TextMining

December 27, 2020

[1]:
```python
import numpy as np
from urllib.request import urlopen
import scipy.optimize
import random
from collections import defaultdict
import nltk
import string
from nltk.stem.porter import *
import gzip
from sklearn import linear_model
import math
from sklearn.metrics import mean_squared_error as MSE
```

[2]:
```python
def parse(f):
    for l in gzip.open(f):
        yield eval(l)
```

[3]:
```python
### Just the first 10000 reviews

print("Reading data...")
data = list(parse("data/train_Category.json.gz"))[:10000]
print("done")
```

```
Reading data…
done
```

[4]:
```python
data[0]
```

[4]:

```
{'userID': 'u74382925',
 'genre': 'Adventure',
 'early_access': False,
 'reviewID': 'r75487422',
 'hours': 4.1,
 'text': 'Short Review:\nA good starting chapter for this series, despite the
main character being annoying (for now) and a short length. The story is good
and actually gets more interesting. Worth the try.\nLong Review:\nBlackwell
Legacy is the first on the series of (supposedly) 5 games that talks about the
main protagonist, Rosangela Blackwell, as being a so called Medium, and in this
```

```
first chapter we get to know how her story will start and how she will meet her
adventure companion Joey…and really, that\'s really all for for now and
that\'s not a bad thing, because in a way this game wants to show how hard her
new job is, and that she cannot escape her destiny as a medium.\nMy biggest
complain for this chapter, except the short length, it\'s the main protagonist
being a "bit" too annoying to be likeable, and most of her dialogues will always
be about complaining or just be annoyed. Understandable, sure, but lighten\' up
will ya!?\nHowever, considering that in the next installments she will be much
more likeable and kind of interesting, I\'d say give it a shot and see if you
like it: if you hate this first game, you might like the next, or can always
stop here.\nI recommend it.',
 'genreID': 3,
 'date': '2014-02-07'}
```

[5]:
```
punctuation = set(string.punctuation)
for d in data:
    r = ''.join([c for c in d['text'].lower() if not c in punctuation])
    d['text'] = r
```

[6]:
```
from nltk import word_tokenize
from nltk.util import ngrams
```

[7]:
```
bigrams = defaultdict(int)

for d in data:
#     token = nltk.word_tokenize(d['text'])
#     bigram = list(ngrams(token, 2))
    text = " ".join(d['text'].splitlines())
    bigram = [b for b in zip(text.split(" ")[:-1], text.split(" ")[1:])]
    for b in bigram:
        bigrams[b] += 1
```

0.0.1 1. How many unique bigrams are there amongst the reviews? List the 5 most-frequently-occurring bigrams along with their number of occurrences in the corpus (1 mark).

257,124 unique bigrams amongst the 10000 reviews

[8]:
```
len(bigrams)
```

[8]: 257124

[ ]:
```

```

5 most frequently occuring bigrams with their number of occurences in the corpus

[9]:
```
sorted(bigrams.items(),key=lambda v: v[1],reverse=True)[:5]
```

[9]:
```
[(('this', 'game'), 4438),
 (('the', 'game'), 4249),
 (('of', 'the'), 3356),
 (('if', 'you'), 2018),
 (('in', 'the'), 2017)]
```

[10]:
```python
mostCommon =sorted(bigrams.items(),key=lambda v: v[1],reverse=True)[:1000]
```

[ ]:
```python

```

0.0.2 2. The code provided performs least squares using the 1000 most common unigrams. Adapt it to use the 1000 most common bigrams and report the MSE obtained using the new predictor (use bigrams only, i.e., not unigrams+bigrams) (1 mark). Note that the code performs regularized regression with a regularization parameter of 1.0. The prediction target should be log2 (hours + 1) (i.e., our transformed time variable).

