NLP Assignment 1

Name: Bo Li

Date: 2/12/2020

2. Data Pre-processing:

1. Split contents in “clothing_shoes_jewelry.txt” into lines using ‘splitlines()’.

```
lines = contents.splitlines()
print(len(lines))
print(type(lines))
print(lines[:10])
```

```
2786318
<class 'list'>
['reviewerID:A1KLRMWW2FWPL4', 'asin:0000031887', 'reviewerName:Amazon Customer "cameramom"', 'helpful:[0, 0]', "revie
wText:This is a great tutu and at a really great price. It doesn't look cheap at all. I'm so glad I looked on Amazon
and found such an affordable tutu that isn't made poorly. A++", 'overall:5.0', 'summary:Great tutu-  not cheaply mad
e', 'unixReviewTime:1297468800', 'reviewTime:02 12, 2011', '']
```

2. Extract lines only including “reviewText” to retrieve all review texts.

```
# Remove title "reviewText:"
reviewText = [r[11:] for r in lines if "reviewText" in r ] # r[11:] is contents after 'reviewText:'
print(len(reviewText))
print(reviewText[:3])
```

```
278677
["This is a great tutu and at a really great price. It doesn't look cheap at all. I'm so glad I looked on Amazon and
found such an affordable tutu that isn't made poorly. A++", 'I bought this for my 4 yr old daughter for dance class,
she wore it today for the first time and the teacher thought it was adorable. I bought this to go with a light blue l
ong sleeve leotard and was happy the colors matched up great. Price was very good too since some of these go for over
$15.00 dollars.', 'What can I say... my daughters have it in orange, black, white and pink and I am thinking to buy f
or they the fuccia one. It is a very good way for exalt a dancer outfit: great colors, comfortable, looks great, easy
to wear, durables and little girls love it. I think it is a great buy for costumer and play too.']
```

3. Save all reviews into a file.

```
# save extracted reviews into file
saveFile = open('/Users/boli/Desktop/nlp/HW1/reviews.txt', 'w')

for r in reviewText:
    saveFile.write(r + '\n')

saveFile.close()
```

4. Using nltk.word_tokenize to tokenize all review text to get single token of each word.
   - nltk.word_tokenize could be used to split tokens based on white space and punctuation in a string.

```
# word_tokenizer
tokens = []

start = time.time()

for t in reviewText:
    tokens += nltk.word_tokenize(t)

print('time:\t ' , time.time()-start)

print(len(tokens))
print(tokens[:20])
```

```
time:     228.00179386138916
19218012
['This', 'is', 'a', 'great', 'tutu', 'and', 'at', 'a', 'really', 'great', 'price', '.', 'It', 'does', "n't", 'look',
'cheap', 'at', 'all', '.']
```

5. Convert all uppercase characters to lowercase in tokens.
   - Using lowercase could be easily identified tokens without uppercase affect.
     e.g. This -> this, it's same as 'this', will be counted together then.

```
# Lowercase
start = time.time()
lowercase = [w.lower() for w in tokens]

print('time:\t ' , time.time()-start)

print(len(lowercase))
print(type(lowercase))
print(lowercase[:20])
```

```
time:     2.9568090438842773
19218012
<class 'list'>
['this', 'is', 'a', 'great', 'tutu', 'and', 'at', 'a', 'really', 'great', 'price', '.', 'it', 'does', "n't", 'look',
'cheap', 'at', 'all', '.']
```

6. Using isalpha() to get all alphabet tokens only.
   - We only care about tokens with alphabet, no punctuations, other characters or numbers.

```
# alphabets
start = time.time()
alphaWords = [w for w in lowercase if w.isalpha()]

print('time:\t ' , time.time()-start)

print(len(alphaWords))
print(type(alphaWords))
print(alphaWords[:20])
```

```
time:     1.952406883239746
16339294
<class 'list'>
['this', 'is', 'a', 'great', 'tutu', 'and', 'at', 'a', 'really', 'great', 'price', 'it', 'does', 'look', 'cheap', 'a
t', 'all', 'i', 'so', 'glad']
```

7. Implement WordNetLemmatizer to lemmatize all tokens.
   - We here only care about verb, noun, adjective, adverb since these are most frequently appearing in the text.
   - Using WordNetLemmatizer to reduce the word forms to linguistically valid lemmas, in order that convert tokens into the form what user expected.
     e.g. cats -> cat, is -> be. etc.

```
# Lemmatization
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet

start = time.time()

def get_wordnet_pos(word):
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}

    return tag_dict.get(tag, wordnet.NOUN)

# Init Lemmatizer
lemmatizer = WordNetLemmatizer()
lemmatizedWords = [lemmatizer.lemmatize(w, get_wordnet_pos(w)) for w in alphaWords]

print('time:\t ' , time.time()-start)

print(len(lemmatizedWords))
print(type(lemmatizedWords))
print(lemmatizedWords[:20])
```

```
time:     2621.689451932907
16339294
<class 'list'>
['this', 'be', 'a', 'great', 'tutu', 'and', 'at', 'a', 'really', 'great', 'price', 'it', 'do', 'look', 'cheap', 'at',
'all', 'i', 'so', 'glad']
```

