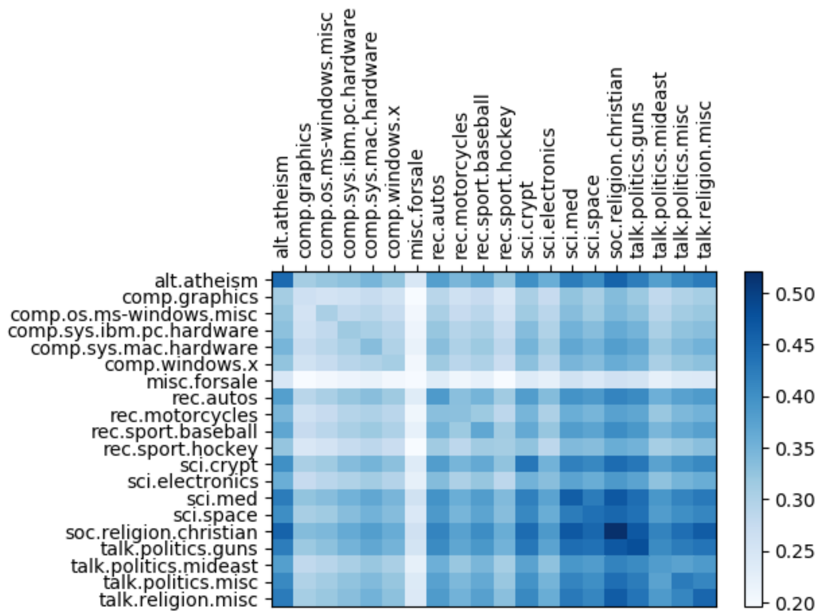
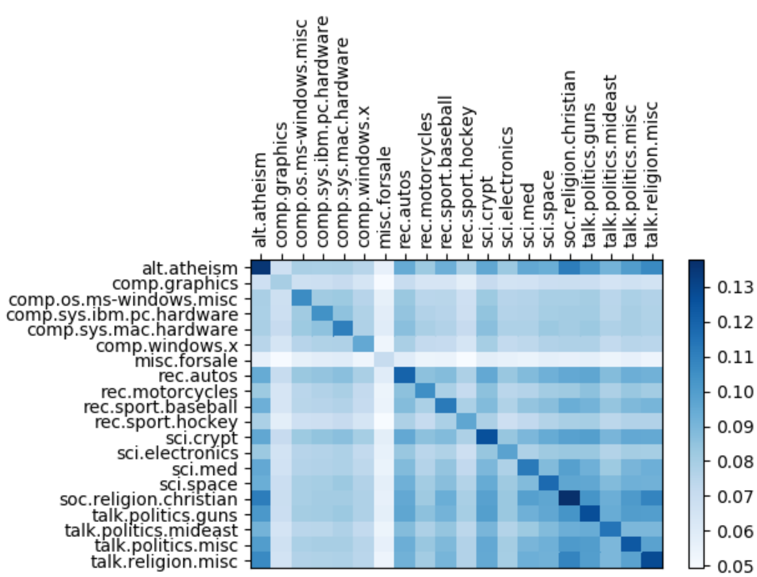
CS168 Project 2
Henry Lin, Kaylee Bement

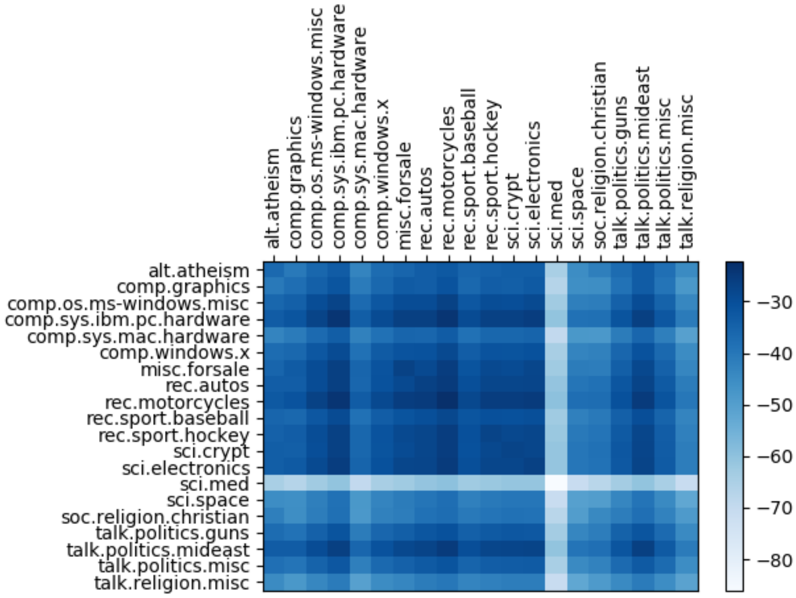
1. A. See code.py
   B. See code.py
   COSINE