[11]:
```python
bigram_words = [b[0] for b in mostCommon]
```

[12]:
```python
bigramId = dict(zip(bigram_words, range(len(bigram_words))))
bigramSet = set(bigram_words)
```

[13]:
```python
def feature_bigram(datum):
    feat = [0]*len(bigramSet)
    t = datum['text']
#     token = nltk.word_tokenize(t)
#     bigram_words = list(ngrams(token, 2))
#     t = t.lower() # lowercase string
#     t = [c for c in t] # non-punct characters
#     t = ''.join(t) # convert back to string
#     words = t.strip().split() # tokenizes
    text = " ".join(t.splitlines())
    bigram_words = [b for b in zip(text.split(" ")[:-1], text.split(" ")[1:])]

    for w in bigram_words:
        if not (w in bigramSet): continue
        feat[bigramId[w]] += 1
    feat.append(1)
    return feat
```

[14]:
```python
X = [feature_bigram(d) for d in data]
y = [math.log(d['hours'] + 1,2) for d in data]
```

[ ]:
```python

```

```
[15]: clf = linear_model.Ridge(1.0, fit_intercept=False)
clf.fit(X, y)
theta = clf.coef_
predictions = clf.predict(X)
```

```
[16]: print("mse: ", MSE(y, predictions))
```

```
mse:  4.399483733665732
```

0.0.3 3. Repeat the above experiment using unigrams and bigrams, still considering the 1000 most common. That is, your model will still use 1000 features (plus an offset), but those 1000 features will be some combination of unigrams and bigrams. Report the MSE obtained using the new predictor (1 mark).

```
[17]: unigrams = defaultdict(int)

for d in data:
#     token = nltk.word_tokenize(d['text'])
#     unigram = list(ngrams(token, 1))
    t = d['text']
    text = " ".join(t.splitlines())
    unigram = text.strip().split()
    for u in unigram:
        unigrams[u] += 1
```

```
[ ]:
```

```
[ ]:
```

```
[18]: ### get 1000 most common unigrams and bigrams from corpus (10,000 reviews)
mostCommonUni =sorted(unigrams.items(),key=lambda v: v[1],reverse=True)[:1000]
mostCommonBi =sorted(bigrams.items(),key=lambda v: v[1],reverse=True)[:1000]
```

```
[19]: combined = []
for i in mostCommonUni:
    combined.append(i)

for i in mostCommonBi:
    combined.append(i)
```

```
[20]: combined = sorted(combined, key = lambda x: x[1], reverse = True)[:1000]
```

```
[21]: combined[:20]
```

```
[21]: [('the', 34211),
 ('and', 19392),
```

```
 ('a', 18791),
 ('to', 18077),
 ('game', 15043),
 ('of', 14095),
 ('is', 13000),
 ('you', 12735),
 ('i', 12204),
 ('it', 11824),
 ('this', 9548),
 ('in', 8274),
 ('that', 7060),
 ('for', 6526),
 ('but', 6321),
 ('with', 5586),
 ('its', 5144),
 ('are', 4849),
 ('on', 4559),
 (('this', 'game'), 4438)]
```

[ ]:

[ ]:

[ ]:

[22]:
```
unigram_words = [u[0] for u in mostCommonUni]
bigram_words = [b[0] for b in mostCommonBi]
```

[23]:
```
combined_words = [w[0] for w in combined]
```

[ ]:

[24]:
```
unigramId = dict(zip(unigram_words, range(len(unigram_words))))
unigramSet = set(unigram_words)
bigramId = dict(zip(bigram_words, range(len(bigram_words))))
bigramSet = set(bigram_words)
```

[ ]:

[25]:
```
def feature_bi_and_uni(datum):
    feat = [0]*len(bigramSet)
    t = datum['text']

    text = " ".join(t.splitlines())
    bigram_words = [b for b in zip(text.split(" ")[:-1], text.split(" ")[1:])]
    unigram_words = text.strip().split()
```

```
#     for w in bigram_words:
#         if not (w in bigramSet): continue
#         feat[bigramId[w]] += 1

#     for w in unigram_words:
#         if not (w in unigramSet): continue
#         feat[unigramId[w]] += 1

    for w in combined_words:
        if not (w in bigramSet): continue
        feat[bigramId[w]] += 1

    for w in combined_words:
        if not (w in unigramSet): continue
        feat[unigramId[w]] += 1


    feat.append(1)
    return feat
```

