# Homework 4

November 30, 2020

[1]:
```
### Libraries
import numpy as np
import ast
import urllib
import matplotlib.pyplot as plt
import random
from collections import defaultdict
import scipy.optimize
from sklearn import linear_model
import math
from sklearn.ensemble import AdaBoostClassifier
import gzip
import string
```

[2]:
```
def parseDataFromFile(fname):
    """ Read and convert the input to a list of dicts"""
    for l in open(fname, encoding="utf-8"):
        yield ast.literal_eval(l)

def readGz(path):
    for l in gzip.open(path, 'rt', encoding="utf-8"):
        yield eval(l)
```

[3]:
```
fname = "C:\\Users\\ramasarma\\Documents\\UCSD\\Fall 2020\\CSE␣
 ↪258\\Assignment1\\assignment1\\train_Category.json"
data = list(parseDataFromFile(fname))
print(len(data))
```

```
175000
```

[4]:
```
print(data[0])
# Considering only the first 10000 samples
dataset = data[:10000]
```

```
{'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]:
```
corpus = []
punctuation = set(string.punctuation)
for d in dataset:
    r = ''.join([c for c in d['text'].lower() if c not in punctuation])
    corpus.append(r)
#print(corpus[:2])
```

Stats related to the dataset

[6]:
```
# Reading through the text after removing punctuation
wordCount = defaultdict(int)
for document in corpus:
    for word in document.split():
        wordCount[word] += 1
print("Number of unique unigrams are {}".format(len(wordCount.keys())))
```

```
Number of unique unigrams are 29692
```

### 0.0.1 Question 1 - Number of unique bigrams

[7]:
```
bigram_map = defaultdict(int)
for document in corpus:
    words = document.split()
    for i in range(len(words) - 1):
        first_word = words[i]
        second_word = words[i+1]
        key = first_word + ' ' + second_word
        bigram_map[key] += 1

wordCount = [(bigram_map[key], key) for key in bigram_map]
wordCount.sort()
wordCount.reverse()
```

```
print("Number of unique bigrams amongst the reviews are {}".
 ↪format(len(bigram_map.keys())))

print(" The 5 top bigrams are mentioned as follows - ")
print(wordCount[:5])
```

```
Number of unique bigrams amongst the reviews are 256618
 The 5 top bigrams are mentioned as follows -
[(4441, 'this game'), (4249, 'the game'), (3359, 'of the'), (2020, 'if you'),
(2017, 'in the')]
```

### 0.0.2 Question 2 - Here, we are using the 1000 Most common Bigrams as features.

[8]:
```
topNGrams = wordCount[:1000]
bigrams = [val[1] for val in topNGrams]
bigramId = dict(zip(bigrams, range(len(bigrams))))
bigramSet = set(bigrams)

def feature(datum):
    feat = ([0]*len(bigrams))
    r = ''.join([c for c in datum['text'].lower() if not c in punctuation])
    words = r.split()
    for i in range(len(words) - 1):
        bigram = words[i] + ' ' + words[i + 1]
        if bigram in bigrams:
            feat[bigramId[bigram]] += 1
    feat.append(1)
    return feat

X = [feature(d) for d in dataset]
y = [math.log2(d['hours'] + 1) for d in dataset]

clf = linear_model.Ridge(1.0, fit_intercept=False) # MSE + 1.0 l2
clf.fit(X, y)
theta = clf.coef_
predictions = clf.predict(X)
```

[9]:
```
def MSE(actual, predictions):
    differences = [(x-y)**2 for (x, y) in zip(actual, predictions)]
    return sum(differences)/len(differences)

# Computing the mean squared error
mse_train = MSE(y, predictions)
print("Training set MSE = {}".format(mse_train))
```

```
Training set MSE = 4.393787247200031
```

### 0.0.3 Question 3 - Here, we are using the 1000 most common unigrams and bigrams as features

[10]:
```python
def feature(datum, K):
    """ args : datum - data sample
        K : Top K Grams to be filtered """
    r = ''.join([c for c in datum['text'].lower() if c not in punctuation])
    words = r.split()
    if(len(words) == 0):
        feat = [0] * K
        feat.append(1)
        return feat
    bigram_map = defaultdict(int)
    unigram_map = defaultdict(int)