JACCARD



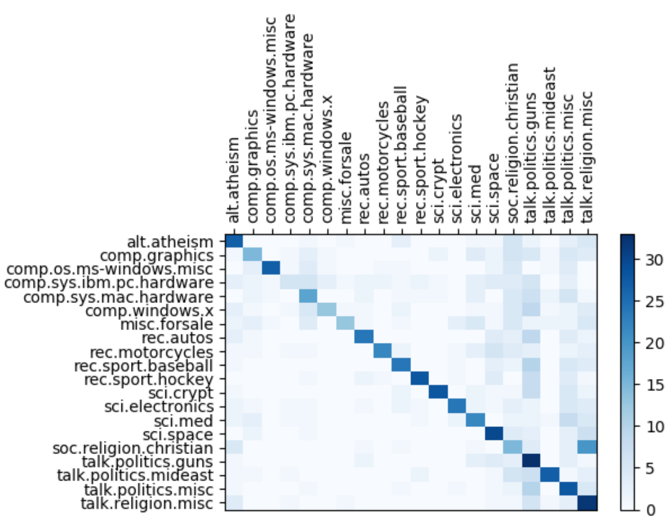
L2 SIMILARITY



C. Based on the heat maps, the Jaccard and cosine similarity metrics seem to both perform reasonably well for this dataset. Since any bag of words model is bound to be sparse by design, it is expected that Jaccard would perform well in comparing the articles of each newsgroup.

The atheism newsgroup seems to share a noticeable similarity with the religion Christian newsgroup. While the two groups may have differing opinions, it is expected that they are rated similar since they discuss the same topics. There is a surprising amount of similarity between science-related and religion-related newsgroups such as religion Christian and medicine, but the two groups definitely have an intersection where there is a significant level of debate, both in the dataset and real life.

