
weighted avg       0.70      0.70      0.70      3199
```

## Support Vector Machine (word features with stopwords)

We can also use linear SVM. Linear SVM is better compare to other non-linear SVM models because text classification has lots of features. Mapping the data in linear SVM with higher dimensional (or large number of features) will improve the performance.

```
import nltk.classify
from sklearn.svm import LinearSVC
```

```
SVM = svm.SVC(C=1.0, kernel='linear', degree=3, gamma='auto')
```

```
SVM.fit(X_train, y_train)
predictions_SVM = SVM.predict(X_test)
```

Here is the result. The accuracy rate is 0.7546 which is mediocre compared with the rest. Therefore, it is better to choose the naïve Bayes model with the word features featureset.

```
print("Support Vector Machine Accuracy:",'\n',accuracy_score(y_test, predictions_SVM))
print("Support Vector Machine Matrix:",'\n',confusion_matrix(y_test,predictions_SVM))
print("Support Vector Machine Report:",'\n',classification_report(y_test,predictions_SVM))
```

```
Support Vector Machine Accuracy:
 0.7546108158799625
Support Vector Machine Matrix:
 [[1189  371]
 [ 414 1225]]
Support Vector Machine Report:
               precision    recall  f1-score   support

         neg       0.74      0.76      0.75      1560
         pos       0.77      0.75      0.76      1639

    accuracy                           0.75      3199
   macro avg       0.75      0.75      0.75      3199
weighted avg       0.75      0.75      0.75      3199
```

## Label Review Text

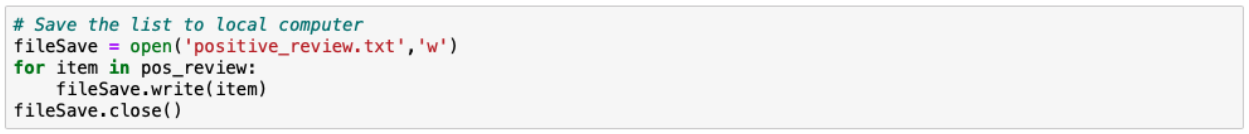
Add label by using the Naïve Bayes classifier with the word_features feature set (which contains stopwords and punctuations).

```
class_review = []
for sent in parsed_sentence_lower:
    feat = document_features(sent, word_features)
    tag = classifier.classify(feat)
    class_review += ((sent, tag),)
```

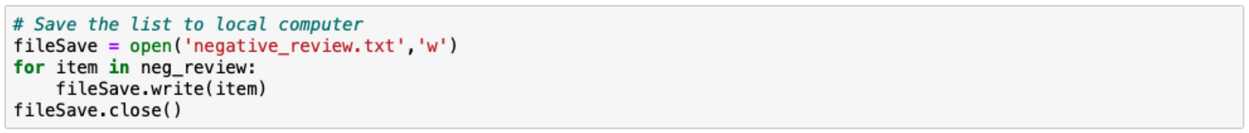
Then, we can split the class_review into two types by using the tag we obtained from above.

```
pos_review = []
neg_review = []
for (sent, tag) in class_review:
    temp = ''.join(sent)
    if tag == "pos":
        pos_review += (temp,)
    else:
        neg_review += (temp,)
```

Saving the pos_review and neg_review objects to local computer

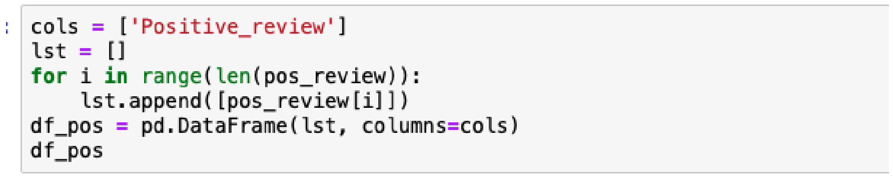


