CS168 Spring Assignment 2

SUNet ID(s): sainid jenny123
Name(s): Divya Saini Jenny Lu
Collaborators: [list all the people you worked with outside your group]

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Part 1

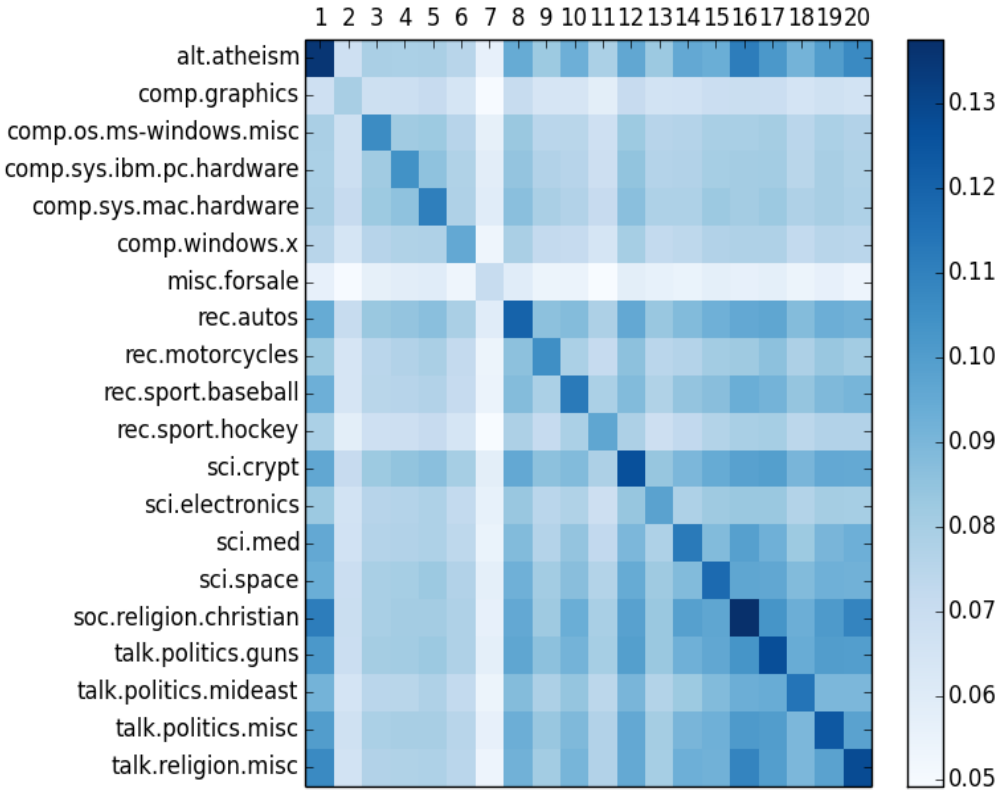
(a)

```
import csv
import numpy as np
from scipy.sparse import csr_matrix
import math
import matplotlib.pyplot as plt
import warnings

#import csv file, read into global variable 'matrix'
def read_data():
        reader = csv.reader(open('data/data50.csv', 'rb'))
        #matrix create here
        global matrix
        matrix = np.zeros((1000, 61067))
        for row in reader:
                articleid = int(row[0]) - 1
                wordid = int(row[1]) - 1
                wordcount = int(row[2])

                matrix[articleid][wordid] = wordcount
        matrix = csr_matrix(matrix)
        return matrix

#read group names into global variable 'groupNames'
def readGroupNames():
        reader = csv.reader(open('data/groups.csv','rb'))
        global groupNames
        groupNames = []
        for row in reader:
                groupNames.append(row[0])
        return groupNames

#given two bag of word vectors in CSR format, calculate cosine similarity
```

```
def calculateCosineSimilarity(aWordVector, bWordVector):
        aNorm = np.linalg.norm(aWordVector.toarray())
        bNorm = np.linalg.norm(bWordVector.toarray())
        dotProduct = np.dot(aWordVector.toarray(), bWordVector.toarray().T)
        similarity = dotProduct / float((aNorm * bNorm))
        return similarity

#given two bag of word vectors in CSR format, calculate Jaccard similarity
def calculateJaccardSimilarity(aWordVector, bWordVector):
        return np.sum(np.minimum(aWordVector.toarray(),bWordVector.toarray())
            )/np.sum(np.maximum(aWordVector.toarray(), bWordVector.toarray()))

#given two bag of word vectors in CSR format, calculate L2 similarity
def calculateL2Similarity(aWordVector, bWordVector):
        return -1.0 * math.sqrt(np.sum(np.square(np.subtract(aWordVector.
            toarray(), bWordVector.toarray()))))


#given which groups and the index of the articles within the groups, return similarity
     based on similarity code
def similarity(a, articleAIndex, b, articleBIndex, similarityCode):
        #for news source a, grab that articleAIndexth word vector
        #for news source b, grab the articleBIndexth word vector
        aWordVector = matrix[a * 50 + articleAIndex]
        bWordVector = matrix[b * 50 + articleBIndex]



        if(similarityCode == 1):
                return calculateJaccardSimilarity(aWordVector, bWordVector)
        elif(similarityCode == 2):
                return calculateL2Similarity(aWordVector, bWordVector)
        else:
                return calculateCosineSimilarity(aWordVector, bWordVector)

#compare all articles between groups a and b based on similarity metric
def compareArticles(a, b, similarityCode):
        averageSim = 0.0
        for k in range(0, 50):
                for l in range(0, 50):
                        averageSim += similarity(a, k, b, l, similarityCode)
        # averageSim *= 2
        averageSim /= float(2500.0)
```