2. A. See code.py

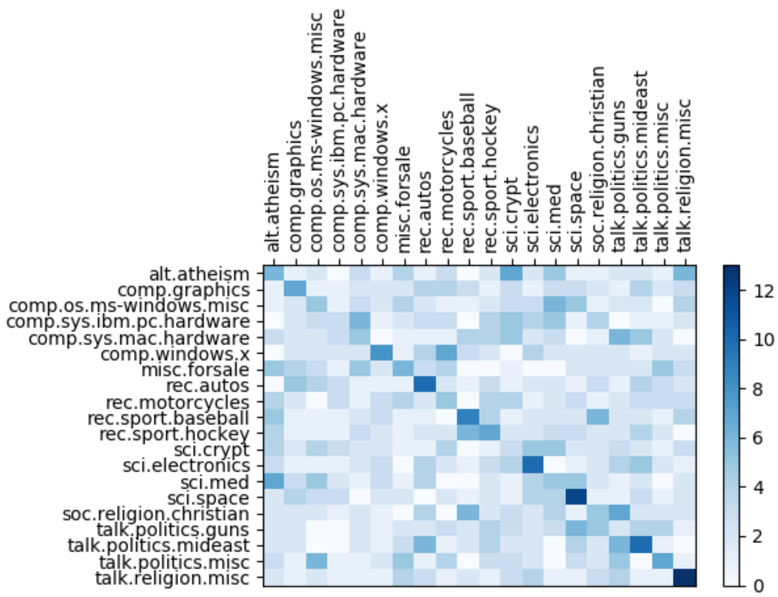


Classification Error: 0.544

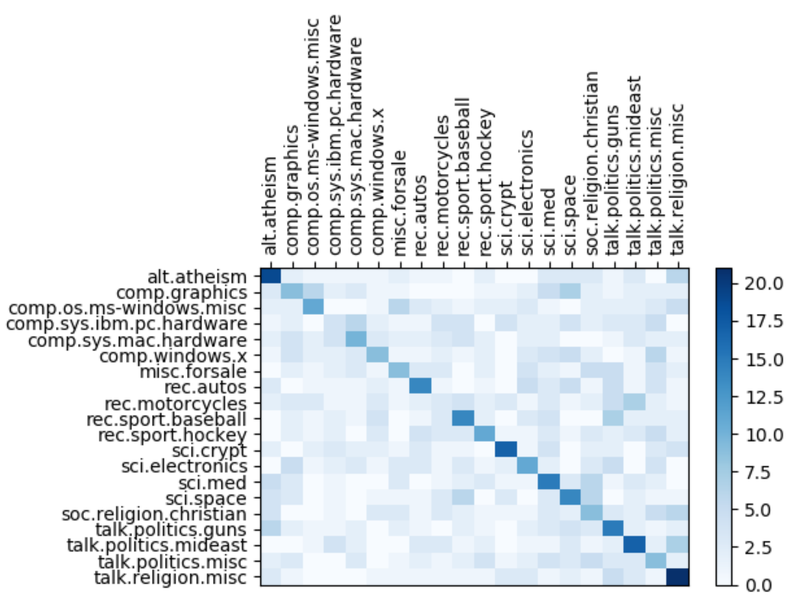
B. For Part 1, the plots dealt with distances between points, and distance is symmetric. The distance from A to B is equal to the distance from B to A. For this matrix, we are dealing with closest neighbors, which is not necessarily a symmetric property. For example, AB = 4, BC = 3, AC= 10. In this case, B would be A's closest neighbor, but B's closest neighbor is actually C. This causes the asymmetric matrix.

C.