```python
# Save the list to local computer
fileSave = open('positive_review.txt','w')
for item in pos_review:
    fileSave.write(item)
fileSave.close()
```

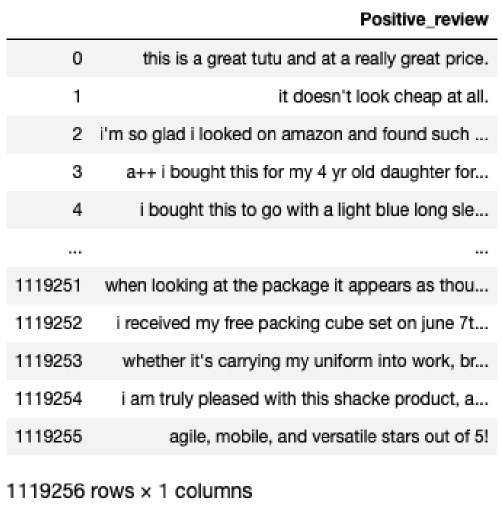


```python
# Save the list to local computer
fileSave = open('negative_review.txt','w')
for item in neg_review:
    fileSave.write(item)
fileSave.close()
```

Then, we can create a dataframe to list those positive and negative reviews. The positive dataframe has 1119256 records while the negative dataframe has 82830 records.

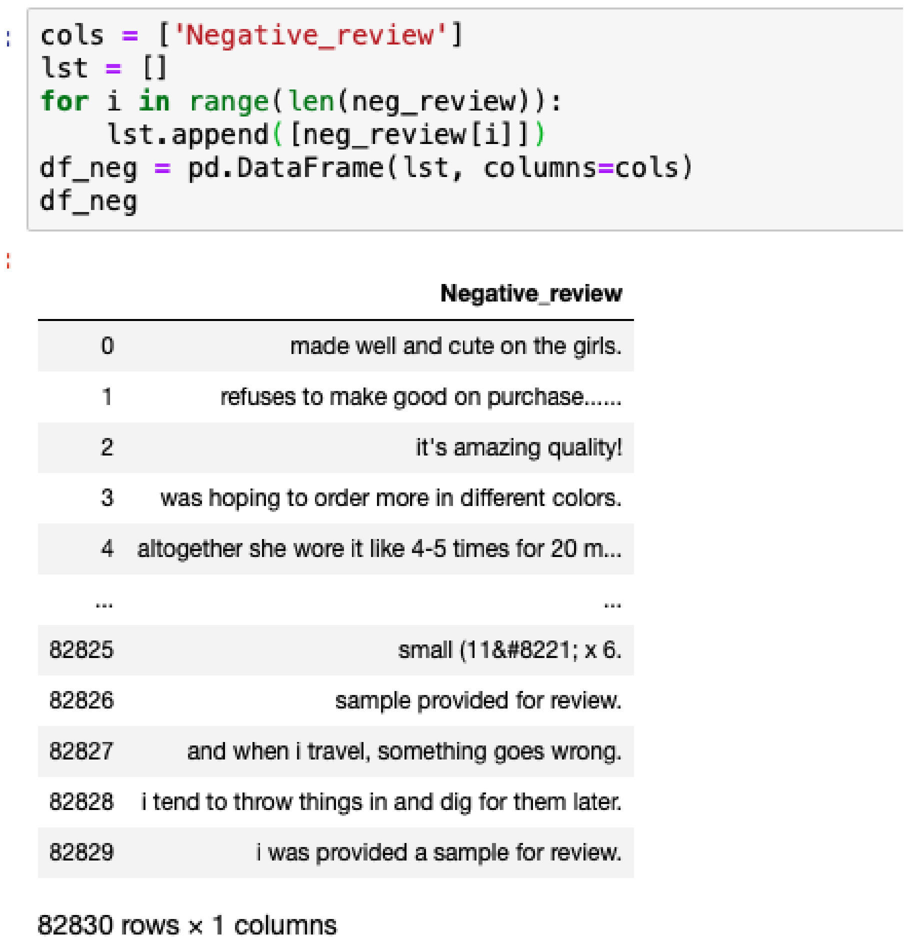