    # Bigram map is used to store bigram frequencies
    # Unigram map is used to store unigram frequencies
    for i in range(len(words) - 1):
        bigram_key = words[i] + ' ' + words[i+1]
        bigram_map[bigram_key] += 1
        unigram_map[words[i]] += 1
    #last_word = words[-1]
    unigram_map[words[-1]] += 1

    # Compute the same for unigram and bigrams
    wordCount_unigram = [(unigram_map[key], key) for key in unigram_map]
    wordCount_bigram = [(bigram_map[key], key) for key in bigram_map]

    # Compute the wordCount for unigram and bigram
    wordCount_unigram.sort()
    wordCount_bigram.sort()
    wordCount_unigram.reverse()
    wordCount_bigram.reverse()

    # List of keys for the unigram and bigram
    unigramKeys = [x[1] for x in wordCount_unigram]
    bigramKeys = [x[1] for x in wordCount_bigram]
    unigramId = dict(zip(unigramKeys, range(len(unigramKeys))))
    bigramId = dict(zip(bigramKeys, range(len(bigramKeys))))

    # Feature of a combination of unigram and bigrams
    topNGrams = []
```

```
    # Problem similar to merging two sorted lists with a condition on the max␣
↪number of lists
    i, i_u, i_b = 0, 0, 0
    while (i < K and (i_u < len(wordCount_unigram) and i_b <␣
↪len(wordCount_bigram))):
        if(wordCount_unigram[i_u][0] > wordCount_bigram[i_b][0]):
            topNGrams.append(wordCount_unigram[i_u][1])
            i_u += 1
        else:
            topNGrams.append(wordCount_bigram[i_b][1])
            i_b += 1
        i += 1

    while (i < K and (i_u < len(wordCount_unigram))):
        topNGrams.append(wordCount_unigram[i_u][1])
        i_u += 1
        i += 1

    while (i < K and (i_b < len(wordCount_bigram))):
        topNGrams.append(wordCount_bigram[i_b][1])
        i_b += 1
        i += 1

    feat = ([0] * K)
    nGrams = [val for val in topNGrams]
    nGramId = dict(zip(nGrams, range(len(nGrams))))
    nGramSet = set(nGramId)
    for nGram in topNGrams:
        # Logic to check whether it's a bigram or unigram?
        if(len(nGram.split()) > 1):
            feat[nGramId[nGram]] += 1
        else:
            assert(len(nGram.split()) == 1)
            feat[nGramId[nGram]] += 1
    feat.append(1)
    return feat

# Compute the feature matrix "X" for the given dataset
X = [feature(d, 1000) for d in dataset]
# Process the dataset for the matrix "Y"
y = [math.log2(d['hours'] + 1) for d in dataset]
# Compute the Linear Regression model here
clf = linear_model.Ridge(1.0, fit_intercept=True) # MSE + 1.0 l2
clf.fit(X, y)
theta = clf.coef_
predictions = clf.predict(X)
```

```
mse_train = MSE(y, predictions)
print("Training set MSE = {}".format(mse_train))
```

```
Training set MSE = 4.9110892467357665
```

### 0.0.4 Question 4 - Inverse Document Frequency

[29]:
```
def computeIDF(corpus, searches):
    idfs = []
    numDocs = len(corpus)
    for searchWord in searches:
        count = 0
        for document in corpus:
            words = searchWord.split()
            if(searchWord in words):
                count += 1
        idfs.append(math.log10(numDocs/(1 + count)))
    return idfs

def computeTF(corpus, searches):
    (rows, cols) = (len(corpus), len(searches))
    tfs = [[0 for i in range(cols)] for j in range(rows)]
    for searchWord in searches:
        for document in corpus:
            words = document.split()
            tf, numWords = 0, len(words)
            word_idx = searches.index(searchWord)
            doc_idx = corpus.index(document)
            if(searchWord in words):
                tfs[doc_idx][word_idx] += 1/numWords

    return tfs

# Go through the entire list of searches to compute the IDF
searches = ["destiny", "annoying", "likeable", "chapter", "interesting"]
idf = computeIDF(corpus, searches)
print("First five IDFs are {}".format(idf[:5]))
tf = computeTF(corpus, searches)
print("First five TFs are {}".format(tf[:5]))
```