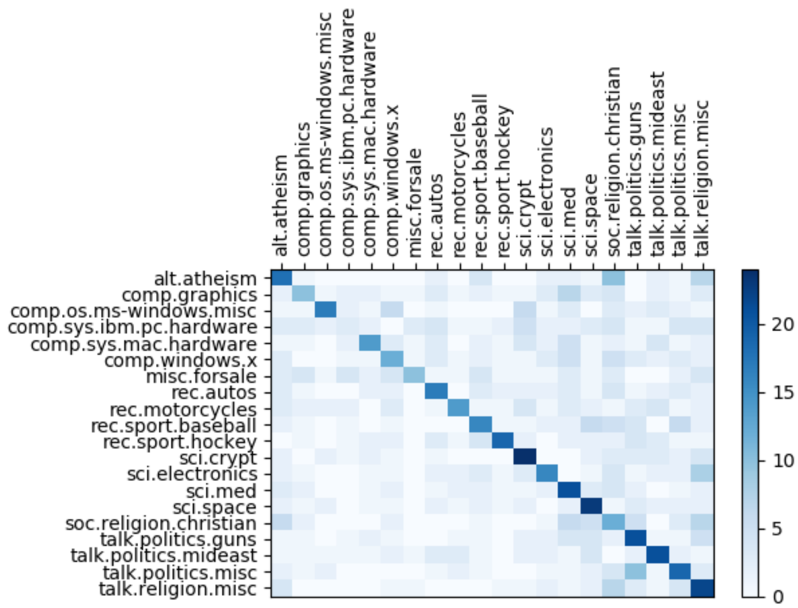
Classification Error, d = 10: 0.88



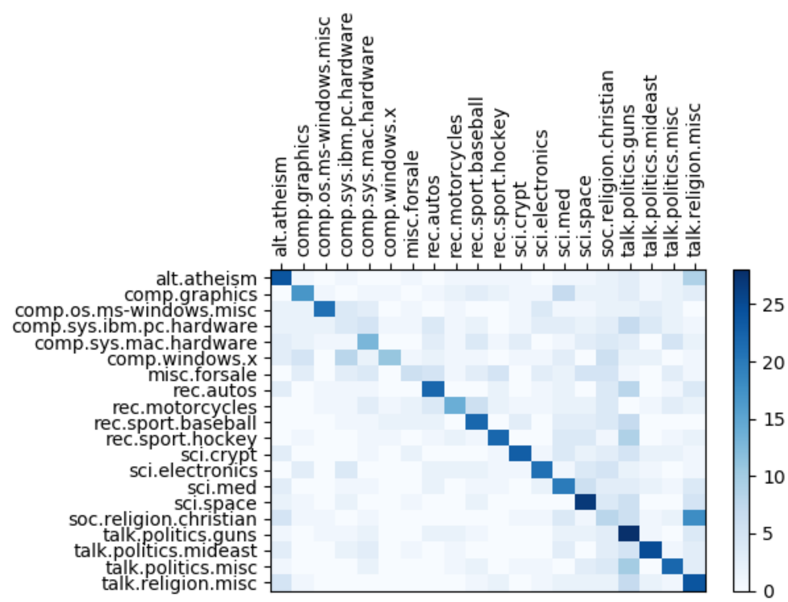
Classification Error, d = 25: 0.771



Classification Error, d = 50: 0.702



Classification Error, d = 100: 0.645



Note: Due to the randomness in the model, these values change slightly with each run of the program.

It is clear that as the target dimension increases, the classification error decreases. For the given values of d, 100 gives the most comparable results to the original dataset.

D.

The runtime in big O for reducing the dimension of the data will be O(dk + dnk), which reduces to O(dnk). dk is required to set up the matrix and the dnk is the runtime of the matrix multiplication.

The runtime of classifying a new article is O(dk + nd) since we already have the d by n matrix we can now use and we also need to do dimensionality reduction.
Since the data is sparse with each tweet containing a relatively small number of words, we can use a dictionary or some other efficient data structure to improve the runtime to O(50n) or O(n).
Comparing this to the dimension reduction nearest-neighbor system, the two have comparable search time with the dimension reduction system being more costly when including the original matrix multiplication.

3. A.
From lecture, we know that choosing each coordinate of the hash vector from a normal distribution creates a vector in a uniformly random direction. For one dimension, the normal vector to the random projection must go through the angle θ to hash x and y to different buckets, so the probability of them hashing to the same bucket is the probability that the normal vector does not go through θ. We then raise this to the power of d for the d dimensions that must match.

$$P = (1 - \frac{\theta}{\pi})^d$$

B.
Following from the previous question, we can simply extend this to consider the new range of angles with an upper bound.

$$P = \left(1 - \frac{2\theta}{\pi}\right)^d$$

$$As\ 2\theta\ nears\ \ \pi, the\ probability\ of\ collision\ goes\ towards\ 0.$$

C.
To solve this problem, we have to consider equations that model the requirements. We first consider than when the angle between the query and a datapoint is less than 0.1 radians, we want the probability that the query hashes to the same bucket in at least one of the tables to be at least 0.9

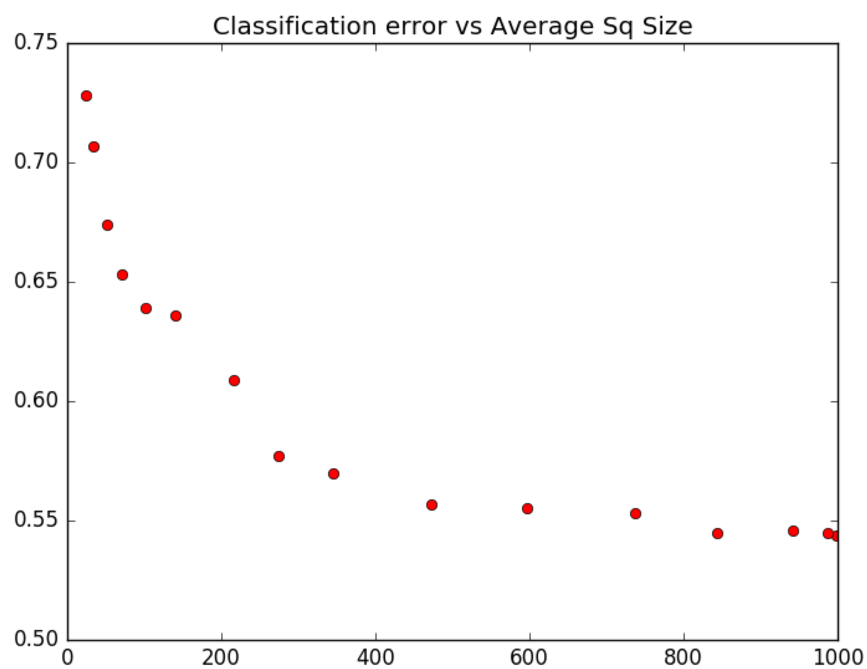
$$0.9 \leq 1 - (1 - (1 - \frac{0.1}{\pi})^d)^l$$