```python
cols = ['Positive_review']
lst = []
for i in range(len(pos_review)):
    lst.append([pos_review[i]])
df_pos = pd.DataFrame(lst, columns=cols)
df_pos
```



| | Positive_review |
|---|---|
| 0 | this is a great tutu and at a really great price. |
| 1 | it doesn't look cheap at all. |
| 2 | i'm so glad i looked on amazon and found such ... |
| 3 | a++ i bought this for my 4 yr old daughter for... |
| 4 | i bought this to go with a light blue long sle... |
| ... | ... |
| 1119251 | when looking at the package it appears as thou... |
| 1119252 | i received my free packing cube set on june 7t... |
| 1119253 | whether it's carrying my uniform into work, br... |
| 1119254 | i am truly pleased with this shacke product, a... |
| 1119255 | agile, mobile, and versatile stars out of 5! |

1119256 rows × 1 columns

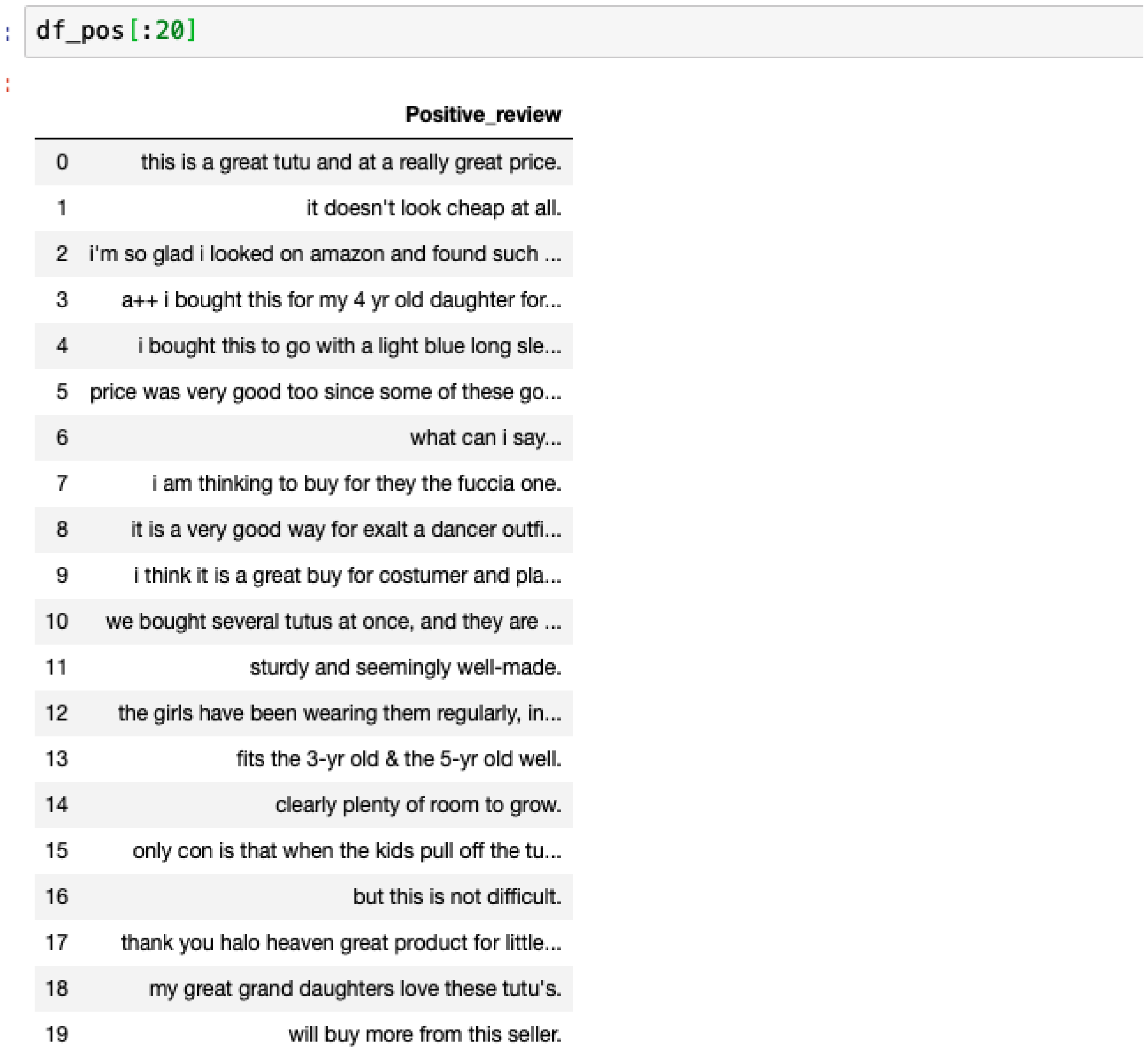