```
First five IDFs are [-4.342727686267685e-05, -4.342727686267685e-05,
-4.342727686267685e-05, -4.342727686267685e-05, -4.342727686267685e-05]
First five TFs are [[0.004694835680751174, 0.004694835680751174,
0.004694835680751174, 0.004694835680751174, 0.004694835680751174], [0, 0, 0, 0,
0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0.007042253521126761], [0, 0, 0, 0, 0]]
```

### 0.0.5 TF IDF scores for the reviewID r75487422

[30]:
```
# review provides the entity in corpus
review = corpus[0]
tf, tf_idf = [0]*len(searches), [0] * len(searches)
for i in range(len(searches)):
    words = review.split()
    numWords = len(words)
    # Number of words = length of words in a document
    count = 0
    if (searches[i] in words):
        count += 1
    tf[i] = count / numWords
    val = 1 if (tf[i] > 0) else 0
    tf_idf[i] = tf[i] * idf[i]
    print("Word = {}, TF score = {}, IDF score = {}, TFIDF score = {}".
 ↪format(searches[i], tf[i], idf[i], tf_idf[i]))
```

```
Word = destiny, TF score = 0.004694835680751174, IDF score =
-4.342727686267685e-05, TFIDF score = -2.0388392893275517e-07
Word = annoying, TF score = 0.004694835680751174, IDF score =
-4.342727686267685e-05, TFIDF score = -2.0388392893275517e-07
Word = likeable, TF score = 0.004694835680751174, IDF score =
-4.342727686267685e-05, TFIDF score = -2.0388392893275517e-07
Word = chapter, TF score = 0.004694835680751174, IDF score =
-4.342727686267685e-05, TFIDF score = -2.0388392893275517e-07
Word = interesting, TF score = 0.004694835680751174, IDF score =
-4.342727686267685e-05, TFIDF score = -2.0388392893275517e-07
```

### 0.0.6 Adapt the unigram model to use the TF-IDF scores

### 0.0.7 Question 5

[31]:
```
def computeTopNUnigrams(corpus, N):
    topNUnigrams = []
    wordCount = defaultdict(int)
    for document in corpus:
        for word in document.split():
            wordCount[word] += 1

    mostCommon = [(wordCount[x], x) for x in wordCount]
    # Sort and reverse the code
    mostCommon.sort()
    mostCommon.reverse()
    for i in range(N):
        topNUnigrams.append(mostCommon[i][1])
```

```
    return topNUnigrams

def computeTFIDF(tf, idf):
    len_tf = len(tf)
    tf_idfs = []
    for i in range(len_tf):
        tf_idf = []
        for j in range(len(tf[i])):
            tf_idf.append(float(tf[i][j] * idf[j]))
        tf_idfs.append(tf_idf)
    return tf_idfs
```

[32]:
```
def feature(corpus, N):
    """ feature() is used to compute the features """
    topNUnigrams = computeTopNUnigrams(corpus, N)
    #print(top1000Unigrams[:5])
    idf = computeIDF(corpus, topNUnigrams)
    tf = computeTF(corpus, topNUnigrams)
    tf_idf = computeTFIDF(tf, idf)
    for i in range(len(tf_idf)):
        # Offset addition
        tf_idf[i].append(1)
    return tf_idf

X_train = feature(corpus, 1000)
Y_train = [math.log2(d['hours'] + 1) for d in dataset]
print("Computed features for the training set")
print(len(X_train), len(X_train[0]))
clf = linear_model.Ridge(1.0, fit_intercept=False) # MSE + 1.0 l2
clf.fit(X_train, Y_train)
theta = clf.coef_
predictions = clf.predict(X_train)
mse_train = MSE(Y_train, predictions)
print("Training set MSE = {}".format(mse_train))
```

```
Computed features for the training set
10000 1001
Training set MSE = 5.242479008006355
```

### 0.0.8 Question 6

[33]:
```
def norm(A):
    val = 0
    for ele in A:
        val += ele**2
    return val
```

