CSE 258 Assignment 4 Linbin Yang A53277054

- When I process punctuations, I follow the rules list in Q7. For example, When I meet "Amazing!", I just split it as ["amazing", "!"]. After analyzing on the data, I found 182246 unique bigrams and the top-5-frequency bigrams are:
 [(4587, 'with-a'), (2595, 'in-the'), (2245, 'of-the'), (2056, 'is-a'), (2033, 'on-the')]
- I set the regularized regression parameters as: Regularization Parameter equals to 1.0 and No Interception. After selecting the 1000 most common bigrams and use frequency as feature. The MSE I got is: 0.3431530140613639
- The tf here is just the frequency of each word appears in the document, no normalization here.
 For the goal words: [foam', 'smell', 'banana', 'lactic', 'tart']
 I got the corresponding IDF:
 [1.1378686206869628, 0.5379016188648442, 1.67778 07052660807, 2.9208187539523753, 1.8068754016455384]
 I got the corresponding TF in document [0]:
 [2, 1, 2, 2, 1]
 I got the final TF-IDF:
 [2.2757372413739256,0.5379016188648442, 3.3555614105321614, 5.841637507904751, 1.8068754016455384]
- Here I note that if you do not use the 1000 most frequency words the MSE is: 0.06691778465356775
 However, If we use the 1000 most frequency words, The MSE changes to: 0.10629834153948432
 I did both of them and the corresponding TF-IDF defaultdicts are listed in my report.
- For this question, I use 1000 most frequency words to build features, here is the result I got:
 Beer Name: Frog's Hollow Double Pumpkin Ale
 Profile Name: Heatwave33
 Max Similarity: 0.31711492224280974
- Use top 1000 frequency unigrams to build tf-idf features, I got the MSE: 0.27875971411652656
- To solve this problem, we need to build input on train data and validation data. All together there 8 combinations:

Combinations	**MSE**
Unigram + remove + frequency	0.6333152912356144
Unigram + remove + tfidf	0.6414039617562557
Unigram + not remove + frequency	0.5984097464419731
Unigram + not remove + tfidf	0.6055174190540394
Bigram + remove + frequency	0.6620016499094443
Bigram + remove + tfidf	0.6681862382828353
Bigram + not remove + frequency	0.6507473729743735
Bigram + not remove + tfidf	0.6593053798364054

 It is better to remove punctuations and have tf-idf-bigrams features to train model.

In [1]:

```
import numpy as np
import urllib.request
import scipy.optimize
import random
from collections import defaultdict
import nltk
import string
from nltk.stem.porter import *
from sklearn import linear_model
```

In [2]:

```
def parseData(fname):
    for l in urllib.request.urlopen(fname):
        yield eval(l)
```

In [3]:

```
print ("Reading data......")
data = list(parseData("http://jmcauley.ucsd.edu/cse190/data/beer/beer_50000.js
on"))[:5000]
print ("done")
```

```
Reading data......
done
```

In [175]:

```
print (data[1])
```


```
{'review/appearance': 3.0, 'beer/style': 'English Strong Ale', 're
view/palate': 3.0, 'review/taste': 3.0, 'beer/name': 'Red Moon', '
review/timeUnix': 1235915097, 'beer/ABV': 6.2, 'beer/beerId': '482
13', 'beer/brewerId': '10325', 'review/timeStruct': {'isdst': 0, '
mday': 1, 'hour': 13, 'min': 44, 'sec': 57, 'mon': 3, 'year': 2009
, 'yday': 60, 'wday': 6}, 'review/overall': 3.0, 'review/text': 'D
ark red color, light beige foam, average.\tIn the smell malt and c
aramel, not really light.\tAgain malt and caramel in the taste, no
t bad in the end.\tMaybe a note of honey in teh back, and a light
fruitiness.\tAverage body.\tIn the aftertaste a light bitterness,
with the malt and red fruit.\tNothing exceptional, but not bad, dr
inkable beer.', 'user/profileName': 'stcules', 'review/aroma': 2.5
}
```

In [4]:

```
# WordCount Eliminate all the puctuations
wordCount = defaultdict(int)
punctuation = set(string.punctuation)
for d in data:
    r = ''.join([c for c in d['review/text'].lower() if not c in punctuation])
    for w in r.split():
        wordCount[w] += 1
```

In [6]:

```
print (f"All together there are {len(wordCount)} words")
```

```
All together there are 19426 words
```

In [7]:

```
counts_w = [(wordCount[w], w) for w in wordCount]
counts_w.sort()
counts_w.reverse()
words = [x[1] for x in counts_w[:1000]]
wordId = dict(zip(words, range(len(words))))
```

In [8]:

```
text_list = defaultdict(list)
for i in range(len(data)):
    r = ''.join([c for c in data[i]['review/text'].lower() if not c in punctua
tion])
    text_list[i] = r.split()
```

In [9]:

```
text_list_with_punc = defaultdict(list)
for i in range(len(data)):
    r = ''.join([c if not c in punctuation else ' '+c+' ' for c in data[i]['re
view/text'].lower()])
    text_list_with_punc[i] = r.split()
```