```
cols = ['Negative_review']
lst = []
for i in range(len(neg_review)):
    lst.append([neg_review[i]])
df_neg = pd.DataFrame(lst, columns=cols)
df_neg
```

| | Negative_review |
|---|---|
| 0 | made well and cute on the girls. |
| 1 | refuses to make good on purchase...... |
| 2 | it's amazing quality! |
| 3 | was hoping to order more in different colors. |
| 4 | altogether she wore it like 4-5 times for 20 m... |
| ... | ... |
| 82825 | small (11&#8221; x 6. |
| 82826 | sample provided for review. |
| 82827 | and when i travel, something goes wrong. |
| 82828 | i tend to throw things in and dig for them later. |
| 82829 | i was provided a sample for review. |

82830 rows × 1 columns

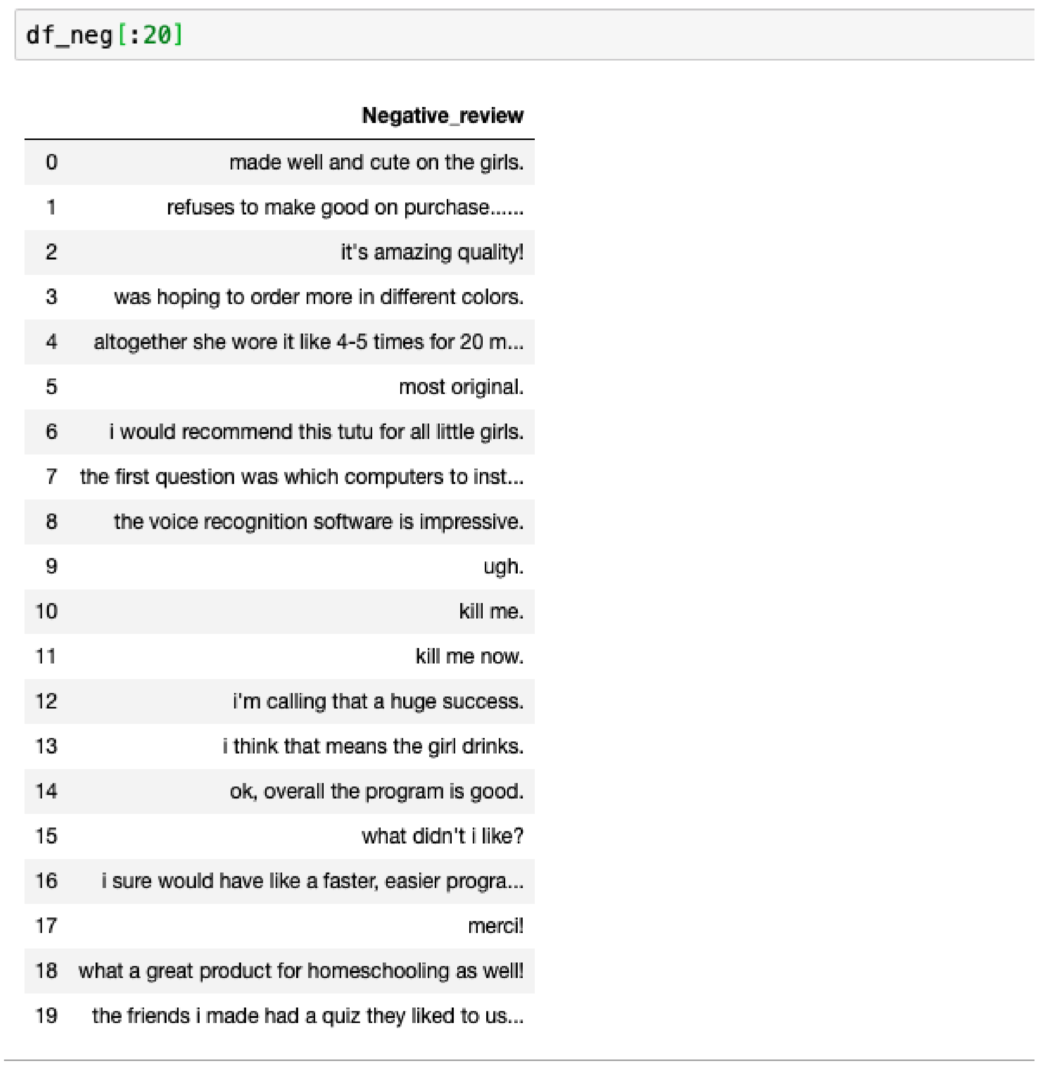
Due to the space limits, I only display the top 20 senetences from those two dataframes.
From the positive_review dataframe, we can see that the top 20 sentences are all positive reviews by customers.