Secondly, we want to make sure that for the query and any datapoint with angle greater than 0.2, we want there to be very few collisions. In the equation below, we want there to be at most 60,000 such collisions.

$$60{,}000 \geq \sum_{i=1}^{1{,}000{,}000} 1 - (1 - (1 - \frac{0.2}{\pi})^d)^l$$

Solving this system of inequalities, we get a range of possible values for l and d that satisfy our threshold, and we decided to use d = 114 and l = 98. These values can change depending on where importance is placed. For instance, decreasing l will decrease greatly the computational cost and expected number of collisions overall, but it will also decrease the probability of finding the nearest neighbor (angle-wise) in the resulting set of points. For d, decreasing it will increase the chances of the nearest neighbor appearing in the set we consider, but it will also noticeably increase the number of collisions and consequently the resulting set.

D. See code.py



d	**Classification Error**	**Average Size of Sq**
5	0.544	998.224
6	0.545	987.548
7	0.546	941.53
8	0.545	843.87
9	0.553	736.49
10	0.555	596.468
11	0.557	472.522
12	0.570	344.504
13	0.577	274.814
14	0.609	216.744
15	0.636	139.622
16	0.639	101.702
17	0.653	70.324
18	0.674	51.522

19	0.707	33.544
20	0.728	24.55

The sweet spot seems to be when d = 13, the classification error is 0.577, and the average size of Sq is 274.814. This is only an increase of about 0.03 from the lowest classification error we received, but the average size of Sq is about 700 articles smaller than the largest average size. The next classification error is 0.609, about 0.03 higher than that for d = 13, for only a decrease of 60 articles for the average Sq size.

E.

The LSH-based classification system increases in error as d increases, while the dimension-reduction-based one decreases in error as d increases. Also, LSH-based classification’s runtime increases as d decreases since Sq will be bigger, while dimension-reduction’s runtime decreases as d decreases. However, it is important to note that the scale for LSH-based classification was d = [5, 20] while for dimension-reduction it was d = [10, 100]. LSH is better when there are a lot of data points, while dimensionality reduction is better when there are a large number of dimensions.

To combine the approaches, one would use LSH first and then dimensionality reduction to speed up the classification system of LSH. This would outperform the single-approach systems when there are a lot of data points with a large number of dimensions because LSH can first partition the data points, and dimensionality reduction can then make it quicker to search through the partitions to classify data.

CODE

```
import csv
from functools import reduce
import math
import matplotlib.pyplot as plt
import numpy as np
import warnings
import scipy.spatial as sp

# QUESTION 1

# groups - groups[i] is name of group i + 1
groups = []
with open('p2_data/groups.csv', 'rt') as csvfile:
   reader = csv.reader(csvfile)
   for row in reader:
      groups.append(row[0])

# labels - labels[i] has list of all articles (0 indexed instead of 1 indexed) in group i + 1
```

```
# labels_reversed[i] gives group id for article i + 1
labels = []
labels_reversed = []
curr_group = 1
with open('p2_data/label.csv', 'rt') as csvfile:
    reader = csv.reader(csvfile)
    articles = []
    index = 0
    for row in reader:
        group_id = int(row[0])
        if group_id != curr_group:
            labels.append(list(articles))
            articles = []
            curr_group = group_id
        articles.append(index)
        index += 1
        labels_reversed.append(group_id)
    labels.append(list(articles))

# articles - articles[i] has dict of (wordId, wordCount) pairs for articleId i + 1
articles = []
curr_article = 1
max_word_id = -1
with open('p2_data/data50.csv', 'rt') as csvfile:
    reader = csv.reader(csvfile)
    word_counts = {}
    for row in reader:
        article_id = int(row[0])
        if article_id != curr_article:
            articles.append(dict(word_counts))
            word_counts = {}
            curr_article = article_id
        word_id = int(row[1])
        if word_id > max_word_id:
            max_word_id = word_id
        count = row[2]
        word_counts[word_id] = int(count)
    articles.append(dict(word_counts))


def jaccard(x, y, word_ids):
    numerator = 0
    denominator = 0
    for word in word_ids:
```

```
        x_count = x.get(word, 0)
        y_count = y.get(word, 0)
        numerator += min(x_count, y_count)
        denominator += max(x_count,y_count)
    return float(numerator)/denominator

def l2sim(x, y, word_ids):
    similarity = 0
    for word in word_ids:
        x_count = x.get(word, 0)
        y_count = y.get(word, 0)
        similarity += math.pow(x_count - y_count, 2)
    similarity = math.sqrt(similarity)
    return -similarity

def cosine(x, y, word_ids):
    numerator = 0
    x_term = 0
    y_term = 0
    for word in word_ids:
        x_count = x.get(word, 0)
        y_count = y.get(word, 0)
        numerator += x_count * y_count
        x_term += math.pow(x_count, 2)
        y_term += math.pow(y_count, 2)
    x_term = math.sqrt(x_term)
    y_term = math.sqrt(y_term)
    return float(numerator)/(x_term * y_term)

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(names,rotation=90)
    ax.set_yticklabels(names)

    plt.tight_layout()

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

def findSimilarity(func, filename):
  num_groups = len(groups)
  matrix = np.zeros((num_groups, num_groups))
  for i in range(num_groups):
    group1 = labels[i]
    for j in range(num_groups):
      group2 = labels[j]
      num_similarities = 0
      total_similarities = 0
      for article_id1 in group1:
        for article_id2 in group2:
          x = articles[article_id1]
          y = articles[article_id2]
          word_ids = reduce(set.union, map(set, map(dict.keys, [x, y])))
          total_similarities += func(x, y, word_ids)
          num_similarities += 1
      matrix[i][j] = float(total_similarities)/num_similarities
  makeHeatMap(matrix, groups, plt.cm.Blues, filename)

print("Jaccard Similarity...")
findSimilarity(jaccard, "jaccard.png")
print("Jaccard Done")
print("L2 Similarity...")
findSimilarity(l2sim, "l2sim.png")
print("L2 Done")
print("Cosine Similarity...")
findSimilarity(cosine, "cosine.png")
print("Cosine Done")

# QUESTION 2

num_articles = len(labels_reversed)
```

```
num_groups = len(groups)

def baseline():
    matrix = np.zeros((num_groups, num_groups))
    error_count = 0
    for i in range(num_articles):
        group_id = labels_reversed[i]
        best_similarity = float("-inf")
        nearest_article_group = -1
        for j in range(num_articles):
            if i == j:
                continue
            x = articles[i]
            y = articles[j]
            word_ids = reduce(set.union, map(set, map(dict.keys, [x, y])))
            similarity = cosine(x, y, word_ids)
            if similarity > best_similarity:
                best_similarity = similarity
                nearest_article_group = labels_reversed[j]
        matrix[group_id - 1][nearest_article_group - 1] += 1
        if group_id != nearest_article_group:
            error_count += 1

    print("Classification Error") # using error from piazza, not pset
    classification_error = float(error_count)/num_articles
    print(classification_error)
    makeHeatMap(matrix, groups, plt.cm.Blues, "baseline.png")

print("Baseline...")
baseline()
print("Baseline Done")

def projection(d):
    new_articles = []
    matrix = np.random.normal(size = (d, max_word_id))
    for article in articles:
        full_vector = np.zeros(max_word_id,)
        for k, v in article.items():
            full_vector[k - 1] = v
        new_articles.append(np.inner(full_vector, matrix))
    return new_articles

def nearest_neighbor(projection, filename):
    matrix = np.zeros((num_groups, num_groups))
```

```
    error_count = 0
    for i in range(num_articles):
        group_id = labels_reversed[i]
        best_similarity = float("-inf")
        nearest_article_group = -1 # what if more than one nearest article?
        for j in range(num_articles):
            if i == j:
                continue
            x = projection[i]
            y = projection[j]
            similarity = cosine_arr(x, y)
            if similarity > best_similarity:
                best_similarity = similarity
                nearest_article_group = labels_reversed[j]
        matrix[group_id - 1][nearest_article_group - 1] += 1
        if group_id != nearest_article_group:
            error_count += 1

    print("Classification Error")
    classification_error = float(error_count)/num_articles
    print(classification_error)

    makeHeatMap(matrix, groups, plt.cm.Blues, filename)

def cosine_arr(x, y):
    numerator = 0
    x_term = 0
    y_term = 0
    for i in range(len(x)):
        x_count = x[i]
        y_count = y[i]
        numerator += x_count * y_count
        x_term += math.pow(x_count, 2)
        y_term += math.pow(y_count, 2)
    x_term = math.sqrt(x_term)
    y_term = math.sqrt(y_term)
    return float(numerator)/(x_term * y_term)

print("Projecting w d = 10...")
projection1 = projection(10)
print("Finding nearest neighbors...")
nearest_neighbor(projection1, "projection1.png")
print("Projecting w d = 25...")
projection2 = projection(25)
```

```
print("Finding nearest neigbors...")
nearest_neighbor(projection2, "projection2.png")
print("Projecting w d = 50...")
projection3 = projection(50)
print("Finding nearest neigbors...")
nearest_neighbor(projection3, "projection3.png")
print("Projecting w d = 100...")
projection4 = projection(100)
print("Finding nearest neighbors...")
nearest_neighbor(projection4, "projection4.png")


# QUESTION 3
l = 128

def create_matrices(d):
    matrices = []
    for table in range(l):
        matrix = np.random.normal(size = (d, max_word_id))
        matrices.append(matrix)
    print("Shape of matrix")
    print(matrices[0].shape)
    return matrices

def hyperplane_hashing(matrices):
    new_articles = []
    for article in articles:
        new_articles.append(hash_article(article, matrices))
    return new_articles

def hash_article(article, matrices):
    full_vector = np.zeros(max_word_id,)
    new_article = []
    for k, v in article.items():
        full_vector[k - 1] = v
    for i in range(l):
        hashvalue_full = np.inner(full_vector, matrices[i])
        hashvalue_i = [1 if i > 0 else 0 for i in hashvalue_full]
        new_article.append(hashvalue_i)
    return new_article

def classification(q, matrices, new_articles):
    hashvalues = hash_article(q, matrices)
    best_similarity = float("-inf")
```

```
    best_group = -1
    sq_size = 0
    for i in range(num_articles):
        group_id = labels_reversed[i]
        datapoint = new_articles[i]
        article = articles[i]
        if datapoint == hashvalues:
            continue
        for j in range(l):
            if hashvalues[j] == datapoint[j]:
                word_ids = reduce(set.union, map(set, map(dict.keys, [q, article])))
                similarity = cosine(q, article, word_ids)
                if similarity > best_similarity:
                    best_similarity = similarity
                    best_group = group_id
                sq_size += 1
                break
    return (best_group, sq_size)


def lsh(d):
    combined_sq_size = 0
    matrices = create_matrices(d)
    new_articles = hyperplane_hashing(matrices)
    error_count = 0
    for i in range(num_articles):
        actual_group = labels_reversed[i]
        suggested_group, sq_size = classification(articles[i], matrices, new_articles)
        combined_sq_size += sq_size
        if actual_group != suggested_group:
            error_count += 1
    print("Classification Error")
    classification_error = float(error_count)/num_articles
    print(classification_error)
    print("Average Size of Sq")
    sq_size = float(combined_sq_size)/num_articles
    print(sq_size)
    return (classification_error, sq_size)


def makePlot(classificationErrors, averageSqSizes, outputFileName):
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        plt.title("Classification error vs Average Sq Size")
```

```
        plt.axis([0, 1000, 0.5, 0.75])
        plt.plot(averageSqSizes, classificationErrors, 'ro')
        plt.savefig(outputFileName, format = 'png')
        plt.close()

classificationErrors = []
averageSqSizes = []
for d in range(5, 21):
    print("LSH for d value " + str(d))
    classification_error, sq_size = lsh(d)
    classificationErrors.append(classification_error)
    averageSqSizes.append(sq_size)
makePlot(classificationErrors, averageSqSizes, "lsh.png")
```