```
# Here, we have the cosine similarity being defined
def cosine_similarity(A, B):
    numerator = np.dot(A, B)
    denominator = math.sqrt(norm(A) * norm(B))
    return numerator/denominator

# For every datapoint in the given dataset
r_idx = -1
N = 1000 # Number of unigrams to process
X = feature(corpus, 1000)
for d in dataset:
    if(d['reviewID'] == "r19740876"):
        r_idx = dataset.index(d)
        break
print("The index of review r75487422 is {}".format(r_idx))
max_sim = -1
mostSimilarReviewId = -1
max_idx = -1
for i in range(len(dataset)):
    if(i != r_idx):
        cos_sim = cosine_similarity(X[r_idx], X[i])
        if(cos_sim > max_sim):
            mostSimilarReviewId = dataset[i]['reviewID']
            max_sim = cos_sim
            max_idx = i

print("Most similar reviewID to r75487422 is {} and is present at index {}".
 ↪format(mostSimilarReviewId, max_idx))
```

```
The index of review r75487422 is 8328
Most similar reviewID to r75487422 is r26288269 and is present at index 420
```

[34]:
```
print(dataset[8328])
```

```
{'userID': 'u46449878', 'genre': 'RPG', 'early_access': False, 'reviewID':
'r19740876', 'hours': 0.6, 'text': 'NOTE: This game is Free To Play, meaning
this review and many others do not count. I cannot stress enough that you play
it for yourself with an open mind to get an undilluted opinion and ignore this
review and any others.\nSo… I know what Depression is. I\'ve been diagnosed
with it for the last 15 years alongside High Anxiety for all that time.\nYou
want to know what Depression is like? Let me tell you instead of you playing
this game.\nImagine sadness. Imagine extreme self-disappointment. Imagine self-
loathing to as high of a degree as it\'ll go. It\'s not always the same, but to
my knowledge… Mix them into one emotion and you get Depression.\nIt is
literally the worst emotion on the planet. Mostly all suicides are caused by
feelings of Depression. I have no study, no way to prove that, other than just
plain logic and understanding.\nThis game does not convey the feelings of
```

Depression.\nIt is sad, yes, but Sadness is not Depression. On it\'s own that is. You need a slew of other different, but similar emotions but also the faint hope that it can get better and mental habits of being pessimistic about your life, you and everything around you.\nI\'ve played and read all lines of both endings and choice options. there\'s multiple choices in most cases, but in most cases the options are locked because the dev most likely didn\'t program that far in a text-based game. I could also say because of [arbitrary/contrived reason].\nLet me be perfectly clear. This game is not depressing, it\'s not saddening. To me… It\'s insulting how hyped this game is on the store page.\nIt\'s got the pacing of a pop-up book, the writing of an overly-emotional stereotype writing about how awful life is and for a game that boasts "multiple endings" I would have imagined that the developer would\'ve coded in a suicide ending, since I literally did every awful option and nothing really different happened.\nThe game is as engaging as listenning to someone whine about how awful their life is. I know, it\'s supposed to convey the feelings of being Depressed, but the thing is… Unless you\'re a wordsmith (which The Quinnspiracy isn\'t) then it will never work in text form.\nThe game would have worked better as a more interactive game that isn\'t text-based or an auto-biography of part of the dev\'s life.\nWhile I was playing-no… While I was reading this e-book it felt like I was reading what amounted to angsty, emo fanfiction and not even good fanfiction.\nSome options of the game aren\'t labeled too great, sometimes the result being the opposite of what you thought the option would do (so… control problems in a text-based game?) which basically boils down to those options being improperly labeled.\nThe soundtrack is… well it\'s actually just a track, one song looping over and over and not even with a clean loop either, at certain times you\'ll hear crackling or popping in the audio as it loops.\nAll in all, this game utterly fails at what it tries to do. Not only that, but it makes Depression laughable, comparable to stupid pre-teens who thinks depression is a cool thing, who we all know are a complete joke, and that\'s the literal opposite of Depression.