8. Import stopwords to remove all commonly used word such as "the", "a", "an", "in".
   - The reason using stopwords is we don't want these words taking up space in our database because they are useless data as pre-processing.

```
# Stop word list
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
filtered_words = [w for w in lemmatizedWords if not w in stop_words ]

print(len(filtered_words))
print(type(filtered_words))
print(filtered_words[:20])
```

```
7734920
<class 'list'>
['great', 'tutu', 'really', 'great', 'price', 'look', 'cheap', 'glad', 'look', 'amazon', 'found', 'affordable', 'tut
u', 'make', 'poorly', 'bought', 'yr', 'old', 'daughter', 'dance']
```

3. Data Analysis:

1. Import FreqDist from nltk to generate frequency distribution.

```
# frequency distribution with FreqDist
from nltk import FreqDist
fdist = FreqDist(filtered_words)
fdistkeys = list(fdist.keys())

print(len(fdistkeys))
print(type(fdistkeys))
print(fdistkeys[:20])
```

```
59347
<class 'list'>
['great', 'tutu', 'really', 'price', 'look', 'cheap', 'glad', 'amazon', 'found', 'affordable', 'make', 'poorly', 'bou
ght', 'yr', 'old', 'daughter', 'dance', 'class', 'wore', 'today']
```

2. Based on Frequency Distribution, get most common tokens in reviews.

```
# 50 words by frequency
topkeys = fdist.most_common(50)
for pair in topkeys:
    print (pair)
```

```
('wear', 108396)
('fit', 107945)
('size', 102016)
('like', 100675)
('look', 99305)
('shoe', 81969)
('love', 80277)
('great', 78821)
('get', 78391)
('well', 75696)
('would', 67419)
('one', 64271)
('good', 60579)
('comfortable', 57492)
('make', 56123)
('order', 55518)
('color', 52403)
('nice', 46256)
...
```

3. Using Regular Expression to identify alphabet filter to retrieve alphabet tokens only.

4. Invoke collocations to generate bigrams among the reviews.

```
from nltk.collocations import *

bigram_measures = nltk.collocations.BigramAssocMeasures()
finder = BigramCollocationFinder.from_words(lowercase)
scored = finder.score_ngrams(bigram_measures.raw_freq)

first = scored[0]
for bscore in scored[:20]:
    print(bscore)
```

```
(('.', 'i'), 0.014311573954683762)
(('.', 'the'), 0.004959028327029871)
((',', 'but'), 0.004068214756031997)
((',', 'and'), 0.0036093743723336212)
(('.', 'it'), 0.003552500643666993)
(('they', 'are'), 0.003226868627202439)
(('.', 'they'), 0.003019927347323958)
(('i', 'have'), 0.0027907673280670236)
(('in', 'the'), 0.0027346741171771563)
((',', 'i'), 0.0027245794205977185)
(('of', 'the'), 0.0025272645266326194)
(('it', 'is'), 0.0024971365404496573)

...
```

5. Apply alpha_filter defined by Regular Expression previously.

```
# alpha bigrams
finder.apply_word_filter(alpha_filter)
scored1 = finder.score_ngrams(bigram_measures.raw_freq)
for bscore in scored1[:50]:
    print(bscore)
```

```
(('they', 'are'), 0.003226868627202439)
(('i', 'have'), 0.0027907673280670236)
(('in', 'the'), 0.0027346741171771563)
(('of', 'the'), 0.0025272645266326194)
(('it', 'is'), 0.0024971365404496573)
(('and', 'i'), 0.002308303272992024)
(('and', 'the'), 0.002118845591312983)
(('is', 'a'), 0.001964042898922115)
(('i', 'am'), 0.0019086781712905582)
(('a', 'little'), 0.0017551763418609584)
(('on', 'the'), 0.0017513778220140564)
(('i', 'was'), 0.001632322843798828)
(('i', 'love'), 0.0016165043501898116)
(('for', 'a'), 0.0015732116308388193)
(('this', 'is'), 0.0015457374050968436)
```

...