```
df_pos[:20]
```

| | Positive_review |
|---|---|
| 0 | this is a great tutu and at a really great price. |
| 1 | it doesn't look cheap at all. |
| 2 | i'm so glad i looked on amazon and found such ... |
| 3 | a++ i bought this for my 4 yr old daughter for... |
| 4 | i bought this to go with a light blue long sle... |
| 5 | price was very good too since some of these go... |
| 6 | what can i say... |
| 7 | i am thinking to buy for they the fuccia one. |
| 8 | it is a very good way for exalt a dancer outfi... |
| 9 | i think it is a great buy for costumer and pla... |
| 10 | we bought several tutus at once, and they are ... |
| 11 | sturdy and seemingly well-made. |
| 12 | the girls have been wearing them regularly, in... |
| 13 | fits the 3-yr old & the 5-yr old well. |
| 14 | clearly plenty of room to grow. |
| 15 | only con is that when the kids pull off the tu... |
| 16 | but this is not difficult. |
| 17 | thank you halo heaven great product for little... |
| 18 | my great grand daughters love these tutu's. |
| 19 | will buy more from this seller. |

From the negative dataframe, we can see that most of the top 20 sentences are negative reviews. However, some sentences seem strange (such as the first one) which caused by the issue of sentence tokenization. What's more,



```python
df_neg[:20]
```

| | Negative_review |
|---|---|
| 0 | made well and cute on the girls. |
| 1 | refuses to make good on purchase...... |
| 2 | it's amazing quality! |
| 3 | was hoping to order more in different colors. |
| 4 | altogether she wore it like 4-5 times for 20 m... |
| 5 | most original. |
| 6 | i would recommend this tutu for all little girls. |
| 7 | the first question was which computers to inst... |
| 8 | the voice recognition software is impressive. |
| 9 | ugh. |
| 10 | kill me. |
| 11 | kill me now. |
| 12 | i'm calling that a huge success. |
| 13 | i think that means the girl drinks. |
| 14 | ok, overall the program is good. |
| 15 | what didn't i like? |
| 16 | i sure would have like a faster, easier progra... |
| 17 | merci! |
| 18 | what a great product for homeschooling as well! |
| 19 | the friends i made had a quiz they liked to us... |

Finally, we can save those dataframes as csv files.

```python
df_pos.to_csv('positive.csv')
```

```python
df_neg.to_csv('negative.csv')
```

## Limitations:

Due to the time limits and CPU limitation, we cannot test all the possibilities in this homework. Here are some points we can improve in the next exercise:

1. **Better Datasets:** The dataset we used to train the model is about movie review. It would be better to find another tagged dataset which focus on online shopping review or social media commends.
2. **More Data Pre-processing techniques:** Play around with the Data pre-processing steps (such as l**emmatization**) and see how it effects the accuracy.
3. **Another Word Vectorization technique:** Try other Word Vectorization techniques such as Word2Vec.

4. **Optimization of Parameter Selection:** Try Parameter tuning with the help of GridSearchCV on these Algorithms.
5. **More Classification Algorithms:** Try other classification Algorithms Like Linear Classifier, Boosting Models and even Neural Networks.