\nI\'ve seen better pacing in books you can read in any library anywhere, the game wasn\'t at all uncomfortable for me, it wasn\'t like an unfiltered mental diary, more like a "pity me, I\'m sad right?" type of story written by some spoiled kid who got grounded for the first time.\nJust like any poorly-made text-based game, you can look at the first sentence/paragraph, the last one aswell and essentially miss nothing story-wise, ignoring everything inbetween, only with this one, you can look at jsut the last sentence or two and know all you need for the next decision.\nThe Choice-based content is actually true to a degree, but the multiple endings are comparable to Mass Effect 3\'s "What color will your ship\'s explosion be" type of ending, each one summing up all you\'ve done, with ONE sentence of difference ignoring all else (summing up your run doesn\'t really count) which doesn\'t matter because this game is extremely far from being long, even if you read it all. I stopped reading all but the last sentence and first one halfway through.\nPros:\n- Content based on Choice is true to a degree.\n- The "art style" of the game works decently.\nCons:\n- Completely makes a joke of Depression.\n- Fails to give an understanding view of depression to the player.\n- Text drones on about the player\'s awful life and at times feels like empty complaining and whining.\n- There isn\'t really music, just

ambience with awful loop crackles. It loops 3 or 4 times in a 15 minute period.\n- You can ignore most of the text without loss of story/choice context.\n- Some choices are literally always locked no matter what you do.\n- There\'s a choice or two on the first screen that is locked.\n- Due to the third point, the game is dull and couldn\'t be further from being engaging even if it\'s in 1st person.\n- The wording in some pages are poorly done, or done wierdly to confuse the reader.\n- The Endings are technically the same save for one or two lines of text that don\'t change anything.\nThis game gets a 1/10. It makes a joke out of Depression by coming off as whiny and complaining about how awful life is, saying how awful everything is rather than using words to imply the feelings so the reader can dig and feel them that way.\nIf that wasn\'t a thing and it did what it\'d do, it would be a 5/10 simply due to everything else just being passable. It\'s not well-made GameSpy, it\'s just passable. I\'ve played many text-based games that are more well executed than this, some of them being Zork, Trials in Tainted Space and Unification Wars. Trials is in beta.\nTo improve it I would suggest a complete redo of the whole game. It needs or be worded better, don\'t tell us, make us imagine it so we can be THERE, not watching ourselves like a ghost would. Be less preachy or Emo-y (if that\'s a word).\nI would also suggest the ambience loop be cleaned up and have the crackles removed.\nMake the endings ALOT more dramatically different. Not a single sentence of difference, I\'m talking the difference between a life that will continue to be happy, or literally suicide (say if you choose all the good options and then all the bad ones in another playthrough).\nIt doesn\'t even need to be longer.\nWill I give it Forbidden status? Well… based on how it\'s been released a full year, I\'m inclined to say yes, but I will hold off on that as per the usual for my statuses.\nI live with depression every day. I know how it feels. This game? Not one single moment did I EVER empathize with the character. It fails at it\'s intended purpose.', 'genreID': 2, 'date': '2015-08-22'}

### 0.0.9 Question 7

[35]:
```
def computeTopNBigrams(corpus, N):
    topNBigrams = []
    wordCount = defaultdict(int)
    for document in corpus:
        words = document.split()
        for i in range(len(words)-1):
            key = word[i] + ' ' + word[i+1]
            wordCount[key] += 1

    mostCommon = [(wordCount[x], x) for x in wordCount]
    # Sort and reverse the code
    mostCommon.sort()
    mostCommon.reverse()
```

```
    for i in range(N):
        topNBigrams.append(mostCommon[i][1])
    return topNBigrams
```

[36]:
```
print(set(string.punctuation))
```

```
{'\\', ';', '$', '`', '/', '!', '%', ']', ':', '|', '-', '+', '"', '=', '{',
'^', '~', '*', "'", ',', '_', ')', '#', '>', '(', '&', '@', '}', '[', '.', '?',
'<'}
```

### 0.0.10 Comparison between unigram and bigram models for different variants of regularization parameters

Unigram + Removed Punctuation + TD IDF Scores = 0, 0, 0

Unigram + Removed Punctuation + Word Counts = 0, 0, 1

Unigram + Preserve Punctuation + TD IDF Scores = 0, 1, 0

Unigram + Preserve Punctuation + Word Counts = 0, 1, 1

Bigram + Removed Punctuation + TD IDF Scores = 1, 0, 0

Bigram + Removed Punctuation + Word Counts = 1, 0, 1

Bigram + Preserve Punctuation + TD IDF Scores = 1, 1, 0

Bigram + Preserve Punctuation + Word Counts = 1, 1, 1

[45]:
```
import re

def computeCount(wordCorpus, gram_type, N):
    """ args - wordCorpus - 2D list (each row - list of words in one review)
                gram_type - Unigram (or) Bigram
                N - Number of grams to search for """
    features = []
    for words in wordCorpus:
        length = len(words) if (gram_type == "unigram") else len(words) - 1
        wordCount = defaultdict(float)
        for i in range(length):
            if(gram_type == "unigram"):
                wordCount[words[i]] += 1
            else:
                wordCount[words[i] + ' ' + words[i+1]] += 1
        mostCommon = [(wordCount[x], x) for x in wordCount]
        mostCommon.sort()
        mostCommon.reverse()
        topNGrams = [] # store for every document in the corpus
        for i in range(N):
            if(i < len(mostCommon)):
```

```
                topNGrams.append(mostCommon[i][1])
        #print("Length of topNGrams = {}".format(len(topNGrams)))
        # Compute the unigramId and accordingly prepare the feature vector
        ngramId = dict(zip(topNGrams, range(len(topNGrams))))
        feature = [0] * N
        for word in words:
            if(word in topNGrams):
                feature[ngramId[word]] += 1
        feature.append(1) # For the constant \theta_0
        features.append(feature)

    return features

def computeTFIDF(wordCorpus, gram_type, N):
    """ args - wordCorpus - 2D list (each row - list of words in one review)
                gram_type - Unigram (or) Bigram
                N - Number of grams to search for """
    features, tfs, idfs = [], [], []
    numDocuments = defaultdict(int)
    for words in wordCorpus:
        length = len(words) if (gram_type == "unigram") else len(words) - 1
        for i in range(length):
            if(gram_type == "unigram"):
                numDocuments[words[i]] += 1
            else:
                numDocuments[words[i] + ' ' + words[i+1]] += 1

    for words in wordCorpus:
        # Logic to compute most frequent unigrams or bigrams
        length = len(words) if (gram_type == "unigram") else len(words) - 1
        wordCount = defaultdict(float)
        for i in range(length):
            if(gram_type == "unigram"):
                wordCount[words[i]] += 1
                numDocuments[words[i]] += 1
            else:
                wordCount[words[i] + ' ' + words[i+1]] += 1
                numDocuments[words[i] + ' ' + words[i+1]] += 1

        mostCommon = [(wordCount[x], x) for x in wordCount]
        mostCommon.sort()
        mostCommon.reverse()
        topNGrams = [] # store for every document in the corpus
        for i in range(N):
            if(i < len(mostCommon)):
                topNGrams.append(mostCommon[i][1])
```

```
        ngramId = dict(zip(topNGrams, range(len(topNGrams))))
        # Compute TF
        tf = [0] * N
        numWords = len(words)
        for word in words:
            if(word in ngramId):
                tf[ngramId[word]] += 1/numWords
        # Compute IDF
        # Number of documents that have the term present in it
        idf = [0] * N
        for i in range(N):
            if(i < len(topNGrams)):
                idf[i] = math.log10(len(wordCorpus)/(numDocuments[topNGrams[i]]␣
 ↪+ 1))
        feature = [tf[i] * idf[i] for i in range(N)]
        # For constant \theta_0
        feature.append(1)
        features.append(feature)

    return features

# Customized implementation of split function
def customSplit(characters, delimiters):
    result = []
    word = ""
    for char in characters:
        if(char in delimiters):
            if(len(word) > 0):
                result.append(word)
            if(char != ' '):
                word = str(char)
                result.append(word)
            word = ""
        else:
            word += char
    return result

def computeFeatures(dataset, gram_type, punct_type, score_type, N):
    """ dataset - Input data for which the feature needs to be extracted
        gram_type - Unigram or Bigram
        punct_type - Preserve or remove punctuations
        score_type - TF-IDF or wordCount """
    features, wordCorpus, corpus = [], [], []
    for d in dataset:
        if(punct_type == "remove"):
            review = ''.join([c for c in d['text'].lower() if c not in␣
 ↪punctuation])
```

```
        elif(punct_type == "preserve"):
            review = ''.join([c for c in d['text'].lower()])
        corpus.append(review)
    punctList = [ch for ch in string.punctuation]
    # For every document in the list of all documents
    for document in corpus:
        delimiters = [' ']
        if(punct_type == "preserve"):
            for delimit in string.punctuation:
                delimiters.append(delimit)
            words = customSplit(document, delimiters)
        else:
            assert(punct_type == "remove")
            words = customSplit(document, delimiters)
        wordCorpus.append(words)

    if(score_type == "wordCount"):
        features = computeCount(wordCorpus, gram_type, N)
    else:
        assert(score_type == "tf_idf")
        features = computeTFIDF(wordCorpus, gram_type, N)

    return features
```

[48]:
```
def validation_pipeline(data):
    random.seed(0)
    random.shuffle(data)
    # size = 10000
    training_set = data[:10000]
    validation_set = data[10000:20000]
    test_set = data[20000:30000]
    # Regularization parameters
    lambdas = [10 ** i for i in np.arange(-2,3,dtype=float)]
    # Different options for the loop
    gramTypes = ["unigram", "bigram"]
    punctuationTypes = ["preserve", "remove"]
    countTypes = ["tf_idf", "wordCount"]
    # Place to store MSE values for all the models
    mseVals = defaultdict(float)
    bestModels = defaultdict(float)
    # For each gramType in the list of gramTypes
    for gramType in gramTypes:
        for puncType in punctuationTypes:
            for countType in countTypes:
                # Training set is as follows
                X_train = computeFeatures(training_set, gramType, puncType,␣
↪countType, 1000)
```

```
                Y_train = [math.log2(d['hours'] + 1) for d in training_set]
                # Validation set is as follows
                X_val = computeFeatures(validation_set, gramType, puncType,␣
↪countType, 1000)
                Y_val = [math.log2(d['hours'] + 1) for d in validation_set]
                # Test set is as follows
                X_test = computeFeatures(test_set, gramType, puncType,␣
↪countType, 1000)
                Y_test = [math.log2(d['hours'] + 1) for d in test_set]
                # Ensure that the length of the training set "equals" 1001
                assert(len(X_train[0]) == 1001)
                minMSE, minlamb = 10000, -1
                for lamb in lambdas:
                    clf = linear_model.Ridge(lamb, fit_intercept=True)
                    clf.fit(X_train, Y_train)
                    #theta = clf.coef_
                    Y_val_pred = clf.predict(X_val)
                    mse_val = MSE(Y_val, Y_val_pred)
                    key = str(gramType) + '|' + str(puncType) + '|' +␣
↪str(countType) + '|' + str(lamb)
                    print("Validation MSE with key = {} is {}".format(key,␣
↪mse_val))
                    mseVals[key] = mse_val
                    if(mse_val < minMSE):
                        minMSE = mse_val
                        minlamb = lamb
                k = str(gramType) + '|' + str(puncType) + '|' + str(countType)␣
↪+ '|' + str(minlamb)
                bestModels[k] = minMSE
                clf = linear_model.Ridge(minlamb, fit_intercept=True)
                clf.fit(X_train, Y_train)
                Y_test_pred = clf.predict(X_test)
                mse = MSE(Y_test, Y_test_pred)
                print("Test MSE at {}, {}, {} and lambda = {} is {}".\
                      format(gramType, puncType, countType, minlamb, mse))

validation_pipeline(data)
```

```
Validation MSE with key = unigram|preserve|tf_idf|0.01 is 5.357231999710872
Validation MSE with key = unigram|preserve|tf_idf|0.1 is 5.332903487594727
Validation MSE with key = unigram|preserve|tf_idf|1.0 is 5.321886691089576
Validation MSE with key = unigram|preserve|tf_idf|10.0 is 5.320347749905078
Validation MSE with key = unigram|preserve|tf_idf|100.0 is 5.317046585844409
Test MSE at unigram, preserve, tf_idf and lambda = 100.0 is 5.198895607621101
Validation MSE with key = unigram|preserve|wordCount|0.01 is 5.514462641035678
Validation MSE with key = unigram|preserve|wordCount|0.1 is 5.489822087879586
Validation MSE with key = unigram|preserve|wordCount|1.0 is 5.417046530978679
```

```
Validation MSE with key = unigram|preserve|wordCount|10.0 is 5.3487883610962035
Validation MSE with key = unigram|preserve|wordCount|100.0 is 5.309129178745707
Test MSE at unigram, preserve, wordCount and lambda = 100.0 is 5.206755660610267
Validation MSE with key = unigram|remove|tf_idf|0.01 is 5.333881336668315
Validation MSE with key = unigram|remove|tf_idf|0.1 is 5.315217056334459
Validation MSE with key = unigram|remove|tf_idf|1.0 is 5.31048010480832
Validation MSE with key = unigram|remove|tf_idf|10.0 is 5.315102638667783
Validation MSE with key = unigram|remove|tf_idf|100.0 is 5.317233944627375
Test MSE at unigram, remove, tf_idf and lambda = 1.0 is 5.210768486373123
Validation MSE with key = unigram|remove|wordCount|0.01 is 5.454944679317683
Validation MSE with key = unigram|remove|wordCount|0.1 is 5.435021036146686
Validation MSE with key = unigram|remove|wordCount|1.0 is 5.381874790634799
Validation MSE with key = unigram|remove|wordCount|10.0 is 5.3386169774959376
Validation MSE with key = unigram|remove|wordCount|100.0 is 5.308760465041253
Test MSE at unigram, remove, wordCount and lambda = 100.0 is 5.2810556682052665
Validation MSE with key = bigram|preserve|tf_idf|0.01 is 5.316094612279973
Validation MSE with key = bigram|preserve|tf_idf|0.1 is 5.316094612279973
Validation MSE with key = bigram|preserve|tf_idf|1.0 is 5.316094612279973
Validation MSE with key = bigram|preserve|tf_idf|10.0 is 5.316094612279973
Validation MSE with key = bigram|preserve|tf_idf|100.0 is 5.316094612279973
Test MSE at bigram, preserve, tf_idf and lambda = 0.01 is 5.199761277161803
Validation MSE with key = bigram|preserve|wordCount|0.01 is 5.316094612279973
Validation MSE with key = bigram|preserve|wordCount|0.1 is 5.316094612279973
Validation MSE with key = bigram|preserve|wordCount|1.0 is 5.316094612279973
Validation MSE with key = bigram|preserve|wordCount|10.0 is 5.316094612279973
Validation MSE with key = bigram|preserve|wordCount|100.0 is 5.316094612279973
Test MSE at bigram, preserve, wordCount and lambda = 0.01 is 5.199761277161803
Validation MSE with key = bigram|remove|tf_idf|0.01 is 5.316094612279973
Validation MSE with key = bigram|remove|tf_idf|0.1 is 5.316094612279973
Validation MSE with key = bigram|remove|tf_idf|1.0 is 5.316094612279973
Validation MSE with key = bigram|remove|tf_idf|10.0 is 5.316094612279973
Validation MSE with key = bigram|remove|tf_idf|100.0 is 5.316094612279973
Test MSE at bigram, remove, tf_idf and lambda = 0.01 is 5.199761277161803
Validation MSE with key = bigram|remove|wordCount|0.01 is 5.316094612279973
Validation MSE with key = bigram|remove|wordCount|0.1 is 5.316094612279973
Validation MSE with key = bigram|remove|wordCount|1.0 is 5.316094612279973
Validation MSE with key = bigram|remove|wordCount|10.0 is 5.316094612279973
Validation MSE with key = bigram|remove|wordCount|100.0 is 5.316094612279973
Test MSE at bigram, remove, wordCount and lambda = 0.01 is 5.199761277161803
```

Unigram model with preserve punctuations and TF IDF score performs better than all models on the test set. The above tables have the values for validation set MSE and test set MSE for a total of 48 configurations (40 for validation set and 8 for test set)

[ ]: