Brian Craft

Sriram Yarlagadda

CSC 575 Project: Testing Classification Algorithms on Yelp Data

Initial Proposal

We plan to use the Reviews dataset to create a text categorization model that will be able to predict the rating of a given review. Each rating is considered a different category. The dataset will be divided into testing and training data. We plan to build an inverted index or a document-term matrix from each of the reviews and use this to build our prediction model. We will explore different term weighting measures, similarity measures, and classification models and pick the combination that maximizes the prediction accuracy on the testing data. Since personal computers have limited memory and processing power, we may employ techniques such as random or stratified sampling to reduce the data we work with. Although having a high accuracy is important, that is not the sole purpose of this project, we are more focused on evaluating different text classification models and picking the best one.

Overview of data

In 2015, Yelp released a dataset containing 1.6M reviews for various business as part of the Yelp Dataset Challenge. The goal was to allow data oriented people everywhere a chance to find something innovative or insightful using the data and potentially earn some money in the process. Our analysis focused on a portion of the data, specifically the text within each review and the ratings for each review. This left us with a JSON file that is 1.4 GB in size. Below is a link to the dataset and more information on the 2015 Yelp Dataset Challenge.

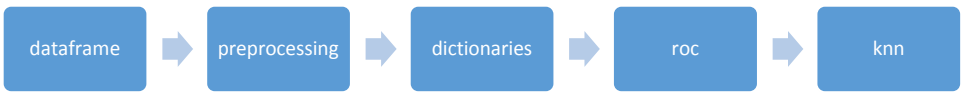
http://engineeringblog.yelp.com/2015/02/yelp-dataset-challenge-is-doubling-up.html

Research Question

Can we use the words within a Yelp review to predict the rating using classification algorithms?

Overview of System

The system uses a series of python functions to first read in the raw JSON data, preprocess the data and create an inverted index used for similarity calculations, after which a rating prediction is made. Each box in the below flow chart represents a different function to be used to run the entire system to completion. An in depth look at each function is seen further along in the report or the appendix.



The system makes use of combinations of different weighting, similarity and classification methodologies, specifically the ones below.

- Weighting Measures
    - TF
    - TF x IDF
    - Binary
- Similarity Measures
    - Dot Product
    - Dice's Coefficient
- Classification Algorithms
    - Rocchio's Method
    - K-Nearest Neighbors (k = {5, 9, 31})

Functions

**all code can be seen in the appendix*

dataframe( ):

The first function is called dataframe, which takes a JSON file containing a document id, text and rating as its only parameter. The data is extracted and put into a global dataframe variable called df.

preprocessing ( ):

The next function is called preprocessing. The function takes a dataframe with attributes document id, text and rating as its only parameter. The function removes stop words, punctuation, and words that are less than 4 characters long from the text attribute in the dataframe. The preprocessed text then replaces the old text within the given dataframe, saving the results in the initial dataframe used as a parameter.

dictionaries( ):

The next function is called dictionaries. The function takes a dataframe with attributes document id, text (stemmed from the previous function) and rating as its only parameter. The function creates a list of training and test documents to be later used in the classification algorithms. As well, returned are the following dictionaries:

- tf_idf {(word, document id) : tf x idf }
    - the tf x idf value for the given term-document id combo

- tf {(word, document id) : tf }
    - the tf value for the given term-document id combo

- binary_weights{(word, document_id), binary tf}

    - binary tf weights

- norm_tf {document id : norm }
    - the norm of each document calculated using tf values

- norm_tf_idf {document id : norm }
    - the norm of each document using tf x idf values

- norm_binary {document id : norm }
    - the norm of each document using binary weights

- inverted_index {word : [document ids] }
    - inverted index for the list of documents and words

roc( ):

This functions takes as paramters a dataframe, weighting measure and similairty measure. The funtion takes the parameters and calculates centroid vectors and uses Rocchio's method to classify test cases. The allowed weighting and similarity measures are the ones listed above in the overview of the system. The function returns the classificaiton accuracy.

knn( ):