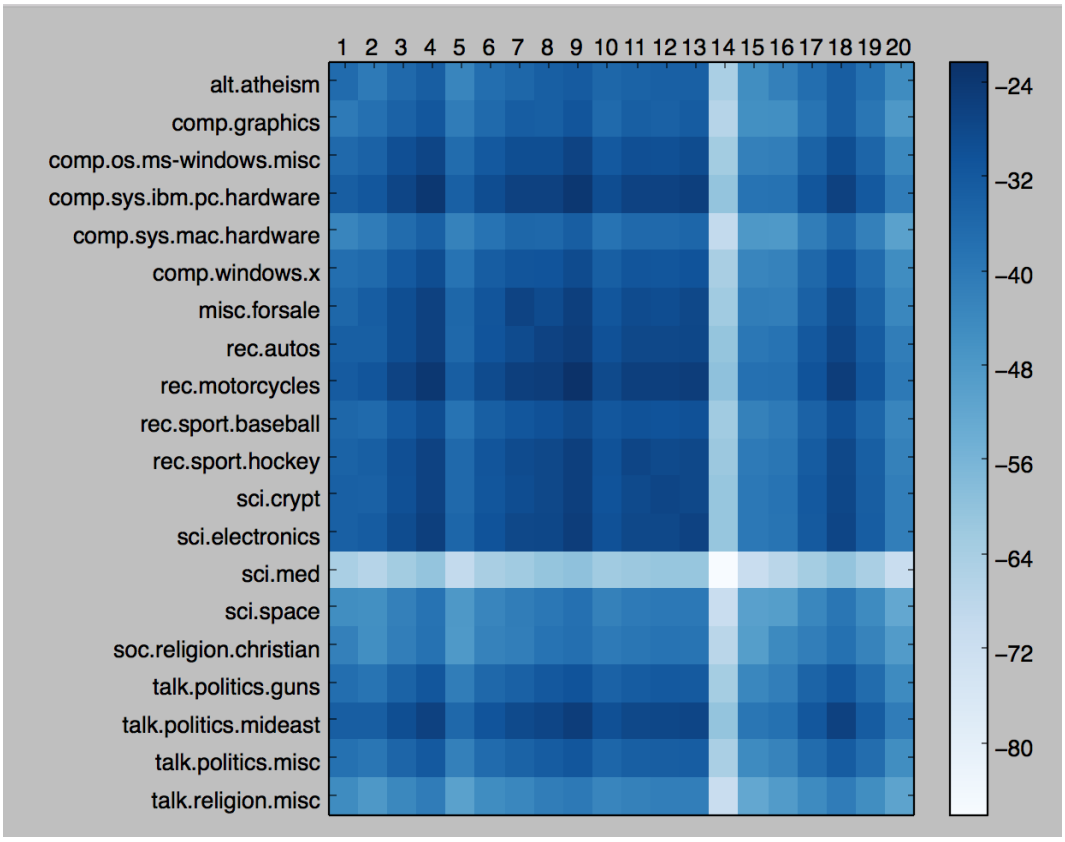
```
        return averageSim

#create 20x20 plot, where each entry corresponds to average similarity over all ways
    of pairing up one article from A with one article from B.
def createPlotMatrix(similarityCode):
        plotMatrix = np.zeros((20, 20))
        for i in range(0, 20):
                for j in range(i + 1):
                        averageSimilarity = compareArticles(i, j, similarityCode)
                        plotMatrix[i][j] = averageSimilarity
                        plotMatrix[j][i] = averageSimilarity
        makeHeatMap(plotMatrix, groupNames, plt.cm.Blues, 'heatMap1')

def makeHeatMap(data, names, color, outputFileName):
        #to catch "falling back to Agg" warning
        with warnings.catch_warnings():
                warnings.simplefilter("ignore")
                #code source: http://stackoverflow.com/questions/14391959/
                    heatmap-in-matplotlib-with-pcolor
                fig, ax = plt.subplots()
                #create the map w/ color bar legend
                heatmap = ax.pcolor(data, cmap=color)
                cbar = plt.colorbar(heatmap)

                # put the major ticks at the middle of each cell
                ax.set_xticks(np.arange(data.shape[0])+0.5, minor=False)
                ax.set_yticks(np.arange(data.shape[1])+0.5, minor=False)

                # want a more natural, table-like display
                ax.invert_yaxis()
                ax.xaxis.tick_top()


                ax.set_xticklabels(range(1, 21))
                ax.set_yticklabels(names)

                plt.tight_layout()

                plt.savefig(outputFileName, format = 'png')
                plt.show()
                plt.close()
```

```
def averageClassificationPrecision(m):
        averageErrors = 0
        for i in range(m.shape[0]):
                averageErrors += m[i,i]
        print averageErrors/float(1000)

read_data()
readGroupNames()
# baselineClassification()

createPlotMatrix(3)
```

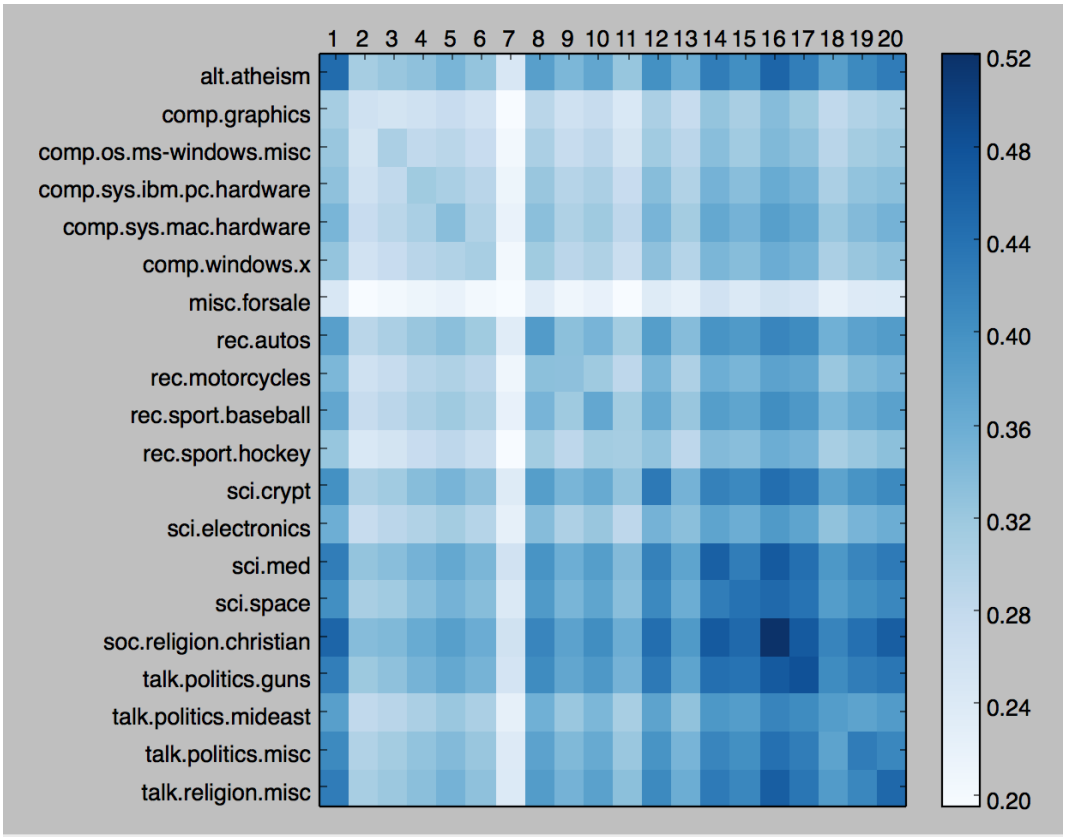


(b) Jaccard:

L2:


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
alt.atheism
comp.graphics
comp.os.ms-windows.misc
comp.sys.ibm.pc.hardware
comp.sys.mac.hardware
comp.windows.x
misc.forsale
rec.autos
rec.motorcycles
rec.sport.baseball
rec.sport.hockey
sci.crypt
sci.electronics
sci.med
sci.space
soc.religion.christian
talk.politics.guns
talk.politics.mideast
talk.politics.misc
talk.religion.misc
−24
−32
−40
−48
−56
−64
−72
−80

Cosine:



(c) Jaccard and cosine similarity both seem to be reasonable measures.

In the case of Jaccard similarity, most articles are classified to have the highest similarity to other articles within the same group. We would expect Jaccard to be a good similarity metric in this case because the similarity metric tends to penalize long articles on the same topic when compared to a short article on the same topic, meaning if a long article and short article are about the same topics, the Jaccard similarity might not classify them as extremely similar by way of the fact that the intersection of word frequencies might be low–more so because of the short length of the document rather than a fundamental difference in the topics. However, in this case, along the diagonal, the average Jaccard Similarity scores between articles of the same newsgroup are rightfully high by the fact that the same length articles are being compared to one another, which minimizes the unnecessary penalization of differing document length word frequencies, as described above. Therefore, at least along the diagonal, we ensure a relatively solid high capture of similarity between articles of extremely similar topic and source content. Additionally, as we spoke about in class, Jaccard Similarity tends to works well with sparse data.

Cosine similarity also identifies the similarities between articles of the same group, but it is more adept at identifying similarities between different newsgroups as well. Cosine

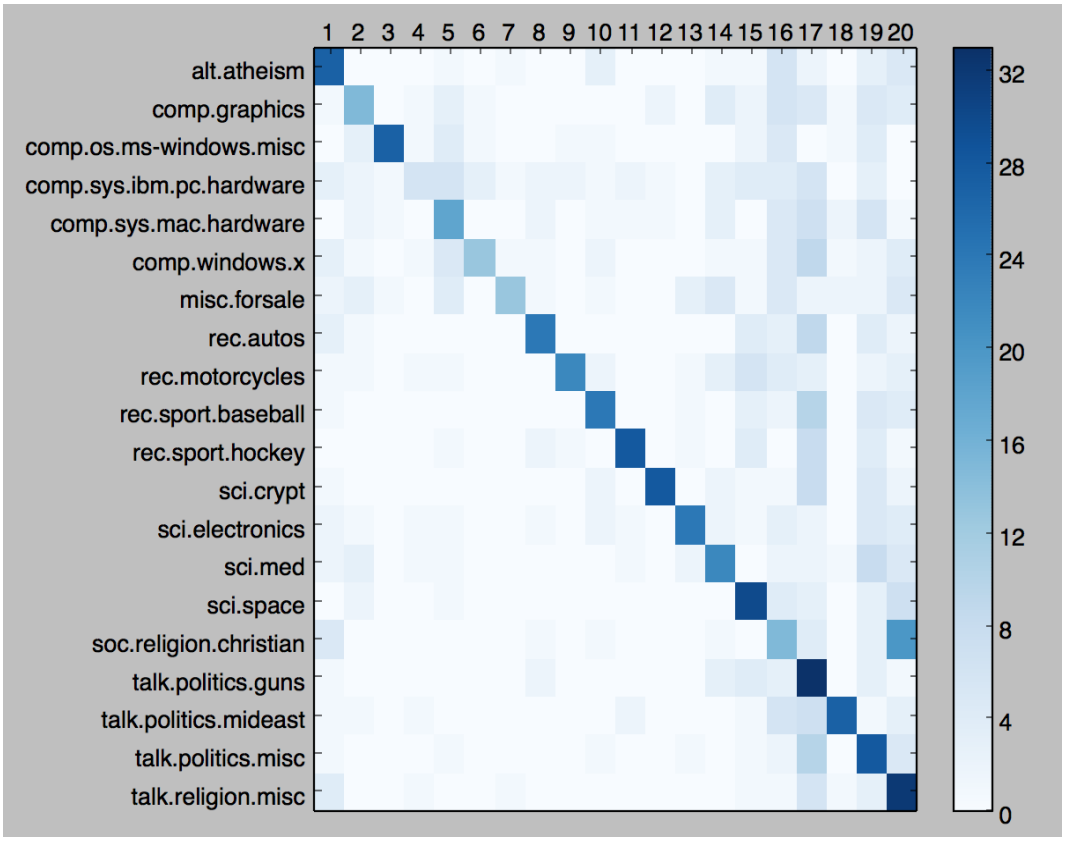
similarity normalizes the document vectors so that severe differences in length are not penalized.

Several groups appear to be more similar; one noticeable pattern is that all newsgroups that deal with religion (alt.atheism, soc.religion.christian, talk.religion.misc) seem to have higher average similarity measures with one another. This makes sense because the terminology used in religion documents are fairly common amongst different religions and contexts.

We also see from the cosine similarity heat map that topics such as soc.religion.christian and talk.politics.guns and soc.religion.christian and talk.politics.misc are similar, largely because these topics tend to have a lot of overlap, thus resulting in the use of similar words.

Part 2

(a)



precision: 0.456

```
    #used for baseline calculation, find the article with the max cosine similarity
        score to the current article
def greatestSimiliarity(hostArticle):
```

```
        currMax = 0.0
        maxArticle = None
        for article in range(matrix.shape[0]):
                if(hostArticle != article):
                        similarity = calculateCosineSimilarity(matrix[hostArticle],
                            matrix[article])
                        if(similarity > currMax):
                                currMax = similarity
                                maxArticle = article
        return maxArticle


#used to compare cosine similarity scores between every article
def baselineClassification():

        nearestNeighborCount = np.zeros ((20,20))
        for article in range(matrix.shape[0]):
                y = greatestSimiliarity(article)
                nearestNeighborCount[article/50, y/50] += 1

        # makeHeatMap(nearestNeighborCount, groupNames, plt.cm.Blues, '
            heatMap1')
        averageClassificationPrecision(nearestNeighborCount)
```

(b) Why is the matrix in (a) not symmetric? We are no longer looking at the average similarity between group a and group b, rather, each entry in the plot represents the number of articles in group A that have their nearest neighbor in group B.

Consider that A has the highest similarity with B. Now that we're looking for the largest similarity, it need not be the case that the largest similarity for A and B are the exact same since there could exist another article out there that is much more similar to B than A is.

This is different from the earlier case because the assumption there was just that the similarity between A and B is the same as the similarity between B and A. Whereas, here, the matrix slot for A,B represents the fact that B was a closest neighbor to A. But the slot B,A in the matrix need not contain the same value since the document B could have found a *maximum* similarity score higher with a different document than A. The fact that we're now storing the counts of the maximum over the entire universe of documents removes the symmetry that existed previously, as described above.

(c)

```
 import csv
import numpy as np
from scipy.sparse import csr_matrix
```

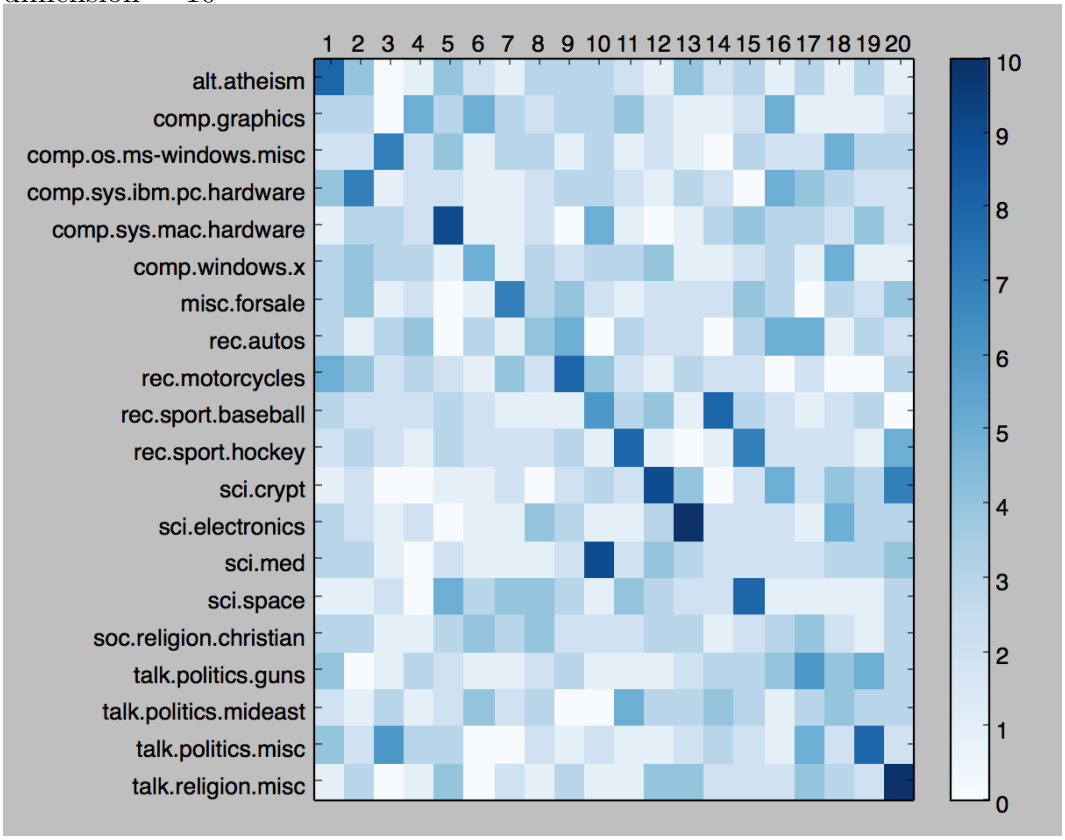
```
import math
import matplotlib.pyplot as plt
import warnings

def read_data():
        reader = csv.reader(open('data/data50.csv', 'rb'))
        #matrix create here
        global matrix
        matrix = np.zeros((1000, 61067))
        for row in reader:
                articleid = int(row[0]) - 1
                wordid = int(row[1]) - 1
                wordcount = int(row[2])

                matrix[articleid][wordid] = wordcount
        matrix = csr_matrix(matrix)
        return matrix

#read group names into global variable 'groupNames'
def readGroupNames():
        reader = csv.reader(open('data/groups.csv','rb'))
        global groupNames
        groupNames = []
        for row in reader:
                groupNames.append(row[0])
        return groupNames

def calculateCosineSimilarity(aWordVector, bWordVector):
        aNorm = np.linalg.norm(aWordVector)
        bNorm = np.linalg.norm(bWordVector)
        dotProduct = np.dot(aWordVector, bWordVector.T)
        similarity = dotProduct / float((aNorm * bNorm))
        return similarity

def greatestSimiliarity(hostArticle, matrix):
        currMax = 0.0
        maxArticle = None
        for article in range(matrix.shape[0]):
                if(hostArticle != article):
                        similarity = calculateCosineSimilarity(matrix[hostArticle],
                            matrix[article])
                        if(similarity > currMax):
```

```
                                currMax = similarity
                                maxArticle = article
        return maxArticle


def baselineClassification(matrix):
        nearestNeighborCount = np.zeros ((20,20))
        for article in range(matrix.shape[0]):
                y = greatestSimiliarity(article, matrix)
                nearestNeighborCount[article/50, y/50] += 1
        print nearestNeighborCount
        makeHeatMap(nearestNeighborCount, groupNames, plt.cm.Blues, 'heatMap1
            ')
        averageClassificationError(nearestNeighborCount)




def makeHeatMap(data, names, color, outputFileName):
        #to catch "falling back to Agg" warning
        with warnings.catch_warnings():
                warnings.simplefilter("ignore")
                #code source: http://stackoverflow.com/questions/14391959/
                    heatmap-in-matplotlib-with-pcolor
                fig, ax = plt.subplots()
                #create the map w/ color bar legend
                heatmap = ax.pcolor(data, cmap=color)
                cbar = plt.colorbar(heatmap)

                # put the major ticks at the middle of each cell
                ax.set_xticks(np.arange(data.shape[0])+0.5, minor=False)
                ax.set_yticks(np.arange(data.shape[1])+0.5, minor=False)

                # want a more natural, table-like display
                ax.invert_yaxis()
                ax.xaxis.tick_top()


                ax.set_xticklabels(range(1, 21))
                ax.set_yticklabels(names)

                plt.tight_layout()
```

```
                plt.savefig(outputFileName, format = 'png')
                plt.show()
                plt.close()


def dimensionReductionFunction(dimension):
        #matrix M
        randList = []
        for i in range(dimension):
                rand = np.random.normal(0, 1.0, 61067)
                randList.append(list(rand))


        M = np.array(randList)
        reducedList = []
        for article in matrix:
                #multiply article transposed by M
                currArticle = article.toarray().T
                newVector = np.dot(M,currArticle)

                newVectorTransposed = newVector.T
                reducedList.append(newVectorTransposed)


        reducedMatrix = np.array(reducedList)
        reducedMatrix = np.reshape(reducedMatrix, (1000,dimension))
        # print reducedMatrix

        #calculate cosine similarity between articles in reducedMatrix
        baselineClassification(reducedMatrix)


def averageClassificationPrecision(matrix):
        averageErrors = 0
        for i in range(matrix.shape[0]):
                averageErrors += matrix[i,i]
        print averageErrors/float(1000)



read_data()
```

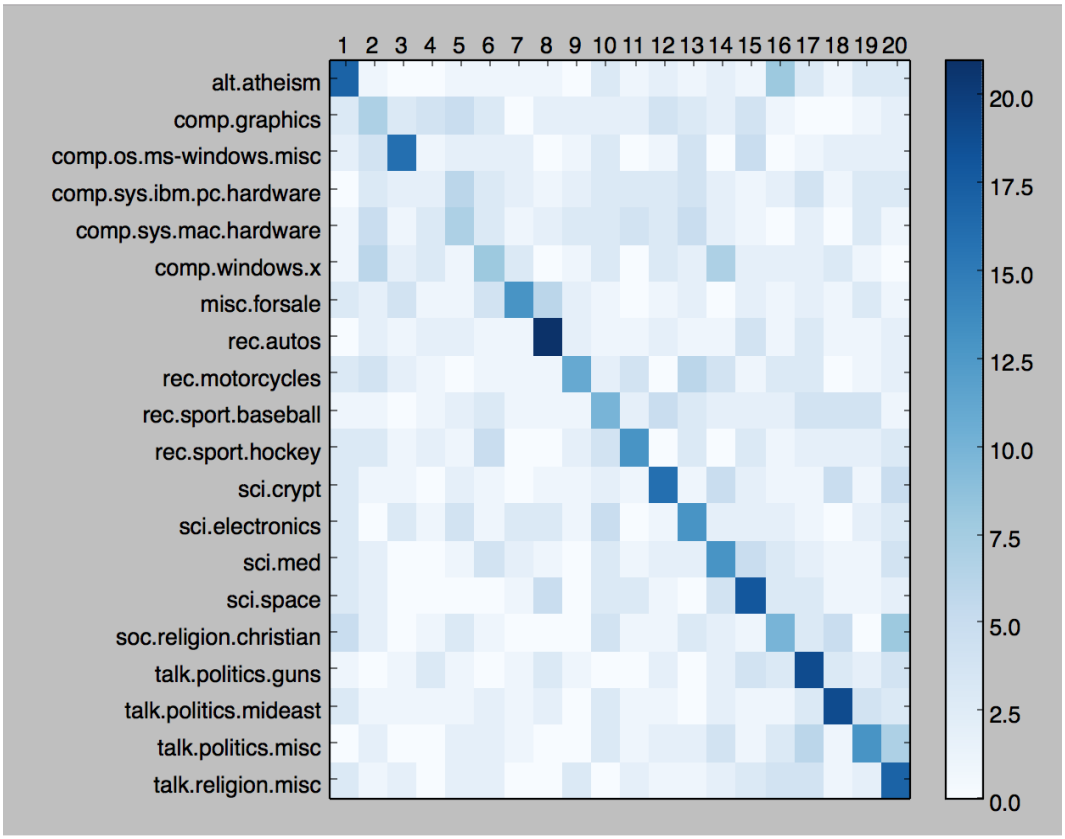
```
readGroupNames()
dimensionReductionFunction(10)
```

dimension = 10



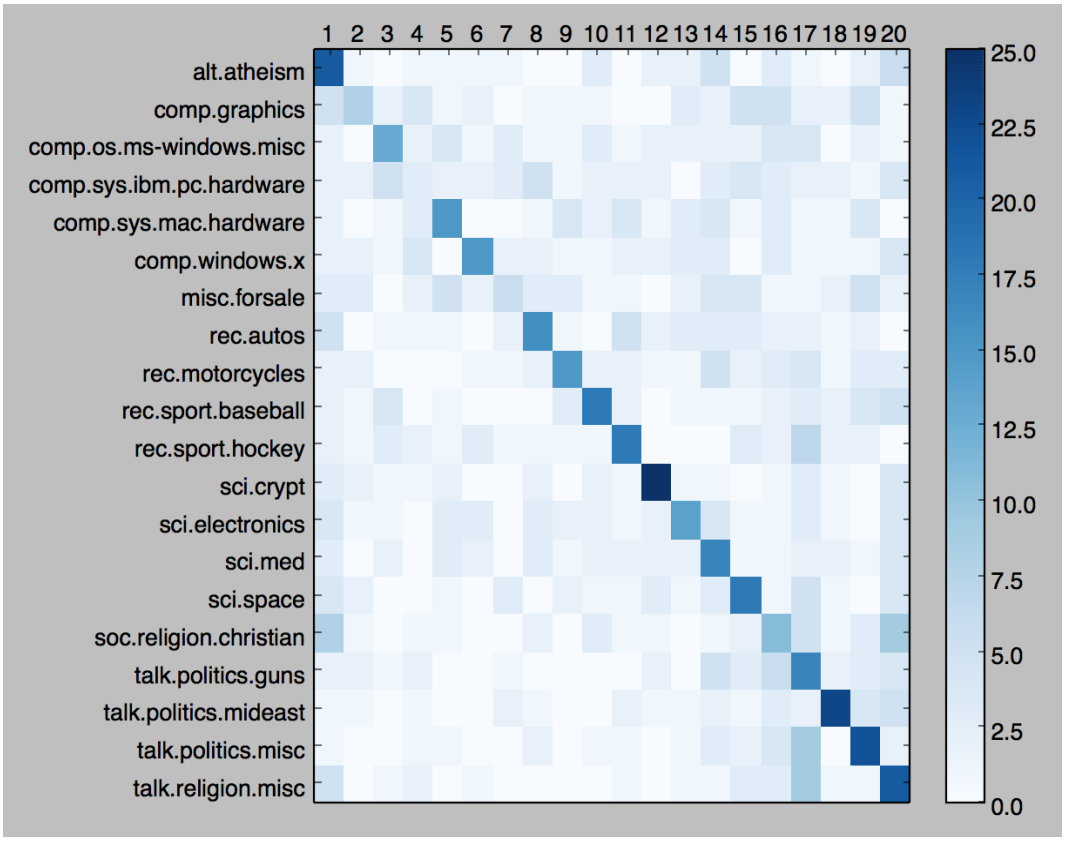
precision: 0.142

dimension = 25



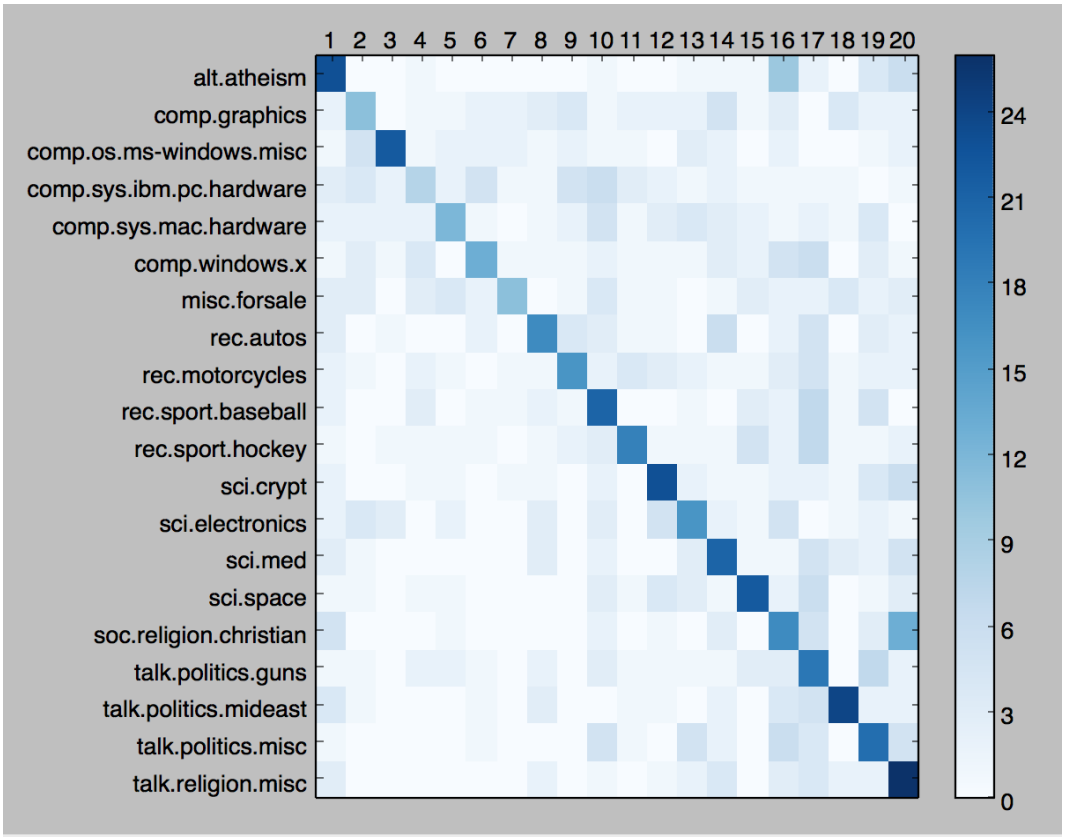
precision: 0.263

dimension = 50



precision: 0.316

dimension = 100



precision: 0.36

For which values of the target dimension are the results comparable to the original dataset?

When the dimension is 25, 50, and 100, we see that there is a more obvious higher similarity measure between articles of the same group (as indicated by the darker diagonal), which corresponds to the pattern seen in the baseline heat map.

With each increment in dimension, we see the precision value nearing the original dataset's precision value. Thus, of these four reduced dimension cases, dimension 100 has the closest precision value to the .456 that we calculated in our original dataset of much higher dimension.

(d) What is the time it takes to reduce the dimensionality of the data?

O(ndk) is the time it takes to reduce the dimensionality of the data. This is by way of the fact that reduction is the process of multiplying each datapoint by the M matrix. This matrix multiplication of a d x k with a k x 1, should take O(dk) time. Since we need to perform this reduction for all n datapoints (articles), we can see that reducing the dimensionality of the data would take n*O(dk), or O(ndk).

What is the overall Big-Oh runtime of classifying a new article, as a function of n (the number of labeled datapoints), k (the original dimension of each datapoint), and d (the reduced dimension)?

O(dk + 2nd). This is derived by the fact that O(dk) is the time it takes to reduce the new vector into d dimensions, as per the matrix multiplication of the d x k M matrix by the k x 1 new article representation. The next step in classification is comparing this vector of reduced dimensionality with every other article's vector, and calculating the cosine similarity between the two in order to find the one with the maximum similarity. Thus, for all n datapoints (articles), we need to calculate the cosine similarity, which, will take time n*O(2d), where O(2d) is the time for each singular cosine similarity calculation that consists of the multiplication of d counts with one another and O(d) time for the normalization of each article representation. Thus, for both stages of the classification, we arrive at a run time of O(dk) + O(2nd) = O(dk + 2nd)

Now suppose you are instead trying to classify tweets; the bag-of-words representation is still a k-dimensional vector, but now each tweet has, say, only 50 words. Explain how you could exploit the sparsity of the data to improve the runtime of the naive cosine-similarity nearest-neighbor classification system

O(n). Since each tweet has at most 50 words, we know that we will perform at most 50 multiplications when calculating cosine similarity, which results in constant time cosine similarity calculation. This is because we can take advantage of the extreme sparsity of the data by not needing to multiply any two words counts between two articles if at least one of them has a value of 0. Since the maximum number of non-zero pairs, under the constraint that each tweet has at most 50 words, between two tweets, can be 50. This is the case when both have 50 distinct words that happen to overlap perfectly with one another. Thus, at worst case, the time to calculate these 50 frequency multiplications in order to find the cosine similarity between two tweets, because of the fact that 50 is a constant, will be constant time, O(1). Thus, classifying a new tweet would take O(n) time if we are calculating its cosine similarity to n other tweets to find the one that is maximum. Therefore, if we classify all n tweets with all n other tweets, it would take $O(n^2)time$.

How does this runtime compare to that of a dimension-reduction nearest-neighbor system (as in the first step of this part) that reduces the dimension to d = 50?

O(50nk + 50k + 50n). Dimension reduction, on the other hand, takes O(nkd) time to reduce the data's dimensions and then O(dk + nd) time to classify a new article, resulting in a total time of O(nkd + dk + nd), where d = 50. Thus, exploiting the constant factor in the case of sparse tweets results in a much better runtime than the dimension reduction nearest neighbor system.

Part 3

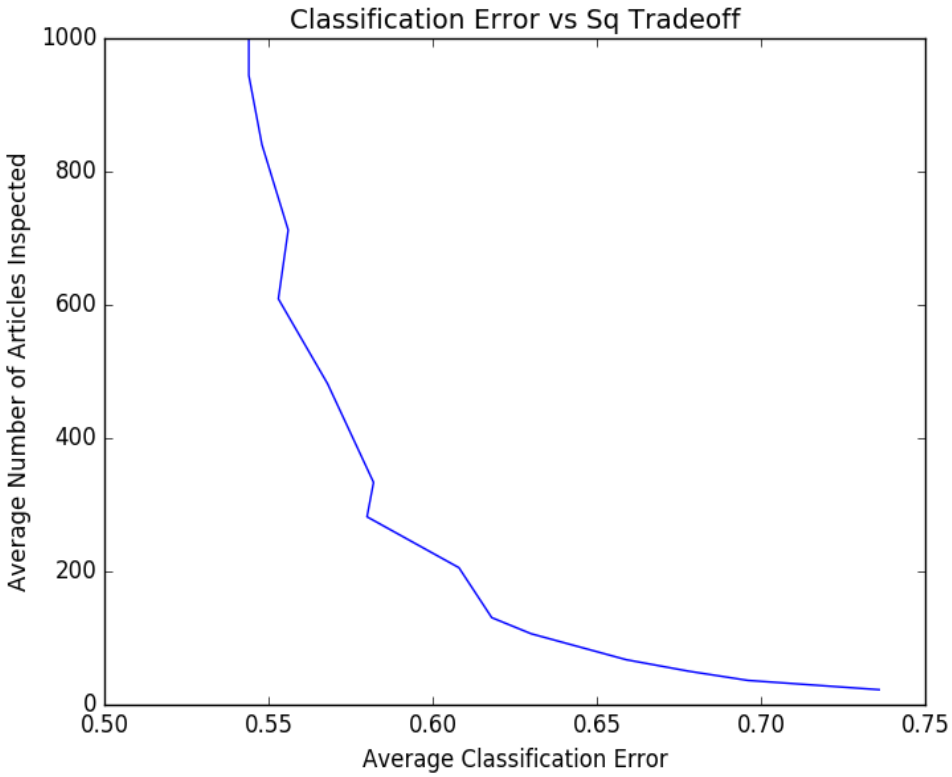
(a)

```
 import csv
import numpy as np
from scipy.sparse import csr_matrix
import math
import matplotlib.pyplot as plt
import warnings


#BASICALLY, CREATES 2 GLOBALS:
#original Matrix : original data without normalization, which we still use to hash
#normalized Matrix: which we only use to compute cosine similarity between 2
    articles
def read_data():
        reader = csv.reader(open('data/data50.csv', 'rb'))
        #matrix create here
        global originalMatrix
        global normalizedMatrix
        originalMatrix = np.zeros((1000, 61067))
        for row in reader:
                articleid = int(row[0]) - 1
                wordid = int(row[1]) - 1
                wordcount = int(row[2])

                originalMatrix[articleid][wordid] = wordcount
        newMatrix = np.zeros((1000, 61067))

        #NORMALIZE THE ORIGINAL MATRIX
        for articleId in range(originalMatrix.shape[0]):
                #divide each article by its norm
                article = originalMatrix[articleId]
                norm = np.linalg.norm(article)
                newArticle = article / float(norm)
```

```
                newMatrix[articleId] = newArticle
        # originalMatrix = newMatrix
        normalizedMatrix = newMatrix
        originalMatrix = csr_matrix(originalMatrix)




def createHashTables(dimension):
        global originalHash
        global M
        originalHash = [dict() for x in range(128)]
        M = []

        for i in range(128):
                randList = []
                for i in range(dimension):
                        rand = np.random.normal(0, 1.0, 61067)
                        randList.append(list(rand))
                M.append(np.array(randList))

        for articleId in range(originalMatrix.shape[0]):
                for x in range(128):
                        currentM = M[x]

                        currArticle = originalMatrix[articleId].toarray().T
                        newVector = np.dot(currentM,currArticle)

                        key = np.where(newVector > 0, 1, 0)
                        key = np.array_str(key)
                        if not key in originalHash[x]:
                                articleIdList = []
                                articleIdList.append(articleId)
                                originalHash[x][key] = articleIdList
                        else:
                                originalHash[x][key].append(articleId)
        print('finished creating hash tables')

def calculateCosineSimilarity(aWordVector, bWordVector):

        #already normalized, just need dot product
        similarity = np.dot(aWordVector, bWordVector.T)
```

```
        return similarity


def classification(dimension):
        global nearestNeighborCount
        nearestNeighborCount = np.zeros ((20,20))
        global SqTotal
        global precisionCount
        precisionCount = 0.0
        SqTotal = 0.0




        for articleId in range(originalMatrix.shape[0]):
                Sq = set()
                for x in range(128):
                        currentM = M[x]
                        currArticle = originalMatrix[articleId].toarray().T
                        newVector = np.dot(currentM,currArticle)
                        key = np.where(newVector > 0, 1, 0)
                        key = np.array_str(key)
                        existingValues = originalHash[x][key]
                        # print('collidingvalues: ', existingValues, 'for hash table: ',
                            x, ' corresponding to key: ' , key, 'and article id: ',
                            articleId)
                        for num in existingValues:
                                if(num != articleId):
                                        Sq.add(num)

                # print('currArticle:', articleId)
                # print('sq: ', Sq)
                SqTotal += len(Sq)


                maxSimilarity = 0.0
                maxArticle = None
                for article in Sq:
                        #cosine similarity
                        similarity = calculateCosineSimilarity(normalizedMatrix[
                            article], normalizedMatrix[articleId])
                        if(similarity > maxSimilarity):
```

```
                                maxArticle = article
                                maxSimilarity = similarity

                        #found max by now
                        if (maxArticle != None):
                                if(maxArticle/50 == articleId/50):
                                        precisionCount += 1
                                nearestNeighborCount[maxArticle/50, articleId/50] += 1
        averagePrecision = precisionCount / float(1000.0)
        averageError = 1.0 - averagePrecision
        averageSqSize = SqTotal / float(1000.0)
        print ('average_error_for_dimension', dimension, ': ', averageError)
        print ('average_error_for_dimension_using_function', dimension, ': ',
            averageClassificationError(nearestNeighborCount))
        print('average_sq_size_for_dimension', dimension,': ', averageSqSize)



def averageClassificationError(m):
        averageErrors = 0
        for i in range(m.shape[0]):
                averageErrors += m[i,i]
        return (1.0 - averageErrors/float(1000))


read_data()
for i in range(16):
        dimension = 5 + i
        createHashTables(dimension)
        classification(dimension)
```



Classification error increases relatively slowly from d = 5 to d = 13, with an error of 0.54 at d = 5 to an error of 0.58 at d = 13. The average number of articles inspected, however, decreases drastically during this time, from 998 articles at d = 5 to 282 articles at d = 13. At d = 14, we have an error of 0.60, which is a drastic increase in classification error compared to the previous rate at which it was increasing. Thus, when the dimension is around 13 seems to be the sweet spot in looking at the trade off between average size of Sq and classification error. A higher dimension would offer a more drastic delta in classification error despite only a moderate delta in average number of articles inspected. Any higher dimension or classification error at that point, though it would decrease running time of classification further, would also offer a more steep increase in our classification error–a tradeoff that does not seem worth the inaccuracy because the size of Sq at that level of dimension would not be changing too drastically and thus runtime would not be decreasing fast enough to justify the increased error we would take on.Meaning, because the slope of the line becomes noticeably flatter after d = 12, we maintain the optimal at d=13.

(b) compare/contrast performance properties

Though LSH requires a longer processing time, by way of the fact that we need to reduce the dimensions of the articles separately for each hashtable, its benefits over the dimension reduction method begin to show for large datasets. In the case of large datasets, LSH is able to perform faster classification because it will only perform cosine

similarity calculations on datapoints that collide into at least one of the hashtable slots of the article at hand...rather than a brute force cosine similarity calculation between the article at hand and every other article in the repository.

The performance of LSH and dimension reduction also varies with the value of d; LSH is more accurate when d is smaller whereas dimension reduction is more accurate with larger values of d. This is because the more we compress the vectors in LSH, the more collisions occur, which results in more comparisons with other datapoints - giving us a more accurate similarity measure. Dimension reduction, on the other hand, compares each datapoint to every other datapoint, resulting in a higher degree of accuracy with less compression.

What properties of an application would suggest that the former would be better choice than the latter, or vice versa? Larger datasets would benefit from a LSH-based nearest-neighbor classification system, for reasons described above, that show the performance advantage LSH tends to provide in the case large datasets with a reasonably low d.

Describe how you might combine the two approaches. First, reduce the vectors and then hash them using the LSH scheme to create a combined approach between dimensionality reduction and LSH.This could help speed up the brute-force calculation in the LSH classification system because once the article vectors have been reduced, finding the hash spot for all l hash functions can be done with a singular matrix multiplication between a matrix where each row consists of an M hashing vector for the ith hashtable. Thus, instead of having to multiply each of the l of M (as a d x k matrix) matrices with the article representation, we can multiple a single matrix of size l x d by the article representation to retrieve all l hash slots at once.

A combination might outperform both of the single-approach systems if we are dealing with a large dataset that has datapoints of high dimension. This is because we save a lot of computation time if we reduce all the datapoints prior to hashing, so when we actually hash the datapoints, we eliminate the computations of multiplying a matrix M by every datapoint vector for every hashtable; rather, we can just multiply our reduced vector by a vector of dimension d (as explained above). This results in a tradeoff of accuracy and speed; combining the approaches will give us a lower accuracy but a faster processing time, so it is ideal in situations where we are willing to trade accuracy for speed when it comes to larger datasets of high dimension datapoints.