In [10]:

```
print (text_list[0])
```

```
['a', 'lot', 'of', 'foam', 'but', 'a', 'lot', 'in', 'the', 'smell'
, 'some', 'banana', 'and', 'then', 'lactic', 'and', 'tart', 'not',
'a', 'good', 'start', 'quite', 'dark', 'orange', 'in', 'color', 'w
ith', 'a', 'lively', 'carbonation', 'now', 'visible', 'under', 'th
e', 'foam', 'again', 'tending', 'to', 'lactic', 'sourness', 'same'
, 'for', 'the', 'taste', 'with', 'some', 'yeast', 'and', 'banana']
```

In [11]:

```
print (text_list_with_punc[0])
```

```
['a', 'lot', 'of', 'foam', '.', 'but', 'a', 'lot', '.', 'in', 'the
', 'smell', 'some', 'banana', ',', 'and', 'then', 'lactic', 'and',
'tart', '.', 'not', 'a', 'good', 'start', '.', 'quite', 'dark', 'o
range', 'in', 'color', ',', 'with', 'a', 'lively', 'carbonation',
'(', 'now', 'visible', ',', 'under', 'the', 'foam', ')', '.', 'aga
in', 'tending', 'to', 'lactic', 'sourness', '.', 'same', 'for', 't
he', 'taste', '.', 'with', 'some', 'yeast', 'and', 'banana', '.']
```

In [62]:

```
# compute the freq of word in all documents
each_word_freq_doc = defaultdict(int)
for each_word in wordCount:
    freq = 0
    for i in range(len(text_list)):
        if each_word in text_list[i]:
            freq += 1
    each_word_freq_doc[each_word] = freq
```

In [63]:

```
each_word_freq_doc_1 = defaultdict(int)
for each_word in words:
    freq = 0
    for i in range(len(text_list)):
        if each_word in text_list[i]:
            freq += 1
    each_word_freq_doc_1[each_word] = freq
```

In [64]:

```
print (len(each_word_freq_doc))
```

```
19426
```

In [73]:

```
print (len(each_word_freq_doc_1))
```

```
1000
```

In [13]:

```
Bigram = defaultdict(int)
punctuation = set(string.punctuation)
for d in data:
    r = ''.join([c for c in d['review/text'].lower() if not c in punctuation])
    for i in range(len(r.split())-1):
        Bigram[r.split()[i]+"-"+r.split()[i+1]] += 1
```

In [151]:

```
print (len(Bigram))
```

```
182246
```

In [14]:

```
print (max(zip(Bigram.values(),Bigram.keys())))
```

```
(4587, 'with-a')
```

In [15]:

```
print(Bigram['with-a'])
```

```
4587
```

In [16]:

```
print (Bigram['deal-with'])
```

```
5
```

In [17]:

```
counts = [(Bigram[biw], biw) for biw in Bigram]
counts.sort()
counts.reverse()
```

In [18]:

```
bi_words = [x[1] for x in counts[:1000]]
```

In [23]:

```
top5freq = counts[:5]
```

In [25]:

```
# List the 5 most-frequently-occurring bigrams along with their number of occu
rrences in the corpus
print (top5freq)
```

```
[(4587, 'with-a'), (2595, 'in-the'), (2245, 'of-the'), (2056, 'is-
a'), (2033, 'on-the')]
```

In [26]:

```
# sentiment analysis
bi_wordID = dict(zip(bi_words, range(len(bi_words))))
```

In [27]:

```
# print (bi_wordID)
bi_wordSet = set(bi_words)
```

In [28]:

```
def feature(datum):
    feat = [0]*len(bi_words)
    r = ''.join([c for c in datum['review/text'].lower() if not c in punctuati
on])
    for i in range(len(r.split())-1):
        bi_unit = r.split()[i]+"-"+r.split()[i+1]
        if  bi_unit in bi_words:
            feat[bi_wordID[bi_unit]] += 1
    feat.append(1)
    return feat
```

In [30]:

```
X = [feature(d) for d in data]
```

In [32]:

```
# print the shape of the feature matrix
print (len(X))
print (len(X[0]))
```

```
5000
1001
```

In [33]:

```
Y = [d['review/overall'] for d in data]
```

In [38]:

```
clf = linear_model.Ridge(1.0, fit_intercept=False)
clf.fit(X,Y)
theta = clf.coef_
predictions = clf.predict(X)
```

In [39]:

```
print (predictions)
```

```
[3.48471909 3.31957086 3.54264439 ... 5.20157626 3.53660705 4.2765
9128]
```

In [40]:

```
# report the MSE on the 5000 data
MSE = sum([(predictions[i]-Y[i])**2 for i in range(len(Y))])/len(Y)
```

In [41]:

```
print (f"MSE obtained using the new predictor: {MSE}")
```

```
MSE obtained using the new predictor: 0.3431530140613639
```

In [42]:

```
# total number of documents is N = 5000
# Compute IDF for 'foam', 'smell', 'banana', 'lactic', and 'tart'
goal_word = ['foam', 'smell', 'banana', 'lactic', 'tart']
freq = defaultdict(int)
for d in data:
    r = ''.join([c for c in d['review/text'].lower() if not c in punctuation])
    for elem in goal_word:
        if elem in r.split():
            freq[elem] += 1
print (freq)
```

```
defaultdict(<class 'int'>, {'foam': 364, 'smell': 1449, 'banana':
105, 'lactic': 6, 'tart': 78})
```

In [43]:

```
IDF = [np.log10(5000/freq[elem]) for elem in freq]
```

In [44]:

```
print (IDF)
```

```
[1.1378686206869628, 0.5379016188648442, 1.6777807052660807, 2.920
8187539523753, 1.8068754016455384]
```

In [45]:

```
# tf: number of times the term appears in the document
first_review = data[0]['review/text']
r = ''.join([c for c in first_review.lower() if not c in punctuation])
tf = [0]*len(goal_word)
for i in range(len(goal_word)):
    for elem in r.split():
        if elem == goal_word[i]:
            tf[i] += 1
print (tf)
```

```
[2, 1, 2, 2, 1]
```

In [46]:

```
# Compute TF-IDF
TF_IDF = [tf[i]*IDF[i] for i in range(len(tf))]
```

In [47]:

```
print (TF_IDF)
```

```
[2.2757372413739256, 0.5379016188648442, 3.3555614105321614, 5.841
637507904751, 1.8068754016455384]
```

In [48]:

```
def ComputeCosineSImilarity(x, y):
    res = 0
    for elem_x in x:
        if elem_x in y:
            res = res + x[elem_x]*y[elem_x]
    part1 = (sum([x[elem]**2 for elem in x]))**(1/2)
    part2 = (sum([y[elem]**2 for elem in y]))**(1/2)
    return res/(part1*part2)
```

In [49]:

```
def computeTF_IDF(gword, text_list, each_word_freq_doc, index):
    unit_freq = each_word_freq_doc[gword]
    tf = 0
    for elem in text_list[index]:
        if elem == gword:
            tf += 1
    return tf*(np.log10(5000/(1+unit_freq)))
```

In [77]:

```
def computeTF_IDF_1(gword, text_list, each_word_freq_doc_1, index):
    unit_freq = each_word_freq_doc_1[gword]
    tf = 0
    for elem in text_list[index]:
        if elem == gword:
            tf += 1
    return tf*(np.log10(5000/(1+unit_freq)))
```

In [67]:

```
# Build tf-idf vector
def tf_idf_builder(index, text_list, each_word_freq_doc):
    review_w = text_list[index]
    rev_vec = defaultdict(float)
    for elem in review_w:
        if elem not in rev_vec:
            rev_vec[elem] = computeTF_IDF(elem, text_list, each_word_freq_doc,
index)
    return rev_vec
```

In [68]:

```
def tf_idf_builder_1(index, text_list, words, each_word_freq_doc_1):
    review_w_1 = text_list[index]
    rev_vec_1 = defaultdict(float)
    for elem in review_w_1:
        if elem not in rev_vec_1 and elem in words:
            rev_vec_1[elem] = computeTF_IDF(elem, text_list, each_word_freq_do
c_1, index)
    return rev_vec_1
```

In [69]:

```
# construct tf-idf vector for review1 and review2
rev_1_vec = tf_idf_builder(0, text_list, each_word_freq_doc)
rev_2_vec = tf_idf_builder(1, text_list, each_word_freq_doc)
print (rev_1_vec)
print (rev_2_vec)
```

```
defaultdict(<class 'float'>, {'a': 0.02414000721952438, 'lot': 2.0
22882086242769, 'of': 0.05148923116234282, 'foam': 2.2733542797590
88, 'but': 0.1662156253435211, 'in': 0.3494074711380801, 'the': 0.
08645087743151567, 'smell': 0.5376020021010439, 'some': 0.67297505
91696887, 'banana': 3.347328278142497, 'and': 0.09763689545177756,
'then': 1.0034883278458213, 'lactic': 5.707743928643524, 'tart': 1
.8013429130455774, 'not': 0.28216313251307434, 'good': 0.370182803
9814841, 'start': 1.4975728800155672, 'quite': 0.8096683018297085,
'dark': 0.550984683652213 6, 'orange': 0.7894139750948435, 'color':
0.46042211665469096, 'with': 0.12416219470443926, 'lively': 1.9586
073148417749, 'carbonation': 0.36916548217194944, 'now': 1.4023048
140744876, 'visible': 1.9136401693252518, 'under': 1.7544873321858
503, 'again': 0.8781120148963188, 'tending': 2.9208187539523753, '
to': 0.1305335899191335, 'sourness': 2.086186147616283 3, 'same': 1
.2856702402547668, 'for': 0.28819277095880 9, 'taste': 0.2912392763
096833, 'yeast': 1.3027706572402824})
defaultdict(<class 'float'>, {'dark': 0.5509846836522136, 'red': 2
.4974417920333156, 'color': 0.46042211665469096, 'light': 1.746075
6584223573, 'beige': 1.749579997691106, 'foam': 1.136677139879544,
'average': 2.3849299438622933, 'in': 0.8735186778452002, 'the': 0.
14408479571919278, 'smell': 0.5376020021010439, 'malt': 1.16228413
78044713, 'and': 0.13018252726903676, 'caramel': 1.34242559929093 0
6, 'not': 0.846489397539223, 'really': 0.6352614449446014, 'again'
: 0.8781120148963188, 'taste': 0.2912392763096833, 'bad': 1.993078
935780987, 'end': 1.068542129310995, 'maybe': 1.1034737825104446,
'a': 0.018105005414643285, 'note': 1.413412695328245, 'of': 0.0514
892311623428 2, 'honey': 1.3809066693732572, 'teh': 2.9208187539523
753, 'back': 1.0132282657337552, 'fruitiness': 1.665546248849069,
'body': 0.5553307690614755, 'aftertaste': 0.9779842601822797, 'bit
terness': 0.6311554931741787, 'with': 0.06208109735221963, 'fruit'
: 1.014124642691606 4, 'nothing': 1.0305840876460186, 'exceptional'
: 1.9208187539523751, 'but': 0.1662156253435211, 'drinkable': 0.87
81120148963188, 'beer': 0.23942674605560585})
```

In [70]:

```
print (f"Cosine similarity between review1 and review2 is {ComputeCosineSImila
rity(rev_1_vec, rev_2_vec)}")
```

```
Cosine similarity between review1 and review2 is 0.066917784653567
75
```

In [71]:

```
rev_1_vec_1 = tf_idf_builder_1(0, text_list, words, each_word_freq_doc_1)
rev_2_vec_1 = tf_idf_builder_1(1, text_list, words, each_word_freq_doc_1)
print (rev_1_vec_1)
print (rev_2_vec_1)
```

```
defaultdict(<class 'float'>, {'a': 0.02414000721952438, 'lot': 2.0
22882086242769, 'of': 0.05148923116234282, 'foam': 2.2733542797590
88, 'but': 0.1662156253435211, 'in': 0.3494074711380801, 'the': 0.
08645087743151567, 'smell': 0.5376020021010439, 'some': 0.67297505
91696887, 'banana': 3.347328278142497, 'and': 0.09763689545177756,
'then': 1.0034883278458213, 'tart': 1.8013429130455774, 'not': 0.2
8216313251307434, 'good': 0.3701828039814841, 'start': 1.497572880
0155672, 'quite': 0.8096683018297085, 'dark': 0.5509846836522136,
'orange': 0.7894139750948435, 'color': 0.46042211665469096, 'with'
: 0.12416219470443926, 'carbonation': 0.36916548217194944, 'now':
1.4023048140744876, 'visible': 1.9136401693252518, 'under': 1.7544
873321858503, 'again': 0.8781120148963188, 'to': 0.130533589919133
5, 'same': 1.2856702402547668, 'for': 0.288192770958809, 'taste':
0.2912392763096833, 'yeast': 1.3027706572402824})
defaultdict(<class 'float'>, {'dark': 0.5509846836522136, 'red': 2
.4974417920333156, 'color': 0.46042211665469096, 'light': 1.746075
6584223573, 'beige': 1.749579997691106, 'foam': 1.136677139879544,
'average': 2.3849299438622933, 'in': 0.8735186778452002, 'the': 0.
14408479571919278, 'smell': 0.5376020021010439, 'malt': 1.16228413
78044713, 'and': 0.13018252726903676, 'caramel': 1.342425599290930
6, 'not': 0.846489397539223, 'really': 0.6352614449446014, 'again'
: 0.8781120148963188, 'taste': 0.2912392763096833, 'bad': 1.993078
935780987, 'end': 1.068542129310995, 'maybe': 1.1034737825104446,
'a': 0.018105005414643285, 'note': 1.413412695328245, 'of': 0.0514
8923116234282, 'honey': 1.3809066693732572, 'back': 1.013228265733
7552, 'fruitiness': 1.665546248849069, 'body': 0.5553307690614755,
'aftertaste': 0.9779842601822797, 'bitterness': 0.6311554931741787
, 'with': 0.0620810973522l963, 'fruit': 1.0141246426916064, 'nothi
ng': 1.0305840876460186, 'exceptional': 1.9208187539523751, 'but':
0.1662156253435211, 'drinkable': 0.8781120148963188, 'beer': 0.239
42674605560585})
```

In [72]:

```
print (f"Cosine similarity between review1 and review2 is {ComputeCosineSImila
rity(rev_1_vec_1, rev_2_vec_1)}")
```

```
Cosine similarity between review1 and review2 is 0.106298341539484
32
```

In [75]:

```
# beer name
# text_of_review
# profile_name
max_similarity = -1.0
beer_name = data[0]['beer/name']
text_of_review = data[0]['review/text']
profile_name = data[0]['user/profileName']
for i in range(1, len(data)):
    new_vec = tf_idf_builder_1(i, text_list, words, each_word_freq_doc_1)
    simi =  ComputeCosineSImilarity(rev_1_vec_1, new_vec)
    if simi > max_similarity:
        max_similarity = simi
        beer_name = data[i]['beer/name']
        text_of_review = data[i]['review/text']
        profile_name = data[i]['user/profileName']
    if i % 1000 == 0:
        print (simi)
```

```
0.04679825585465400 4
0.0823649190481172
0.02692744768455985

/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:8: Ru
ntimeWarning: invalid value encountered in double_scalars


0.014222534886462471
```

In [76]:

```
# output the goal_beer
print (beer_name)
print (text_of_review)
print (profile_name)
print (max_similarity)
```

```
Frog's Hollow Double Pumpkin Ale
Poured from a 22oz bottle to a Dogfish Head Snifter.
Color: Slight hazy orange with an off white head.
Smell: Cinnamon, banana, pumpkin and nutmeg.            Taste: Alc
ohol, pumpkin, nutmeg, allspice and a hint of banana.
Mouthfeel: Medium carbonation, smooth, medium dryness on the palat
e.                Overall: The smell is GREAT! The banana was a huge
surprise for me. The taste had too much alcohol presence. Seemed t
o overpower the other flavors. Cheers!
Heatwave33
0.31711492224280974
```

In [79]:

```
def new_feature(index, text_list, each_word_freq_doc_1):
    feat = [0]*len(words)
    goal_review = text_list[index]
    for w in goal_review:
        if w in words and feat[wordId[w]] == 0:
            feat[wordId[w]] = computeTF_IDF_1(w, text_list, each_word_freq_doc
_1, index)
    feat.append(1)
    return feat
```

In [80]:

```
X = [new_feature(index, text_list, each_word_freq_doc_1) for index in range(le
n(text_list))]
```

In [81]:

```
clf = linear_model.Ridge(1.0, fit_intercept=False)
clf.fit(X,Y)
theta = clf.coef_
predictions = clf.predict(X)
```

In [82]:

```
print (predictions)
```

```
[3.09655868 3.57328571 3.58483271 ... 4.290122   3.42816778 4.2491
8487]
```

In [83]:

```
MSE_TF_IDF = sum([(predictions[i]-Y[i])**2 for i in range(len(Y))])/len(Y)
```

In [84]:

```
print (f"MSE of predict model base on tfidf feature is {MSE_TF_IDF}")
```

```
MSE of predict model base on tfidf feature is 0.27875971411652656
```

In [85]:

```
# for question 7
# first we shuffle the data
print ("Reading data......")
data_all = list(parseData("http://jmcauley.ucsd.edu/cse190/data/beer/beer_5000
0.json"))
print ("done")
```

```
Reading data......
done
```

In [86]:

```
random.shuffle(data_all)
```

In [87]:

```
# split the data into training set, validation set and testing set
data_train = data_all[:5000]
data_validate = data_all[5000+1:5001+5000]
data_test = data_all[5001+5000+1:5001+5000+1+5000]
```

In [88]:

```
# we store data in memory-unigram
# remove punctuation or not remove
```

In [89]:

```
text_list_m = defaultdict(list)
for i in range(len(data_train)):
    r = ''.join([c for c in data_train[i]['review/text'].lower() if not c in p
unctuation])
    text_list_m[i] = r.split()
```

In [90]:

```
text_list_with_punc_m = defaultdict(list)
for i in range(len(data_train)):
    r = ''.join([c if not c in punctuation else ' '+c+' ' for c in data_train[
i]['review/text'].lower()])
    text_list_with_punc_m[i] = r.split()
```

In [341]:

```
# we store data in memory-bigram
# remove punctuation or not remove
```

In [91]:

```
bi_text_list_m = defaultdict(list)
for i in range(len(data_train)):
    bi_unit = []
    r = ''.join([c for c in data_train[i]['review/text'].lower() if not c in p
unctuation])
    for j in range(len(r.split())-1):
        bi_unit.append(r.split()[j]+"-"+r.split()[j+1])
    bi_text_list_m[i] = bi_unit
```

In [92]:

```
bi_text_list_with_punc_m = defaultdict(list)
for i in range(len(data_train)):
    bi_unit = []
    r = ''.join([c if not c in punctuation else ' '+c+' ' for c in data_train[
i]['review/text'].lower()])
    for j in range(len(r.split())-1):
        bi_unit.append(r.split()[j]+"-"+r.split()[j+1])
    bi_text_list_with_punc_m[i] = bi_unit
```

In [93]:

```
# word count
# remove punctuation or not remove
```

In [94]:

```
wordCount_m = defaultdict(int)
for i in range(len(text_list_m)):
    for w in text_list_m[i]:
        wordCount_m[w] += 1
```

In [95]:

```
wordCount_with_punc_m = defaultdict(int)
for i in range(len(text_list_with_punc_m)):
    for w in text_list_with_punc_m[i]:
        wordCount_with_punc_m[w] += 1
```

In [345]:

```
# bi_word count
# remove punctuation or not remove
```

In [96]:

```
bi_wordCount_m = defaultdict(int)
for i in range(len(bi_text_list_m)):
    for w in bi_text_list_m[i]:
        bi_wordCount_m[w] += 1
```

In [97]:

```
bi_wordCount_with_punc_m = defaultdict(int)
for i in range(len(bi_text_list_with_punc_m)):
    for w in bi_text_list_with_punc_m[i]:
        bi_wordCount_with_punc_m[w] += 1
```

In [101]:

```
# Select the top 1000 as features
counts_m = [(wordCount_m[w], w) for w in wordCount_m]
counts_m.sort()
counts_m.reverse()
words_m = [x[1] for x in counts_m[:1000]]
wordsId_m = dict(zip(words_m, range(len(words_m))))
# note here is changed!
```

In [102]:

```
counts_with_punc_m = [(wordCount_with_punc_m[w], w) for w in wordCount_with_pu
nc_m]
counts_with_punc_m.sort()
counts_with_punc_m.reverse()
words_with_punc_m = [x[1] for x in counts_with_punc_m[:1000]]
wordsId_with_punc_m = dict(zip(words_with_punc_m, range(len(words_with_punc_m)
)))
```

In [103]:

```
bi_counts_m = [(bi_wordCount_m[biw], biw) for biw in bi_wordCount_m]
bi_counts_m.sort()
bi_counts_m.reverse()
bi_words_m = [x[1] for x in bi_counts_m[:1000]]
bi_wordsId_m = dict(zip(bi_words_m, range(len(bi_words_m))))
```

In [104]:

```
bi_counts_with_punc_m = [(bi_wordCount_with_punc_m[biw], biw) for biw in bi_wo
rdCount_with_punc_m]
bi_counts_with_punc_m.sort()
bi_counts_with_punc_m.reverse()
bi_words_with_punc_m = [x[1] for x in bi_counts_with_punc_m[:1000]]
bi_wordsId_with_punc_m = dict(zip(bi_words_with_punc_m, range(len(bi_words_wit
h_punc_m))))
```

In [105]:

```
each_freq_doc_m = defaultdict(int)
for each_word in words_m:
    freq = 0
    for i in range(len(text_list_m)):
        if each_word in text_list_m[i]:
            freq += 1
    each_freq_doc_m[each_word] = freq
```

In [106]:

```
each_freq_doc_with_punc_m = defaultdict(int)
for each_word in words_with_punc_m:
    freq = 0
    for i in range(len(text_list_with_punc_m)):
        if each_word in text_list_with_punc_m[i]:
            freq += 1
    each_freq_doc_with_punc_m[each_word] = freq
```

In [107]:

```
bi_each_freq_doc_m = defaultdict(int)
for each_word in bi_words_m:
    freq = 0
    for i in range(len(bi_text_list_m)):
        if each_word in bi_text_list_m[i]:
            freq += 1
    bi_each_freq_doc_m[each_word] = freq
```

In [108]:

```
bi_each_freq_doc_with_punc_m = defaultdict(int)
for each_word in bi_words_with_punc_m:
    freq = 0
    for i in range(len(bi_text_list_with_punc_m)):
        if each_word in bi_text_list_with_punc_m[i]:
            freq += 1
    bi_each_freq_doc_with_punc_m[each_word] = freq
```

In [109]:

```
def computeTF_IDF(gword, content_list, each_freq, index):
    unit_freq = each_freq[gword]
    tf = 0
    for elem in content_list[index]:
        if elem == gword:
            tf += 1
    return tf*(np.log10(5000/(1+unit_freq)))
```

In [110]:

```
def tfidf_feature(index, contents_list, each_freq, unitset, unitid):
    feat = [0]*len(unitset)
    goal_review = contents_list[index]
    for w in goal_review:
        if w in unitset and feat[unitid[w]] == 0:
            feat[unitid[w]] = computeTF_IDF(w, contents_list, each_freq, index
)
    feat.append(1)
    return feat
```

In [111]:

```
def freq_feature(index, contents_list, unitset, unitid):
    feat = [0]*len(unitset)
    goal_review = contents_list[index]
    for w in goal_review:
        if  w in unitset:
            feat[unitid[w]] += 1
    feat.append(1)
    return feat
```

In [382]:

```
##############################################
```

In [112]:

```
p_text_list_m = defaultdict(list)
for i in range(len(data_validate)):
    r = ''.join([c for c in data_validate[i]['review/text'].lower() if not c in punctuation])
    p_text_list_m[i] = r.split()
```

In [113]:

```
p_text_list_with_punc_m = defaultdict(list)
for i in range(len(data_validate)):
    r = ''.join([c if not c in punctuation else ' '+c+' ' for c in data_validate[i]['review/text'].lower()])
    p_text_list_with_punc_m[i] = r.split()
```

In [114]:

```
p_bi_text_list_m = defaultdict(list)
for i in range(len(data_validate)):
    bi_unit = []
    r = ''.join([c for c in data_validate[i]['review/text'].lower() if not c in punctuation])
    for j in range(len(r.split())-1):
        bi_unit.append(r.split()[j]+"-"+r.split()[j+1])
    p_bi_text_list_m[i] = bi_unit
```

In [115]:

```
p_bi_text_list_with_punc_m = defaultdict(list)
for i in range(len(data_validate)):
    bi_unit = []
    r = ''.join([c if not c in punctuation else ' '+c+' ' for c in data_validate[i]['review/text'].lower()])
    for j in range(len(r.split())-1):
        bi_unit.append(r.split()[j]+"-"+r.split()[j+1])
    p_bi_text_list_with_punc_m[i] = bi_unit
```

In [116]:

```
p_wordCount_m = defaultdict(int)
for i in range(len(p_text_list_m)):
    for w in p_text_list_m[i]:
        p_wordCount_m[w] += 1
```

In [117]:

```
p_wordCount_with_punc_m = defaultdict(int)
for i in range(len(p_text_list_with_punc_m)):
    for w in p_text_list_with_punc_m[i]:
        p_wordCount_with_punc_m[w] += 1
```

In [118]:

```
p_bi_wordCount_m = defaultdict(int)
for i in range(len(p_bi_text_list_m)):
    for w in p_bi_text_list_m[i]:
        p_bi_wordCount_m[w] += 1
```

In [119]:

```
p_bi_wordCount_with_punc_m = defaultdict(int)
for i in range(len(p_bi_text_list_with_punc_m)):
    for w in p_bi_text_list_with_punc_m[i]:
        p_bi_wordCount_with_punc_m[w] += 1
```

In [120]:

```
p_counts_m = [(p_wordCount_m[w], w) for w in p_wordCount_m]
p_counts_m.sort()
p_counts_m.reverse()
p_words_m = [x[1] for x in p_counts_m[:1000]]
p_wordsId_m = dict(zip(p_words_m, range(len(p_words_m))))
```

In [121]:

```
p_counts_with_punc_m = [(p_wordCount_with_punc_m[w], w) for w in p_wordCount_w
ith_punc_m]
p_counts_with_punc_m.sort()
p_counts_with_punc_m.reverse()
p_words_with_punc_m = [x[1] for x in p_counts_with_punc_m[:1000]]
p_wordsId_with_punc_m = dict(zip(p_words_with_punc_m, range(len(p_words_with_p
unc_m))))
```

In [122]:

```
p_bi_counts_m = [(p_bi_wordCount_m[biw], biw) for biw in p_bi_wordCount_m]
p_bi_counts_m.sort()
p_bi_counts_m.reverse()
p_bi_words_m = [x[1] for x in p_bi_counts_m[:1000]]
p_bi_wordsId_m = dict(zip(p_bi_words_m, range(len(p_bi_words_m))))
```

In [124]:

```
p_bi_counts_with_punc_m = [(p_bi_wordCount_with_punc_m[biw], biw) for biw in p
_bi_wordCount_with_punc_m]
p_bi_counts_with_punc_m.sort()
p_bi_counts_with_punc_m.reverse()
p_bi_words_with_punc_m = [x[1] for x in p_bi_counts_with_punc_m[:1000]]
p_bi_wordsId_with_punc_m = dict(zip(p_bi_words_with_punc_m, range(len(p_bi_wor
ds_with_punc_m))))
```

In [125]:

```
p_each_freq_doc_m = defaultdict(int)
for each_word in p_words_m:
    freq = 0
    for i in range(len(p_text_list_m)):
        if each_word in p_text_list_m[i]:
            freq += 1
    p_each_freq_doc_m[each_word] = freq
```

In [126]:

```
p_each_freq_doc_with_punc_m = defaultdict(int)
for each_word in p_words_with_punc_m:
    freq = 0
    for i in range(len(p_text_list_with_punc_m)):
        if each_word in p_text_list_with_punc_m[i]:
            freq += 1
    p_each_freq_doc_with_punc_m[each_word] = freq
```

In [127]:

```
p_bi_each_freq_doc_m = defaultdict(int)
for each_word in p_bi_words_m:
    freq = 0
    for i in range(len(p_bi_text_list_m)):
        if each_word in p_bi_text_list_m[i]:
            freq += 1
    p_bi_each_freq_doc_m[each_word] = freq
```

In [128]:

```
p_bi_each_freq_doc_with_punc_m = defaultdict(int)
for each_word in p_bi_words_with_punc_m:
    freq = 0
    for i in range(len(p_bi_text_list_with_punc_m)):
        if each_word in p_bi_text_list_with_punc_m[i]:
            freq += 1
    p_bi_each_freq_doc_with_punc_m[each_word] = freq
```

In [129]:

```
pX_1 = [freq_feature(index, p_text_list_m, p_words_m, p_wordsId_m) for index in range(len(p_text_list_m))]
pX_2 = [tfidf_feature(index, p_text_list_m, p_each_freq_doc_m, p_words_m, p_wordsId_m) for index in range(len(p_text_list_m))]
pX_3 = [freq_feature(index, p_text_list_with_punc_m, p_words_with_punc_m, p_wordsId_with_punc_m) for index in range(len(p_text_list_with_punc_m))]
pX_4 = [tfidf_feature(index, p_text_list_with_punc_m, p_each_freq_doc_with_punc_m, p_words_with_punc_m, p_wordsId_with_punc_m) for index in range(len(p_text_list_with_punc_m))]
pX_5 = [freq_feature(index, p_bi_text_list_m, p_bi_words_m, p_bi_wordsId_m) for index in range(len(p_bi_text_list_m))]
pX_6 = [tfidf_feature(index, p_bi_text_list_m, p_bi_each_freq_doc_m, p_bi_words_m, p_bi_wordsId_m) for index in range(len(p_bi_text_list_m))]
pX_7 = [freq_feature(index, p_bi_text_list_with_punc_m, p_bi_words_with_punc_m, p_bi_wordsId_with_punc_m) for index in range(len(p_bi_text_list_with_punc_m))]
pX_8 = [tfidf_feature(index, p_bi_text_list_with_punc_m, p_bi_each_freq_doc_with_punc_m, p_bi_words_with_punc_m, p_bi_wordsId_with_punc_m) for index in range(len(p_bi_text_list_with_punc_m))]
```

In [130]:

```
Y_prime_val =  [d['review/overall'] for d in data_validate]
```

In [132]:

```
##############################################
```

In [131]:

```
Y = [d['review/overall'] for d in data_train]
```

In [133]:

```
# Unigram + remove + freq
X_1 = [freq_feature(index, text_list_m, words_m, wordsId_m) for index in range(len(text_list_m))]
```

In [134]:

```
# Unigram + remove + tfidf
X_2 = [tfidf_feature(index, text_list_m, each_freq_doc_m, words_m, wordsId_m) for index in range(len(text_list_m))]
```

In [135]:

```
# Unigram + not remove + freq
X_3 = [freq_feature(index, text_list_with_punc_m, words_with_punc_m, wordsId_with_punc_m) for index in range(len(text_list_with_punc_m))]
```

In [136]:

```
# Unigram + not remove + tfidf
X_4 = [tfidf_feature(index, text_list_with_punc_m, each_freq_doc_with_punc_m, words_with_punc_m, wordsId_with_punc_m) for index in range(len(text_list_with_punc_m))]
```

In [137]:

```
# Bigram + remove + freq
X_5 = [freq_feature(index, bi_text_list_m, bi_words_m, bi_wordsId_m) for index in range(len(bi_text_list_m))]
```

In [138]:

```
# Bigram + remove + tfidf
X_6 = [tfidf_feature(index, bi_text_list_m, bi_each_freq_doc_m, bi_words_m, bi_wordsId_m) for index in range(len(bi_text_list_m))]
```

In [139]:

```
# Bigram + not remove + freq
X_7 = [freq_feature(index, bi_text_list_with_punc_m, bi_words_with_punc_m, bi_wordsId_with_punc_m) for index in range(len(bi_text_list_with_punc_m))]
```

In [141]:

```
# Bigram + not remove + tfidf
X_8 = [tfidf_feature(index, bi_text_list_with_punc_m, bi_each_freq_doc_with_punc_m, bi_words_with_punc_m, bi_wordsId_with_punc_m) for index in range(len(bi_text_list_with_punc_m))]
```

In [142]:

```
def train_out_MSE(x, y, x_p, y_p):
    clf = linear_model.Ridge(1.0, fit_intercept=False)
    clf.fit(x,y)
    theta = clf.coef_
    predictions = clf.predict(x_p)
    MSE_temp = sum([(predictions[i]-y_p[i])**2 for i in range(len(y_p))])/len(y_p)
    return MSE_temp
```

In [143]:

```
# Unigram + remove + freq
print (train_out_MSE(X_1, Y, pX_1, Y_prime_val))
```

```
0.6333152912356144
```

In [144]:

```
# Unigram + remove + tfidf
print (train_out_MSE(X_2, Y, pX_2, Y_prime_val))
```

0.6414039617562557

In [145]:

```
# Unigram + not remove + freq
print (train_out_MSE(X_3, Y, pX_3, Y_prime_val))
```

0.5984097464419731

In [146]:

```
# Unigram + not remove + tfidf
print (train_out_MSE(X_4, Y, pX_4, Y_prime_val))
```

0.6055174190540394

In [147]:

```
# Bigram + remove + freq
print (train_out_MSE(X_5, Y, pX_5, Y_prime_val))
```

0.6620016499094443

In [148]:

```
# Bigram + remove + tfidf
print (train_out_MSE(X_6, Y, pX_6, Y_prime_val))
```

0.6681862382828353

In [149]:

```
# Bigram + not remove + freq
print (train_out_MSE(X_7, Y, pX_7, Y_prime_val))
```

0.6507473729743735

In [150]:

```
# Bigram + not remove + tfidf
print (train_out_MSE(X_8, Y, pX_8, Y_prime_val))
```

0.6593053798364054