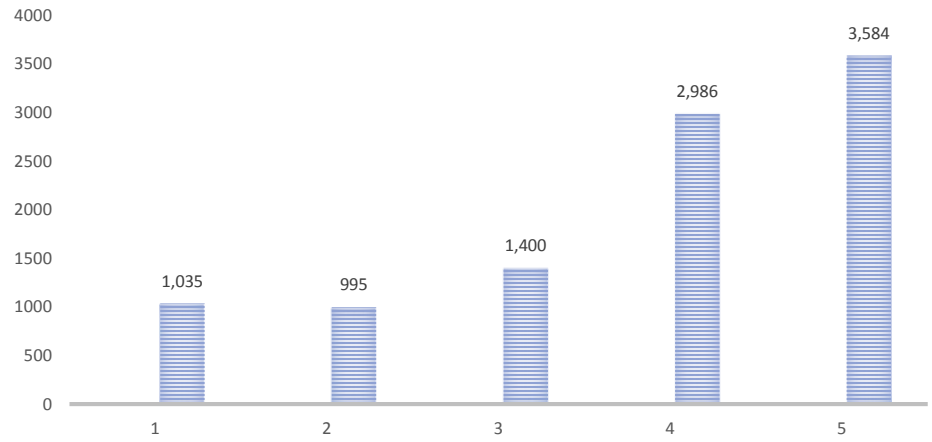
This function takes 3 paramaters, the first being the number of K's, the second being the weighting method and the third being the similarity measure to be used. The allowed weighting and similarity measures are the ones listed above in the overview of the system. The function then runs the K-nearest neighbor algorithm and returns the accuracy in predicting the ratings of each test case.

Testing

To test our code, we took a sample of 10 reviews and created the term document matrices by hand (see attached file) and found the norms, tf x idf values, cosine similarity, dice coefficient, prototype vectors and tested the classification methods. We then ran each of the 5 functions above for the same 10 reviews and compared the relevant values to the hand calculated values. Once we found the values were consistently calculated manually and by the program, we proceeded to run script on a larger sample size.

Results

Our sample dataset consisted of 10,000 reviews, 8,000 training and 2,000 testing, with the below rating distribution. As seen, the reviews skew high, with 4 and 5 star reviews representing more than 50% of the total samples.



Each weight-similarity-classification combination was run twice, with the results and runtimes shown below.

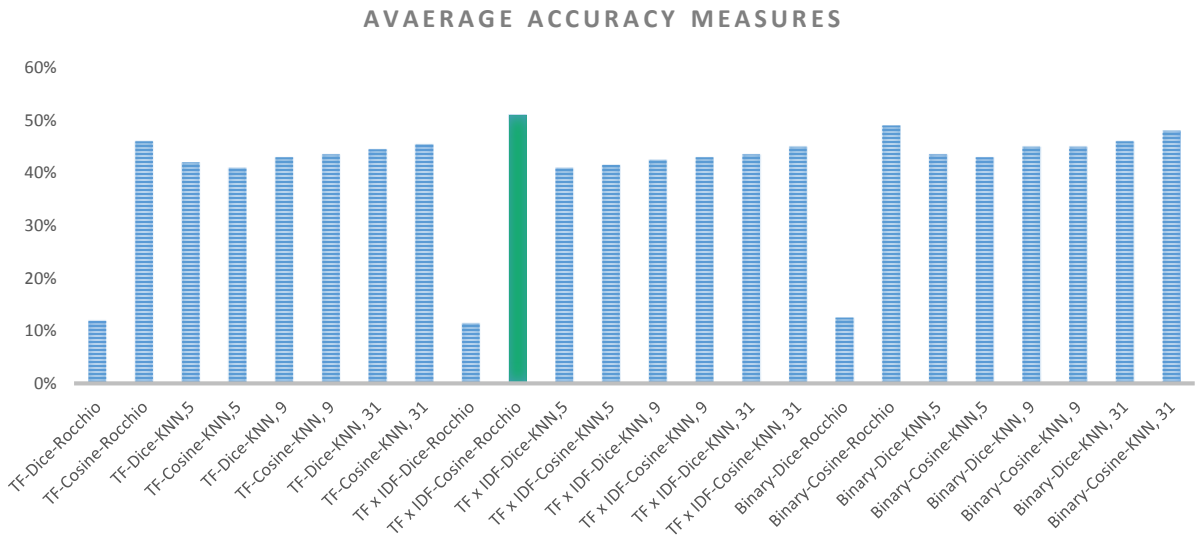
Weight	Similarity	Classification	Accuracy 1	Accuracy 2	Avg. Accuracy	Runtime 1	Runtime 2	Avg. Runtime
TF	Dice	Rocchio	12%	12%	12%	207	202	205
TF	Cosine	Rocchio	47%	45%	47%	199	203	201
TF	Dice	KNN,5	43%	41%	43%	404	408	406
TF	Cosine	KNN,5	41%	41%	41%	405	406	406
TF	Dice	KNN, 9	43%	43%	43%	406	410	408
TF	Cosine	KNN, 9	44%	43%	44%	400	409	405
TF	Dice	KNN, 31	46%	43%	46%	407	410	409
TF	Cosine	KNN, 31	45%	46%	45%	404	409	407
TF x IDF	Dice	Rocchio	11%	12%	11%	201	205	203
TF x IDF	Cosine	Rocchio	50%	52%	50%	205	209	207
TF x IDF	Dice	KNN,5	40%	42%	40%	426	422	424
TF x IDF	Cosine	KNN,5	41%	42%	41%	407	423	415
TF x IDF	Dice	KNN, 9	43%	42%	43%	420	422	421
TF x IDF	Cosine	KNN, 9	43%	43%	43%	401	410	406
TF x IDF	Dice	KNN, 31	45%	42%	45%	427	422	425
TF x IDF	Cosine	KNN, 31	45%	45%	45%	408	410	409
Binary	Dice	Rocchio	12%	13%	12%	203	208	206
Binary	Cosine	Rocchio	49%	49%	49%	203	202	203
Binary	Dice	KNN,5	45%	42%	45%	405	410	408
Binary	Cosine	KNN,5	44%	42%	44%	405	407	406
Binary	Dice	KNN, 9	46%	44%	46%	404	407	406
Binary	Cosine	KNN, 9	45%	45%	45%	401	418	410
Binary	Dice	KNN, 31	48%	44%	48%	404	444	424
Binary	Cosine	KNN, 31	48%	45%	48%	404	408	406

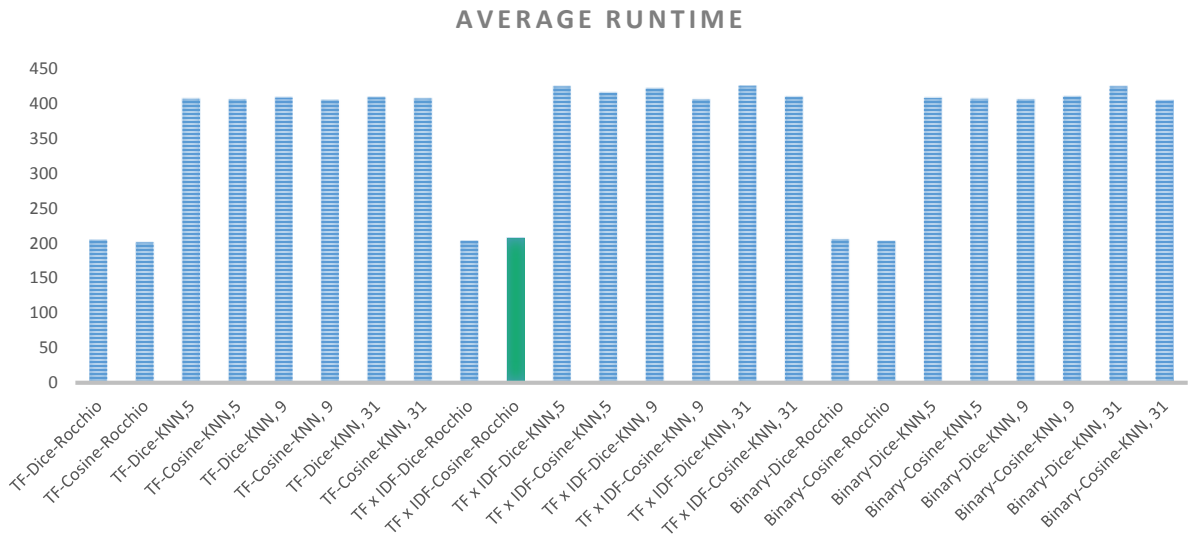
The first thing to point out, highlighted in green, is the lower runtimes using Rocchio's method. On average, a combination using Rocchio's method ran in 204 seconds roughly, while the average for any KNN method ran double that, with a max runtime of up to 444 seconds (seen in purple). Overall, this is not surprising, as Rocchio's method makes use of the centroid vectors where KNN must loop through every training set for each test case.

The second thing of notice, highlighted in yellow, is the poor accuracy measures when coupling Rocchio's method with Dice's similarity measure. Overall, the accuracies were approaching just 10%, or half the accuracy of simply guessing blind. If we eliminate those from consideration, Rocchio's Method on average classified 49% of the testing reviews correctly. As well, we notice in teal, the best combination was using tf x idf weighting with cosine similarity and Rocchio's classification method. This produced accuracy measures of 50% for both runs.

Looking at KNN, we see fairly consistent runtimes ranging from 401-444. As well, we see fairly consistent accuracy measures, ranging from 41% up to 48%. As a whole, KNN performed more consistent then Rocchio's method for classification, however, when using cosine similarity with tf x idf weights, Rocchio's method achieved the highest accuracies with a low runtime.

The below bar charts visualize the average runtime and accuracy measures for each combination. The green bar represents the tf x idf-cosine-Rocchio's combination, which performed the best when taking into account both accuracy and runtime.





Changes to Approach

Our system was initially built in an ad hoc manner, meaning we had a problem and began creating some script to find the measures we needed. However, being neither of us are software engineers or software designers by trait, we didn't consider a better way to implement the system until after the fact. Therefore, our code was condensed into a number of functions that each solved a few specific pieces of our project. It would have been useful to dedicate more time in the planning stage so our code could have been more versatile. For instance, to run the knn or roc function, one must first run the dataframe, preprocessing and stemming functions.

To no surprise either, we had an issue with version control, as we each wrote some code then condensed it. This created issues with variable names and deleting pieces of information another person's code relied on.

Another issue was the size of the dataset. In many cases, attempting to run the code on a 10,000 observation sample crashed laptops.

As well, this dataset was for an array of business, from shoe stores to restaurants. Dividing the dataset based on a business category could potentially produce better results.

Summary

After implementing and running the various weight-similarity-classification combinations, we can say that the words in a Yelp review can be used to predict the rating. Specifically, our goal was to find a methodology that achieved greater than a 20% accuracy, or a simple guess. If we were to implement a system at a larger scale, we would use tf x idf weights with cosine similarity measures and Rocchio's method for classification. Based on our results, this would reduce runtime and also preserve accuracy.

Such applications of this system could be in text analytics. For instance, we could further mine the text to the find the most influential words within a negative review or conversely, positive review.

Appendix

```
# -*- coding: utf-8 -*-

#import packages
import json
import os
import numpy as np
import pandas as pd
import random
import nltk
from nltk import PorterStemmer
import re
import string
import math
import operator
import time

#change the directiory
os.chdir('C:/Users/syarlag1/Desktop/yelp_dataset_challenge_academic_dataset')

#sample run
random.seed(99)
dataframe('yelp_academic_dataset_review.json')

start_pre = time.time()
preprocessing(df)
end_pre = time.time()
pretime = end_pre - start_pre

start_dict = time.time()
dictionaries(df)
end_dict = time.time()
dicttime = end_dict - start_dict

del df

##ROCCHIO
start_rocidf = time.time() ##rocchio's with tfidf weights, cosine similarity
roc(w = tf_idf, sim = 'cosine')
end_rocidf = time.time()
rocidftime = end_rocidf - start_rocidf

start_roctf = time.time() ##rocchio's with tf weights, cosine similarity
roc(w = tf, sim = 'cosine')
end_roctf = time.time()
roctftime = end_roctf - start_roctf

start_rocbinary = time.time() ##rocchio's with binary weights, cosine similarity
roc(w = binary, sim = 'cosine')
end_rocbinary = time.time()
rocbinarytime = end_rocbinary - start_rocbinary

start_drocidf = time.time() ##rocchio's with tfidf weights, dice similarity
roc(w = tf_idf, sim = 'dice')
end_drocidf = time.time()
rocdidftime = end_drocidf - start_drocidf

start_droctf = time.time() ##rocchio's with tf weights, dice similarity
```

```
roc(w = tf, sim = 'dice')
end_droctf = time.time()
rocdtftime = end_droctf - start_droctf

start_drocbinary = time.time() ##rocchio's with binary weights, dice similarity
roc(w = binary, sim = 'dice')
end_drocbinary = time.time()
rocdbinarytime = end_drocbinary - start_drocbinary


#5NN
start_5nnidf = time.time() ##5-NN with tf_idf weights, cosine similarity
knn(k=5, w=tf_idf, sim = 'cosine')
end_5nnidf = time.time()
idf5nntime = end_5nnidf - start_5nnidf

start_5nntf = time.time() ##5-NN with tf weights, cosine similarity
knn(k=5, w=tf, sim = 'cosine')
end_5nntf = time.time()
tf5nntime = end_5nntf - start_5nntf

start_5nnbinary = time.time() ##5-NN with binary weights, cosine similarity
knn(k=5, w=binary, sim = 'cosine')
end_5nnbinary = time.time()
binary5nntime = end_5nnbinary - start_5nnbinary

start_d5nnidf = time.time() ##5-NN with tf_idf weights, dice similarity
knn(k=5, w=tf_idf, sim = 'dice')
end_d5nnidf = time.time()
idfd5nntime = end_d5nnidf - start_d5nnidf

start_d5nntf = time.time() ##5-NN with tf weights, dice similarity
knn(k=5, w=tf, sim = 'dice')
end_d5nntf = time.time()
tfd5nntime = end_d5nntf - start_d5nntf

start_d5nnbinary = time.time() ##5-NN with binary weights, dice similarity
knn(k=5, w=binary, sim = 'dice')
end_d5nnbinary = time.time()
binary5nntime = end_d5nnbinary - start_d5nnbinary

#9NN
start_9nnidf = time.time() ##9-NN with tf_idf weights, cosine similarity
knn(k=9, w=tf_idf, sim = 'cosine')
end_9nnidf = time.time()
idf9nntime = end_9nnidf - start_9nnidf

start_9nntf = time.time() ##9-NN with tf weights, cosine similarity
knn(k=9, w=tf, sim = 'cosine')
end_9nntf = time.time()
tf9nntime = end_9nntf - start_9nntf

start_9nnbinary = time.time() ##9-NN with binary weights, cosine similarity
knn(k=9, w=binary, sim = 'cosine')
end_9nnbinary = time.time()
binary9nntime = end_9nnbinary - start_9nnbinary

start_d9nnidf = time.time() ##9-NN with tf_idf weights, dice similarity
knn(k=9, w=tf_idf, sim = 'dice')
end_d9nnidf = time.time()
```

```
idfd9nntime = end_d9nnidf - start_d9nnidf

start_d9nntf = time.time() ##9-NN with tf weights, dice similarity
knn(k=9, w=tf, sim = 'dice')
end_d9nntf = time.time()
tfd9nntime = end_d9nntf - start_d9nntf

start_d9nnbinary = time.time() ##9-NN with binary weights, dice similarity
knn(k=9, w=binary, sim = 'dice')
end_d9nnbinary = time.time()
binary9nntime = end_d9nnbinary - start_d9nnbinary

#31NN
start_31nnidf = time.time() ##31-NN with tf_idf weights, cosine similarity
knn(k=31, w=tf_idf, sim = 'cosine')
end_31nnidf = time.time()
idf31nntime = end_31nnidf - start_31nnidf

start_31nntf = time.time() ##31-NN with tf weights, cosine similarity
knn(k=31, w=tf, sim = 'cosine')
end_31nntf = time.time()
tf9nntime = end_31nntf - start_31nntf

start_31nnbinary = time.time() ##31-NN with binary weights, cosine similarity
knn(k=9, w=binary, sim = 'cosine')
end_31nnbinary = time.time()
binary9nntime = end_31nnbinary - start_31nnbinary

start_d31nnidf = time.time() ##31-NN with tf_idf weights, dice similarity
knn(k=9, w=tf_idf, sim = 'dice')
end_d31nnidf = time.time()
idfd9nntime = end_d31nnidf - start_d31nnidf

start_d31nntf = time.time() ##31-NN with tf weights, dice similarity
knn(k=9, w=tf, sim = 'dice')
end_d31nntf = time.time()
tfd9nntime = end_d31nntf - start_d31nntf

start_d31nnbinary = time.time() ##31-NN with binary weights, dice similarity
knn(k=9, w=binary, sim = 'dice')
end_d31nnbinary = time.time()
binary9nntime = end_d31nnbinary - start_d31nnbinary


#**************************************************************************
#provide a path to a json file...will open and return a dataframe**********
#**************************************************************************
def dataframe(path):
    '''Take the path of the json file and then creates a dataframe with
    the reviews and ratings from the file
    '''

    file = open(path, 'r')
    dataAll = file.read().split('\n')

    #get a 10,000 row sample
    data = random.sample(dataAll, 10000)

    #create the idd, review and ratings empty lists
    idd = []
    reviews = []
```

```
    ratings = []

    #extract the entries within the data sample of 10k
    for entry in data:
        extract = json.loads(entry)
        idd.append(extract['review_id'])
        reviews.append(extract['text'])
        ratings.append(extract['stars'])


    #create a dataframe of the json data
    data_dict = {'id':idd,'reviews': reviews, 'ratings': ratings}
    global df
    df = pd.DataFrame(data_dict);


#*****************************************************************************
#returns the original dataframe, with stemmed text*****************************
#*****************************************************************************
def preprocessing(df):
    '''Takes the previously created dataframe and preprocesses the data:
    stemming, stop word removal, punctuation removal.
    '''


    stopWords = ['a','able','about','across','after','all','almost','also',
                 'am','among','an','and','any','are','as','at','be','because',
                 'been','but','by','can','cannot','could','dear','did','do',
                 'does','either','else','ever','every','for','from','get','got'
                 ,'had','has','have','he','her','hers','him','his','how',
                 'however','i','if','in','into','is','it','its','just','least'
                 ,'let','like','likely','may','me','might','most','must','my',
                 'neither','no','nor','not','of','off','often','on','only',
                 'or','other','our','own','rather','said','say','says','she'
                 ,'should','since','so','some','than','that','the','their',
                 'them','then','there','these','they','this','tis','to',
                 'too','twas','us','wants','was','we','were','what','when',
                 'where','which','while','who','whom','why','will','with',
                 'would','yet','you','your']

    punctuations = ['"'," ",".","/",";","'","?","&","-", ",","!", "]", "["]

    collect = []

    for i in df.index:
        text = df.xs(i)['reviews'].lower().strip()
        textLst = re.findall(r"\w+(?:[-']\w+)*|'|[-.(]+|\S\w*",text)

        countof = True

        while countof:
            y = 0
            for i,j in enumerate(textLst):

                textLst[i] = PorterStemmer().stem_word(j.lower())

                if j in stopWords or j in punctuations or len(j) <= 3 or j.isdigit():

                    textLst.pop(i)
                    y = y+1
                    continue
```

```
                for c in punctuations:
                    if c in j:
                        textLst.pop(i)
                        y = y+1
                        break
            if y == 0:
                countof = False

        collect.append(" ".join(textLst))

    df['reviews'] = collect;

#*****************************************************************************
#create an inverted index*****************************************************
#*****************************************************************************
def dictionaries(df):
    '''This function takes the preprocessed dataframe and creates inverted index'
    as dictionaries: a rating dictionary with the ID and rating values; the tf dictiona
    ry
    with words, postings, and their corresponding term frequencies; a tf_idf dictionary

    with words, postings, and their corresponding tfxidf values; inverted_index diction
    ary with
    the words and their corresponding postings; norm_tf with documents with their norma
    lization
    factors using term frequencies; and  norm_tf_idf with documents and their normaliza
    tion
    factors using tfxidf values
    '''

    #initalize the needed dictionaries
    global rating
    global tf
    global tf_idf
    global inverted_index
    global trainDoc
    global testDoc
    global norm_tf
    global norm_tf_idf
    global norm_binary
    global binary

    rating = {}
    tf_idf = {}
    norm = {}
    inverted_index = {}
    tf = {}
    norm_tf_idf = {}
    norm_2 = {}
    norm_tf = {}
    norm_3 = {}
    norm_binary = {}
    binary = {}

    for index,row in df.iterrows():

        #put each column in a list
        review_id = row[0]
        review_text = row[2]
        review_text = review_text.split(' ')
        review_rating = row[1]
```

```
296.
297.         #add to the reviews dictionary
298.         rating[review_id] = review_rating
299.         norm_2[review_id] = []
300.         norm[review_id] = []
301.         norm_3[review_id] = []
302.
303.         #training and testing split, 80%-20%
304.         docLst = np.array(list(rating.keys()))
305.
306.         #np.random.shuffle(docLst)
307.         trainDoc = docLst[:(round(0.8*len(docLst)))]
308.         testDoc = docLst[(round(0.8*len(docLst))):]
309.
310.         #create the inverted index {term : [doct list]}
311.         for word in review_text:
312.
313.             if word not in inverted_index.keys():
314.                 inverted_index[word] = [review_id]
315.             else:
316.                 inverted_index[word].append(review_id)
317.
318.         for key,value in inverted_index.items():
319.             inverted_index[key] =  list(set(value))
320.
321.
322.     for index,row in df.iterrows():
323.
324.         #put each column in a list
325.         review_id = row[0]
326.         review_text = row[2]
327.         review_text = review_text.split(' ')
328.         review_rating = row[1]
329.
330.         for word in review_text:
331.
332.             binary[(word,review_id)] = 1
333.
334.             #tf dictionary
335.             counter = 0
336.             counter = review_text.count(word)
337.             tf[(word,review_id)] = counter
338.
339.             #get idf values
340.             count = len(rating)
341.             length = len(inverted_index[word])
342.             num = count/length
343.             idf = math.log(num,2)
344.
345.             #tf_idf postings
346.             posting = idf * counter
347.             tf_idf[(word, review_id)] = posting
348.
349.
350.     for key,values in tf.items():
351.         norm_2[key[1]].append(values)
352.
353.     for key,values in tf_idf.items():
354.         norm[key[1]].append(values)
355.         norm_3[key[1]].append(1)
356.
```

```
357.
358.
359.    for key,value in norm_2.items():
360.        norm_tf[key] = sum(i*i for i in value)
361.
362.    for key, value in norm.items():
363.        norm_tf_idf[key] = sum(i*i for i in value)
364.
365.    for key,value in norm_3.items():
366.        norm_binary[key] = sum(i*i for i in value)
367.
368.
369.
370.####VERSION 1: CATEGORIZING DOCUMENTS USING ROCCHIOS WITH TFXIDF TERM WEIGHTS AND COSIN
    E SIMILARITY
371.
372.#*****************************************************************************
373.#Rochios Method - First getcentroids vectors for each class*******************
374.#*****************************************************************************
375.
376.def roc(w = tf_idf, sim = 'cosine'):
377.    '''with the term weights specified, this function finds the required centroid vecto
    rs,
378.    calculates similarity with the test documents using Rocchio's method, predicts the

379.    document category, and then returns the prediction accuracy
380.    '''
381.
382.    global protoVec
383.    global termw
384.    global rocSim
385.    global accuracyRoc
386.    global predActRating
387.
388.    if w == tf_idf:
389.        n = norm_tf_idf
390.
391.    if w == tf:
392.        n = norm_tf
393.
394.    if w == binary:
395.        n = norm_binary
396.
397.
398.    docs_rating = {}
399.    protoVec = {}
400.    termW = {}
401.
402.    uniqueRatings =  set(rating.values())
403.
404.    for x in uniqueRatings:
405.        count = 0
406.
407.        for doc,rate in rating.items():
408.            if rate == x and doc in trainDoc:
409.                count = count + 1
410.                docs_rating[rate] = count
411.
412.
413.        for term,ID in w.keys():
414.            if ID in trainDoc and rating[ID] == x:
```

```
                if term not in termW.keys():
                    termW[term] = w[(term,ID)]

                else:
                    termW[term] = termW[term] + w[(term,ID)]
            protoVec[x] = termW

        termW = {};


    dict_pNorm = {} #the normalization factors for the protovectors

    for x in protoVec.keys():
        sum = 0
        for term,wi in protoVec[x].items():
            sum = sum + wi**2
        dict_pNorm[x] = sum;

    rocSim = {}

    for doc in testDoc:
        value = {}
        for x in protoVec.keys():
            sum = 0
            for term,wi in protoVec[x].items():
                if (term,doc) in w.keys():
                    if sim == 'cosine':
                        sum = sum + w[(term,doc)]*protoVec[x][term]
                    elif sim == 'dice':
                        sum = sum + 2*w[(term,doc)]*protoVec[x][term]
            if dict_pNorm[x] != 0:
                if sim == 'cosine':
                    value[x] =  sum/(dict_pNorm[x]*n[doc])**0.5
                if sim == 'dice':
                    value[x] =  sum/(dict_pNorm[x]+n[doc])

            else:
                value[x] = 0
            rocSim[doc] = value;

    predActRating = {}

    for doc in rocSim.keys():
        predActRating[doc] = [max(rocSim[doc], key=rocSim[doc].get), rating[doc]]

    #Computing accuracy
    count = 0

    for val in predActRating.values():
        if val[0] == val[1]:
            count = count + 1

    accRoc = count/len(testDoc)

    print(accRoc)


####VERSION 2: CATEGORIZING DOCUMENTS USING K-
    NN WITH TFXIDF TERM WEIGHTS AND COSINE SIMILARITY

```

```
475.def knn(k = 5, w=tf_idf, sim = 'cosine'):
476.    '''Takes a value for k (5 is default) and calculates the predicted rating of the

477.    test documents based on its cosine similarity with the training documents. In the

478.    where there is a tie, we pick the higher rating as the predicted rating
479.    '''
480.    global allSim
481.    global accKnn
482.    global ratingCounts
483.    global knnpredRatings
484.
485.    if w == tf_idf:
486.        n = norm_tf_idf
487.    if w == tf:
488.        n = norm_tf
489.    if w == binary:
490.        n = norm_binary
491.
492.    allSim = {}
493.
494.    for testdoc in testDoc:
495.        allSim[testdoc] = {}
496.        for word, traindoc in w.keys():
497.            if (word, testdoc) in w.keys():
498.                if traindoc != testdoc:
499.                    if sim == 'cosine':
500.                        if traindoc in allSim[testdoc].keys():
501.                            allSim[testdoc][traindoc] = allSim[testdoc][traindoc] + w[(
    word, testdoc)]*w[(word, traindoc)]
502.                        else:
503.                            allSim[testdoc][traindoc] = w[(word, testdoc)]*w[(word, tra
    indoc)]
504.                    elif sim == 'dice':
505.                        if traindoc in allSim[testdoc].keys():
506.                            allSim[testdoc][traindoc] = allSim[testdoc][traindoc] + 2*w
    [(word, testdoc)]*w[(word, traindoc)]
507.                        else:
508.                            allSim[testdoc][traindoc] = 2*w[(word, testdoc)]*w[(word, t
    raindoc)]
509.
510.    for testdoc in allSim.keys(): ##normalizing to find the cosine similarity
511.        for traindoc in allSim[testdoc].keys():
512.            if sim == 'cosine':
513.                allSim[testdoc][traindoc] = allSim[testdoc][traindoc]/(n[testdoc]*n[tra
    indoc])**0.5
514.            elif sim == 'dice':
515.                allSim[testdoc][traindoc] = allSim[testdoc][traindoc]/(n[testdoc]+n[tra
    indoc])
516.
517.    #subsetting to only the nearest neighbours
518.    for testdoc in allSim.keys():
519.        allSim[testdoc] = sorted(allSim[testdoc].items(),key = operator.itemgetter(1),
    reverse = True)[0:k]
520.
521.    #based on the nearest neighbours, we predict the rating, in a tie, we pick the high
    est rating
522.    knnRatings = {}
523.
524.    for testdoc in allSim.keys():
525.        knnRatings[testdoc] = {}
```

```
526.         for traindoc in allSim[testdoc]:
527.             knnRatings[testdoc][traindoc[0]] = rating[traindoc[0]]
528.
529.     ratingCounts = {}
530.     for testdoc in knnRatings.keys():
531.         ratingCounts[testdoc] = {}
532.         for rate in set(knnRatings[testdoc].values()):
533.             ratingCounts[testdoc][rate] = sum(1 for x in knnRatings[testdoc].values() i
    f x == rate)
534.
535.     knnpredRatings = {}
536.
537.     for testdoc in ratingCounts.keys():
538.        knnpredRatings[testdoc] = sorted(ratingCounts[testdoc].items(),key = operator.it
    emgetter(1,0), reverse = True)[0]
539.
540.
541.     #Computing accuracy
542.     count = 0
543.
544.     for doc in knnpredRatings.keys():
545.         if knnpredRatings[doc][0] == rating[doc]:
546.             count = count + 1
547.
548.     accKnn = count/len(testDoc)
549.
550.     print(accKnn)
551.
552.
553.##########################################################################################
    ################################
```