6. Remove stop words, to get Bigrams result

```
# stopword bigrams  -> Bigrams Result
finder.apply_word_filter(lambda w: w in stop_words)
scored2 = finder.score_ngrams(bigram_measures.raw_freq)
for bscore in scored2[:50]:
    print(bscore)
```

```
(('well', 'made'), 0.0004659691127261238)
(('would', 'recommend'), 0.00032719305201807554)
(('good', 'quality'), 0.0003203765301010323)
(('highly', 'recommend'), 0.0002809863996338435)
(('really', 'like'), 0.0002634507669159536 3)
(('fit', 'perfectly'), 0.0002323341248824 28)
(('fit', 'well'), 0.00022728677659270897)
(('look', 'like'), 0.00022275977348749703)
(('looks', 'great'), 0.00021282118046341 11)
(('another', 'pair'), 0.00020043696507214168)
(('look', 'great'), 0.0001960140310038312)
(('looks', 'like'), 0.00018826088775467515)
(('feel', 'like'), 0.00018227691813284329)
(('year', 'old'), 0.00018175657294833617)
(('great', 'price'), 0.00017504412006819436)
(('even', 'though'), 0.00016978863370467246)
(('fit', 'great'), 0.00016609418289467194)
(('usually', 'wear'), 0.00016177531786326285)
(('light', 'weight'), 0.00015324165683734613)
(('one', 'size'), 0.00014902686084283847)
```

...

7. Using PIM measure and filter tokens which frequency are less than 5

```
# Mutual Info Scored (min freq 5)
finder.apply_freq_filter(5)
scored3 = finder.score_ngrams(bigram_measures.pmi)
for bscore in scored3[:50]:
    print(bscore)
```

```
(('badgley', 'mischka'), 21.87402767411015)
(('salvatore', 'exte'), 21.87402767411015)
(('spatestruck', 'lenders'), 21.87402767411015)
(('tessuto', 'vela'), 21.87402767411015)
(('krav', 'maga'), 21.610993268276356)
(('pepto', 'bismol'), 21.610993268276353)
(('herman', 'munster'), 21.388600846939905)
(('hypo', 'allergenic'), 21.388600846939905)
(('birko', 'flor'), 21.19595576899751)
(('myia', 'passiello'), 21.19595576899751)
(('norman', 'reedus'), 21.19595576899751)
(('hola', 'gente'), 21.16620842560346)
(('saudi', 'arabia'), 21.16620842560346)
(('charlotte', 'russe'), 20.874027674110 15)
(('giorgio', 'brutini'), 20.874027674110145)
(('grady', 'harp'), 20.73652415036021)
(('sherpani', 'soleil'), 20.710528941827267)
(('laurel', 'burch'), 20.584521056915165)
```

...

4. Interpretation of the results:

1. Top words by frequency:

   In the result of the word frequency, we could see that most common appearing tokens relate to products (clothes, shoes, jewelry). e.g. “wear”, “fit”, “size”, “like”, “love”, “great”, “well”, “comfortable”, and rest of most common tokens all could be related to clothes, shoes, jewelry which we expect.

   The data could show how much customers care about on the priority below:

   Quality > Time >= Price > … (so on) where “quality” includes “wear”, “fit”, “size” …

2. Top bigrams by frequency:

   From the output of bigrams, we could see that most of bigrams are positive by customers’ reviews such that “well made”, “would recommend”, “good quality”, etc. Also, these bigrams describe products (clothes, shoes, jewelry) directly that we expect.

   In this result, we also could observe that the quality is the most concern by customers. The price and time they also concern but seems they are less important than quality.

3. Top bigrams by MI scores (using min Freq 5):

   In the result of bigrams by MI score, we get bigrams which does not look like English word groups. However, when we search them from reviews, we could obviously see that they are probably brands of products or use some other languages (non-English) that customers reviewed. Some of them are probably names of customers/sellers. Or some of them are word groups that people don’t speak/use often which is uncommon.

Addition Analysis:

In this project, I don’t think the current analysis methods I applied are sufficient enough to perform analysis. Here are some issues:

1. **Tokenizer:** I used word_tokenize, which is one of the most common tools to tokenize string using nltk. However, some tokens are not accurate to be tokenized. e.g. “doesn’t” convert to two tokens “does”, “n’t” which we don’t want to expect.
2. **Lemmatization**: I used WordNetlemmatizer to lemmatize tokens. In this part, I only care about four types of vocabulary: Noun, Verb, Adv, Adj. which means that it’s not 100% precisely lemmatize all tokens. e.g. “a”, “an”, “the” could be same type of vocabulary, but I do not lemmatize them as same. Also, using WordNetLemmatizer cannot lemmatize all tokens accurately, and the running time takes so long time to execute.

In order to improve these issues, I think the better way we could fix them out is that I could create specific regular expression to identify them and get the results what I expect. Or we could train some set of text to get more accurate POS tags for each token then.