[26]:
```
X = [feature_bi_and_uni(d) for d in data]
y = [math.log(d['hours'] + 1,2) for d in data]
```

[27]:
```
clf = linear_model.Ridge(1.0, fit_intercept=False)
clf.fit(X, y)
theta = clf.coef_
predictions = clf.predict(X)
```

[28]:
```
print("mse: ", MSE(y, predictions))
```

```
mse:  5.242478901430268
```

note the increase in MSE

0.0.4 another idea I had was to instead create a random combination of unigrams and bigrams as a feature, of course each option has a .5 chance of being incorporated as a feature so I wouldn't expect the model's performance to differ too much but i wanted to observe the results nonetheless

[29]:
```
def feature_bi_and_uni(datum):
    feat = [0]*len(bigramSet)
    t = datum['text']
#     token = nltk.word_tokenize(t)
#     bigram_words = list(ngrams(token, 2))
#     unigram_words = list(ngrams(token, 1))
```

```
    text = " ".join(t.splitlines())
    bigram_words = [b for b in zip(text.split(" ")[:-1], text.split(" ")[1:])]
    unigram_words = text.strip().split()

    uni_or_bi = random.choice(['uni', 'bi'])
    if uni_or_bi == 'bi':
        for w in bigram_words:
            if not (w in bigramSet): continue
            feat[bigramId[w]] += 1
    else:
        for w in unigram_words:
            if not (w in unigramSet): continue
            feat[unigramId[w]] += 1

    feat.append(1)
    return feat
```

[30]:
```
X = [feature_bi_and_uni(d) for d in data]
y = [math.log(d['hours'] + 1,2) for d in data]
```

[31]:
```
clf = linear_model.Ridge(1.0, fit_intercept=False)
clf.fit(X, y)
theta = clf.coef_
predictions = clf.predict(X)
```

[32]:
```
print("mse: ", MSE(y, predictions))
```

```
mse:  4.403605155664077
```

note the increase in mse

[ ]:
```

```

[ ]:
```

```

0.0.5 4. What is the inverse document frequency of the words 'destiny', 'annoying', 'likeable', 'chapter', and 'interesting'? What are their tf-idf scores in review ID r75487422 (using log base 10, unigrams only, following the first definition of tf-idf given in the slides) (1 mark)?

[33]:
```
#
docFreq = defaultdict(set)
for d in data:
#     token = nltk.word_tokenize(d['text'])
#     unigram = list(ngrams(token, 1))
    t = d['text']
    text = " ".join(t.splitlines())
```

```
    unigram = text.strip().split()

    for u in unigram:
        docFreq[u].add(d['reviewID'])
```

[34]:
```
# uniqueReviews = set()
# totalRevs = 0
# for d in data:
#     if d['reviewID'] in uniqueReviews: continue
#     uniqueReviews.add(d['reviewID'])
```

[35]:
```
idf_destiny = np.log10( len(data)/ len(docFreq['destiny']))
idf_annoying = np.log10( len(data)/ len(docFreq['annoying']))
idf_likeable = np.log10(len(data) / len(docFreq['likeable']))
idf_chapter = np.log10( len(data)/ len(docFreq['chapter']))
idf_interesting = np.log10( len(data)/ len(docFreq['interesting']))
print("idf-destiny:",idf_destiny )
print("idf-annoying:",idf_annoying)
print("idf-likeable:",idf_likeable)
print("idf-chapter:",idf_chapter)
print("idf-interesting:",idf_interesting)
```

```
idf-destiny: 3.3979400086720375
idf-annoying: 1.8386319977650252
idf-likeable: 3.0969100130080562
idf-chapter: 2.221848749616356
idf-interesting: 1.3585258894959005
```

[36]:
```
# r75487422 = data[0]['text']
tf = defaultdict(int)
for d in data:
#     token = nltk.word_tokenize(d['text'])
#     unigram = list(ngrams(token, 1))
    t = d['text']
    text = " ".join(t.splitlines())
    unigram = text.strip().split()
    for u in unigram:
        tf[u] +=1
    break
```

[37]:
```
tf_idf_destiny = tf['destiny'] * idf_destiny
tf_idf_annoying = tf['annoying'] * idf_annoying
tf_idf_likeable = tf['likeable'] * idf_likeable
tf_idf_chapter = tf['chapter'] * idf_chapter
tf_idf_interesting = tf['interesting'] * idf_interesting
print("tf_idf-destiny:",tf_idf_destiny )
print("tf_idf-annoying:",tf_idf_annoying)
```

```
print("tf_idf-likeable:",tf_idf_likeable)
print("tf_idf-chapter:",tf_idf_chapter)
print("tf_idf-interesting:",tf_idf_interesting)
```

```
tf_idf-destiny: 3.3979400086720375
tf_idf-annoying: 3.6772639955300503
tf_idf-likeable: 6.1938200260161125
tf_idf-chapter: 6.665546248849068
tf_idf-interesting: 2.717051778991801
```

0.0.6 5. Adapt your unigram model to use the tfidf scores of words, rather than a bag-of-words representation. That is, rather than your features containing the word counts for the 1000 most common unigrams, it should contain tfidf scores for the 1000 most common unigrams. Report the MSE of this new model (1 mark).

[38]:
```
docFreq = defaultdict(set)
for d in data:
#     token = nltk.word_tokenize(d['text'])
#     unigram = list(ngrams(token, 1))
    t = d['text']
    text = " ".join(t.splitlines())
    unigram = text.strip().split()
    for u in unigram:
        docFreq[u].add(d['reviewID'])
```

[39]:
```
tf = defaultdict(int)
for d in data:
#     token = nltk.word_tokenize(d['text'])
#     unigram = list(ngrams(token, 1))
    t = d['text']
    text = " ".join(t.splitlines())
    unigram = text.strip().split()
    for u in unigram:
        tf[u] +=1
```

[40]:
```
mostCommonUni =sorted(tf.items(),key=lambda v: v[1],reverse=True)[:1000]
unigram_words = [u[0] for u in mostCommonUni]
unigramId = dict(zip(unigram_words, range(len(unigram_words))))
unigramSet = set(unigram_words)
```

[41]:
```
def feature_uni(datum):
    feat = [0]*len(unigramSet)
    t = datum['text']
#     token = nltk.word_tokenize(t)
#     unigram_words = list(ngrams(token, 1))
    text = " ".join(t.splitlines())
```

```
    unigram_words= text.strip().split()

    for u in unigram_words:
        if not (u in unigramSet): continue
        tf_idf_word = np.log10(len(data)/ len(docFreq[u])) * tf[u]
        feat[unigramId[u]] = tf_idf_word

    feat.append(1)
    return feat
```

[42]:
```
X = [feature_uni(d) for d in data]
y = [math.log(d['hours'] + 1,2) for d in data]
```

[43]:
```
clf = linear_model.Ridge(1.0, fit_intercept=False)
clf.fit(X, y)
theta = clf.coef_
predictions = clf.predict(X)
```

[44]:
```
print("mse: ", MSE(y, predictions))
```

```
mse:  4.106655150599012
```

0.0.7 6. Which other review has the highest cosine similarity compared to review ID r75487422, in terms of their tf-idf representations (considering unigrams only). Provide the reviewID, or the text of the review (1 mark)?

[ ]:
```

```

[90]:
```
data[0]
```

[90]:
```
{'userID': 'u74382925',
 'genre': 'Adventure',
 'early_access': False,
 'reviewID': 'r75487422',
 'hours': 4.1,
 'text': 'short review\na good starting chapter for this series despite the main
character being annoying for now and a short length the story is good and
actually gets more interesting worth the try\nlong review\nblackwell legacy is
the first on the series of supposedly 5 games that talks about the main
protagonist rosangela blackwell as being a so called medium and in this first
chapter we get to know how her story will start and how she will meet her
adventure companion joeyand really thats really all for for now and thats not a
bad thing because in a way this game wants to show how hard her new job is and
that she cannot escape her destiny as a medium\nmy biggest complain for this
chapter except the short length its the main protagonist being a bit too
annoying to be likeable and most of her dialogues will always be about
complaining or just be annoyed understandable sure but lighten up will
```

```
ya\nhowever considering that in the next installments she will be much more
likeable and kind of interesting id say give it a shot and see if you like it if
you hate this first game you might like the next or can always stop here\ni
recommend it',
 'genreID': 3,
 'date': '2014-02-07'}
```

[45]:
```python
def cosineSim(s1, s2):
    numer = np.dot(s1,s2) # intersection of sets / dot product between sets
    denom = np.linalg.norm(s1) * np.linalg.norm(s2)# magnitude of s1 *␣
 ↪magnitude of s2
    dot_product = np.dot(a, b)
    if denom == 0:
        return 0
    else:
        return numer / denom
```

[46]:
```python
first_review = data[0]
cosinesims = []
Xfirst = feature_uni(first_review)
for d in data[1:]:
    review_i = feature_uni(d)
    cosinesims.append((d['reviewID'],cosineSim(Xfirst, review_i)))
```

[92]:
```python
sorted(cosinesims,key=lambda tup: tup[1], reverse = True)[0]
```

[92]:
```
('r89686923', 0.9020892284292362)
```

0.0.8 7. Implement a validation pipeline for this same data, by randomly shuffling the data, using 10,000 reviews for training, another 10,000 for validation, and another 10,000 for testing.1 Consider regularization parameters in the range {0.01, 0.1, 1, 10, 100}, and report MSEs on the test set for the model that performs best on the validation set. Using this pipeline, compare the following alternatives in terms of their performance (all using 1,000 dimensional word features):

- Unigrams vs. bigrams
- Removing punctuation vs. preserving it. The model that preserves punctuation should treat punctuation characters as separate words, e.g. “Amazing!” would become [‘amazing’, ‘!’]
- tfidf scores vs. word counts

0.0.9 In total you should compare $2 \times 2 \times 2 \times 5 = 40$ models (8 models and 5 regularization parameters), andproduce a table comparing their performance (2 marks)

[48]:
```
import pandas as pd
```

[49]:
```
full_data = list(parse("data/train_Category.json.gz"))
```

[50]:
```
X = [d for d in full_data]
y = [math.log(d['hours'] + 1,2) for d in full_data]
```

[51]:
```
#shuffle data
Xy = list(zip(X,y))
random.shuffle(Xy)
X = np.array([d[0] for d in Xy])
y = np.array([d[1] for d in Xy])
```

[52]:
```
Xtrain = X[:10000]
Xvalid = X[10000:20000]
Xtest = X[20000:30000]

ytrain = y[:10000]
yvalid = y[10000:20000]
ytest = y[20000:30000]
```

[53]:
```
A = [.01, .1, 1, 10, 100]
```

[54]:
```
###from piazza

#Unigrams, keep punc, tfidf

#unigrams, discard punc, tfidf

#bigrams, keep punc, tfidf

#bigrams, discard punc, tfidf

#unigrams, keep punc, counts

#unigrams, discard punc, counts

#bigrams, keep punc, counts

#bigrams, discard punc, counts
```

[55]:
```
#Unigrams, keep punc, tfidf
#training data
```

```
unigrams = defaultdict(int)
for d in Xtrain:
#     token = nltk.word_tokenize(d['text'])
#     unigram = list(ngrams(token, 1))
    t = d['text']
    text = " ".join(t.splitlines())
    unigram = text.strip().split()
    for u in unigram:
        unigrams[u] += 1

#1000 most common from training set
mostCommonUni =sorted(unigrams.items(),key=lambda v: v[1],reverse=True)[:1000]
unigram_words = [u[0] for u in mostCommonUni]
unigramId = dict(zip(unigram_words, range(len(unigram_words))))
unigramSet = set(unigram_words)
```

[56]:
```
#docFreq and tf
#training data
docFreq = defaultdict(set)
for d in Xtrain:
#     token = nltk.word_tokenize(d['text'])
#     unigram = list(ngrams(token, 1))
    t = d['text']
    text = " ".join(t.splitlines())
    unigram = text.strip().split()
    for u in unigram:
        docFreq[u].add(d['reviewID'])

#term freq
tf = unigrams
```

[57]:
```
def feature_uni_punc_tfidf(datum):
    feat = [0]*len(unigramSet)
#     t = datum['text']
#     token = nltk.word_tokenize(t)
#     unigram_words = list(ngrams(token, 1))
    t = datum['text']
    text = " ".join(t.splitlines())
    unigram_words = text.strip().split()

    for u in unigram_words:
        if not (u in unigramSet): continue
        tf_idf_word = np.log10(len(Xtrain)/ len(docFreq[u])) * tf[u]
        feat[unigramId[u]] = tf_idf_word

    feat.append(1)
    return feat
```

[58]:
```python
Xtrain_1 = [feature_uni_punc_tfidf(d) for d in Xtrain]
Xvalid_1 = [feature_uni_punc_tfidf(d) for d in Xvalid]
```

[59]:
```python
#unigrams, discard punc, tfidf
def feature_uni_nopunc_tfidf(datum):
    feat = [0]*len(unigramSet)
    t = datum['text']
    t = ''.join([c for c in t.lower() if not c in punctuation])

    text = " ".join(t.splitlines())
    unigram_words = text.strip().split()
#     token = nltk.word_tokenize(t)
#     unigram_words = list(ngrams(token, 1))

    for u in unigram_words:
        if not (u in unigramSet): continue
        tf_idf_word = np.log10(len(Xtrain)/ len(docFreq[u])) * tf[u]
        feat[unigramId[u]] = tf_idf_word

    feat.append(1)
    return feat
```

[60]:
```python
Xtrain_2 = [feature_uni_nopunc_tfidf(d) for d in Xtrain]
Xvalid_2 = [feature_uni_nopunc_tfidf(d) for d in Xvalid]
```

[61]:
```python
#unigrams, keep punc, counts
def feature_uni_punc_wc(datum):
    feat = [0]*len(unigramSet)
    t = datum['text']
#     token = nltk.word_tokenize(t)
#     unigram_words = list(ngrams(token, 1))
    text = " ".join(t.splitlines())
    unigram_words = text.strip().split()

    for u in unigram_words:
        if not (u in unigramSet): continue
        feat[unigramId[u]] += 1

    feat.append(1)
    return feat
```

[62]:
```python
Xtrain_3 = [feature_uni_punc_wc(d) for d in Xtrain]
Xvalid_3 = [feature_uni_punc_wc(d) for d in Xvalid]
```

[63]:
```python
#unigrams, discard punc, counts
def feature_uni_nopunc_wc(datum):
    feat = [0]*len(unigramSet)
```

```
    t = datum['text']
    t = ''.join([c for c in t.lower() if not c in punctuation])

    text = " ".join(t.splitlines())
    unigram_words = text.strip().split()
#     token = nltk.word_tokenize(t)
#     unigram_words = list(ngrams(token, 1))
    for u in unigram_words:
        if not (u in unigramSet): continue
        feat[unigramId[u]] += 1

    feat.append(1)
    return feat
```

[64]:
```
Xtrain_4 = [feature_uni_nopunc_wc(d) for d in Xtrain]
Xvalid_4 = [feature_uni_nopunc_wc(d) for d in Xvalid]
```

[65]:
```
#start of bigram models
bigrams = defaultdict(int)

for d in Xtrain:
#     token = nltk.word_tokenize(d['text'])
#     bigram = list(ngrams(token, 2))
    text = " ".join(d['text'].splitlines())
    bigram = [b for b in zip(text.split(" ")[:-1], text.split(" ")[1:])]
    for b in bigram:
        bigrams[b] += 1

#1000 most common from training set
mostCommonBi =sorted(bigrams.items(),key=lambda v: v[1],reverse=True)[:1000]
bigram_words = [u[0] for u in mostCommonBi]
bigramId = dict(zip(bigram_words, range(len(bigram_words))))
bigramSet = set(bigram_words)
```

[66]:
```
#docFreq and tf
#training data
docFreq = defaultdict(set)
for d in Xtrain:
#     token = nltk.word_tokenize(d['text'])
#     bigram = list(ngrams(token, 2))
    text = " ".join(d['text'].splitlines())
    bigram = [b for b in zip(text.split(" ")[:-1], text.split(" ")[1:])]
    for b in bigram:
        docFreq[b].add(d['reviewID'])

#term freq
tf = bigrams
```

[67]:
```
#bigrams, keep punc, tfidf
def feature_bi_punc_tfidf(datum):
    feat = [0]*len(bigramSet)
    t = datum['text']
#     token = nltk.word_tokenize(t)
#     bigram_words = list(ngrams(token, 2))
    text = " ".join(t.splitlines())
    bigram_words = [b for b in zip(text.split(" ")[:-1], text.split(" ")[1:])]

    for b in bigram_words:
        if not (b in bigramSet): continue
        tf_idf_word = np.log10(len(Xtrain)/ len(docFreq[b])) * tf[b]
        feat[bigramId[b]] = tf_idf_word

    feat.append(1)
    return feat
```

[68]:
```
Xtrain_5 = [feature_bi_punc_tfidf(d) for d in Xtrain]
Xvalid_5 = [feature_bi_punc_tfidf(d) for d in Xvalid]
```

[69]:
```
#bigrams, discard punc, tfidf
def feature_bi_nopunc_tfidf(datum):
    feat = [0]*len(bigramSet)
    t = datum['text']
#     token = nltk.word_tokenize(t)
#     bigram_words = list(ngrams(token, 2))
    t = ''.join([c for c in t.lower() if not c in punctuation])
    text = " ".join(t.splitlines())
    bigram_words = [b for b in zip(text.split(" ")[:-1], text.split(" ")[1:])]

    for b in bigram_words:
        if not (b in bigramSet): continue
        tf_idf_word = np.log10(len(Xtrain)/ len(docFreq[b])) * tf[b]
        feat[bigramId[b]] = tf_idf_word

    feat.append(1)
    return feat
```

[70]:
```
Xtrain_6 = [feature_bi_nopunc_tfidf(d) for d in Xtrain]
Xvalid_6 = [feature_bi_nopunc_tfidf(d) for d in Xvalid]
```

[71]:
```
#bigrams, keep punc, counts
def feature_bi_punc_wc(datum):
    feat = [0]*len(bigramSet)
    t = datum['text']

#     token = nltk.word_tokenize(t)
```

```
#     bigram_words = list(ngrams(token, 2))
    text = " ".join(t.splitlines())
    bigram_words = [b for b in zip(text.split(" ")[:-1], text.split(" ")[1:])]

    for b in bigram_words:
        if not (b in bigramSet): continue
        feat[bigramId[b]] += 1

    feat.append(1)
    return feat
```

[72]:
```
Xtrain_7 = [feature_bi_punc_wc(d) for d in Xtrain]
Xvalid_7 = [feature_bi_punc_wc(d) for d in Xvalid]
```

[73]:
```
#bigrams, discard punc, counts
def feature_bi_nopunc_wc(datum):
    feat = [0]*len(bigramSet)
    t = datum['text']

#     token = nltk.word_tokenize(t)
#     bigram_words = list(ngrams(token, 2))

    t = ''.join([c for c in t.lower() if not c in punctuation])
    text = " ".join(t.splitlines())
    bigram_words = [b for b in zip(text.split(" ")[:-1], text.split(" ")[1:])]

    for b in bigram_words:
        if not (b in bigramSet): continue
        feat[bigramId[b]] += 1

    feat.append(1)
    return feat
```

[74]:
```
Xtrain_8 = [feature_bi_nopunc_wc(d) for d in Xtrain]
Xvalid_8 = [feature_bi_nopunc_wc(d) for d in Xvalid]
```

[75]:
```
# pipeline
to_fit = [Xtrain_1, Xtrain_2, Xtrain_3, Xtrain_4, Xtrain_5, Xtrain_6, Xtrain_7,␣
 ↪Xtrain_8]
to_pred = [Xvalid_1, Xvalid_2, Xvalid_3, Xvalid_4, Xvalid_5, Xvalid_6,␣
 ↪Xvalid_7, Xvalid_8]
model_performances = []
for i in range(len(to_fit)):
    for a in A:
        clf = linear_model.Ridge(a, fit_intercept=False)
        clf.fit(to_fit[i], ytrain)
        theta = clf.coef_
```

```
        predictions = clf.predict(to_pred[i])
        model_performances.append(MSE(yvalid, predictions))
```

[ ]:

```
[76]: model_names = ["unigrams, keep punc, tfidf",
"unigrams, discard punc, tfidf",
"unigrams, keep punc, counts",
"unigrams, discard punc, counts",
"bigrams, keep punc, tfidf",
"bigrams, discard punc, tfidf",
"bigrams, keep punc, counts",
"bigrams, discard punc, counts"]

index_names = []
for model in model_names:
    for a in A:
        index_names.append((model,a))
```

```
[77]: index = pd.MultiIndex.from_tuples(index_names, names=['model','regularization␣
 ↪param'])
```

```
[78]: df = pd.DataFrame(data = model_performances, index = index, columns = ['mse'])
df
```

[78]:

model	regularization param	mse
unigrams, keep punc, tfidf	0.01	5.215339
	0.10	5.215344
	1.00	5.215392
	10.00	5.215917
	100.00	5.224640
unigrams, discard punc, tfidf	0.01	4.982110
	0.10	4.982116
	1.00	4.982173
	10.00	4.982780
	100.00	4.992304
unigrams, keep punc, counts	0.01	6.108231
	0.10	6.104438
	1.00	6.070248
	10.00	5.854864
	100.00	5.329262
unigrams, discard punc, counts	0.01	5.435183
	0.10	5.434333
	1.00	5.426033
	10.00	5.358437
	100.00	5.141136

```
bigrams, keep punc, tfidf       0.01              5.565673
                                0.10              5.565676
                                1.00              5.565711
                                10.00             5.566079
                                100.00            5.571945
bigrams, discard punc, tfidf    0.01              5.376637
                                0.10              5.376641
                                1.00              5.376683
                                10.00             5.377123
                                100.00            5.383752
bigrams, keep punc, counts      0.01              5.806254
                                0.10              5.803601
                                1.00              5.777180
                                10.00             5.581628
                                100.00            5.230566
bigrams, discard punc, counts   0.01              5.759028
                                0.10              5.752778
                                1.00              5.713767
                                10.00             5.497860
                                100.00            5.147350
```

[79]:
```
df.sort_values(by = 'mse').iloc[0]
```

[79]:
```
mse    4.98211
Name: (unigrams, discard punc, tfidf, 0.01), dtype: float64
```

[ ]:

[80]:
```
Xtrain_ = [feature_uni_nopunc_wc(d) for d in Xtrain]
Xtest_ = [feature_uni_nopunc_wc(d) for d in Xtest]
```

[81]:
```
clf = linear_model.Ridge(100, fit_intercept=False)
clf.fit(Xtrain_, ytrain)
theta = clf.coef_
predictions = clf.predict(Xtest_)
print("mse: ", MSE(ytest, predictions))
```

```
mse:  4.981581181129463
```

[ ]:

[ ]: