Part 1: Similarity Metrics

- three types of similarity metrics:

Jaccard Similarity: $J(x, y) = \frac{\sum_i min(x_i, y_i)}{\sum_i max(x_i, y_i)}$

L_2 Similarity: $L_2(x, y) = -\|x - y\|_2 = -\sqrt{\sum_i (x_i - y_i)^2}$

Cosine Similarity: $S_C(x, y) = \frac{\sum_i x_i \cdot y_i}{\|x\|_2 \cdot \|y\|_2}$

(a) import the data from data50.csv, groups.csv and label.csv

In [1]:

```
import numpy as np
import csv
import time
import sys
from makeHeatMap import *

DATA = []
with open('./p2_data/data50.csv') as data:
    data50_csv = csv.reader(data)
    DATA50 = list(data50_csv)
    for i in range(len(DATA50)):
        article_number, word, word_frequency = int(DATA50[i][0]), int(DATA5
        if article_number > len(DATA):
            DATA.append([])
        DATA[article_number-1].append([word, word_frequency])

LABELS = []
with open('./p2_data/label.csv') as data:
    labels_csv = csv.reader(data)
    LABELS = list(labels_csv)
    for i in range(len(LABELS)):

        LABELS[i] = int(LABELS[i][0]) -1

GROUPS = []
with open('./p2_data/groups.csv') as data:
    groups_csv = csv.reader(data)
    GROUPS = list(groups_csv)

    GROUPS = np.array(GROUPS).reshape(-1)


WORD_SIZE = 62000
DATA_SIZE = len(DATA)
CLASS_SIZE = max(LABELS) + 1
BATCH_SIZE = 50

np.random.seed(0)
```

In [2]:

```
print(len(DATA))
print(len(LABELS))
print(CLASS_SIZE)
```

```
1000
1000
20
```

(b) Implement the three similarity metrics described above, and plot three rainbow colormap(20*20) about the similarity between each two groups.

In [2]:

```
def readArticle(articleId, KDMATRIX = None):
    if type(articleId) == int:
        articleWords = np.zeros(WORD_SIZE)
        for word_index, word_frequency in DATA[articleId]:
            articleWords[word_index] = word_frequency
    elif type(articleId) == list or type(articleId) == np.ndarray:
        articleWords = np.zeros((len(articleId),WORD_SIZE))
        for i in range(len(articleId)):
            for word_index, word_frequency in DATA[articleId[i]]:
                articleWords[i][word_index] = word_frequency
    else:
        raise RuntimeError('Invalid articleId')

    if type(KDMATRIX) != type(None):
        articleWords = articleWords @ KDMATRIX
    return articleWords

print(readArticle(0)[10])
print(np.array(readArticle(0)).shape)
print(readArticle([0,1])[0][10])
print(np.array(readArticle([0,1])).shape)
KDMATRIX = np.zeros((62000,10))
print(np.array(readArticle([0,1], KDMATRIX)))
```

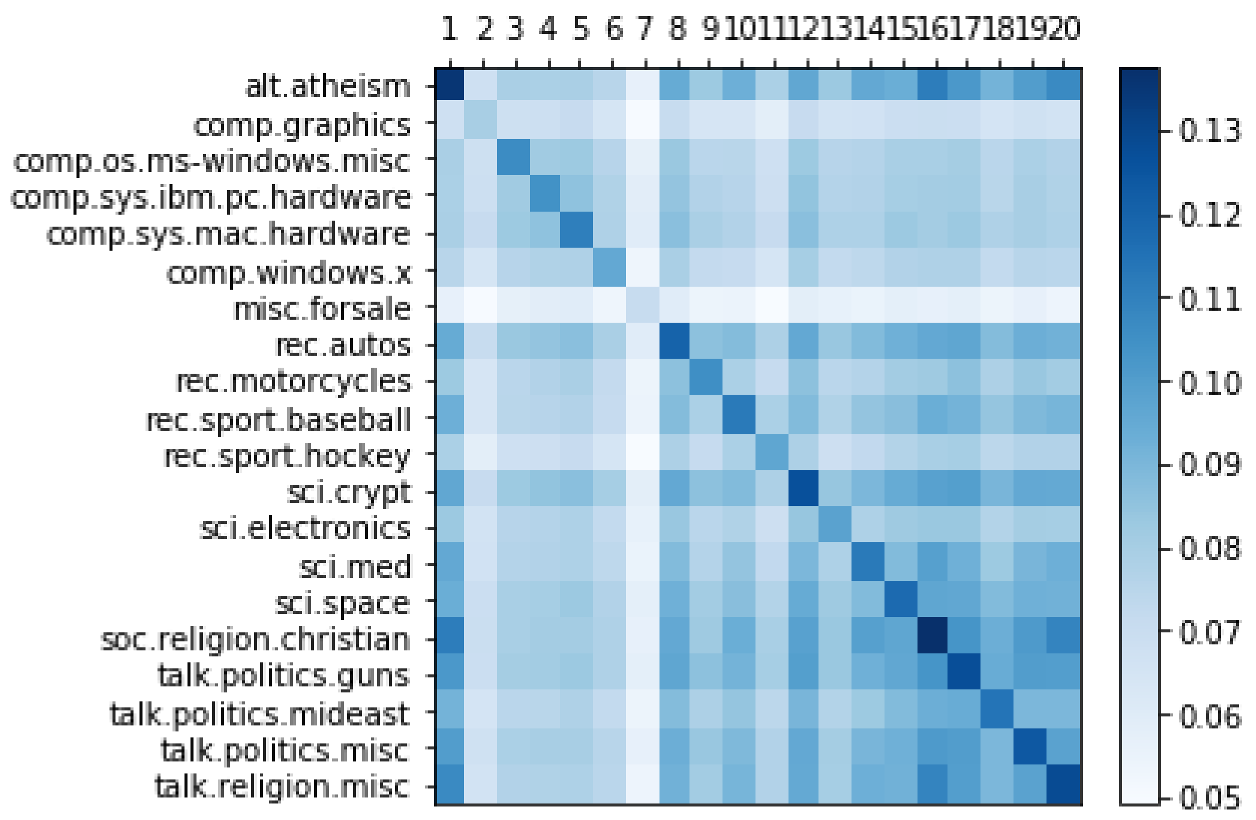
```
1.0
(62000,)
1.0
(2, 62000)
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

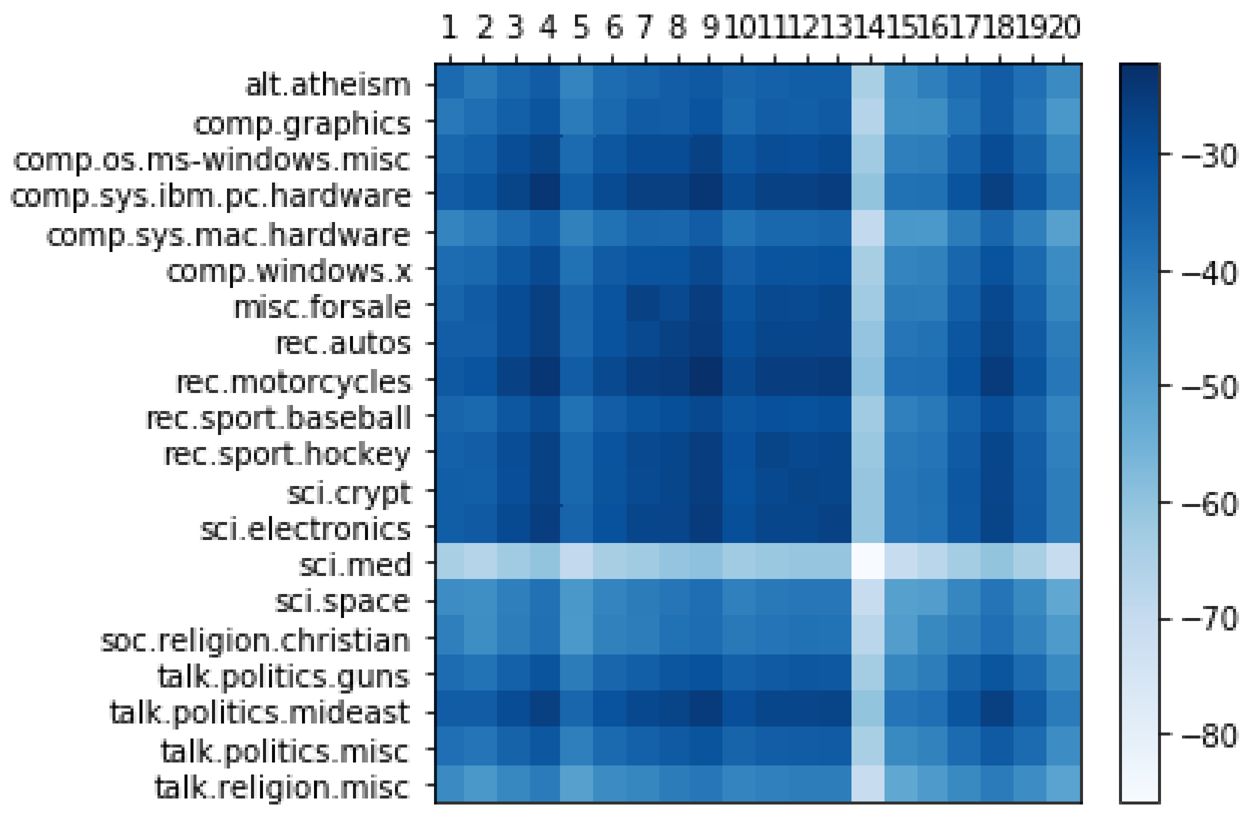
In [3]:

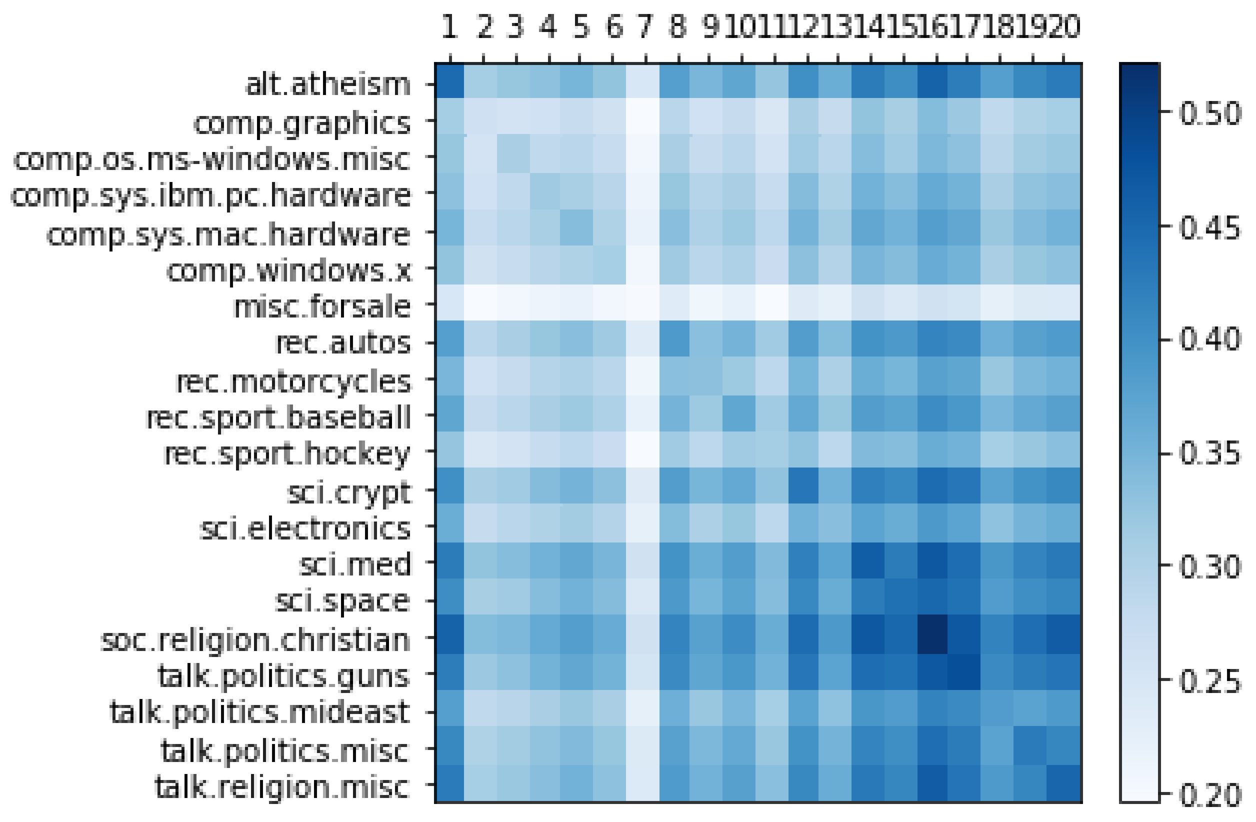
```
def getGroupList(label_index):
    return list(range(label_index * BATCH_SIZE, label_index * BATCH_SIZE + 

  对比两个ˆ b   E Mó & ß MÓ
def compareGroup(groupLabel_x, groupLabel_y, Metric):
    similarity = 0

    group_x_articles = readArticle(getGroupList(groupLabel_x))
    group_y_articles = readArticle(getGroupList(groupLabel_y))

    for article_x in group_x_articles:
        for article_y in group_y_articles:
            similarity += Metric(article_x, article_y)

    return similarity / (BATCH_SIZE * BATCH_SIZE)


def JaccardSimilarity(x, y):
    x = np.reshape(x, (-1, 1))
    y = np.reshape(y, (-1, 1))
    combine = np.concatenate([x,y], axis = 1)
    min_count = np.sum(np.min(combine, axis = 1))
    max_count = np.sum(np.max(combine, axis = 1))
    return min_count / max_count


def L2Similarity(x, y):
    difference = x - y
    return -np.sqrt(np.sum(difference ** 2))


def CosineSimilarity(x, y):
    upper = np.sum(x * y)
    norm_x = np.sqrt(np.sum(x**2))
    norm_y = np.sqrt(np.sum(y**2))
    return upper / (norm_x * norm_y)

  / ½- é
def getMetricMatrix(Metric):
    metric_matrix = np.zeros((CLASS_SIZE,CLASS_SIZE), dtype = np.float64)
    for i in range(CLASS_SIZE):
        for j in range(i, CLASS_SIZE):
            metric_matrix[j][i] = metric_matrix[i][j] = compareGroup(i, j, 
    return metric_matrix
```

In [5]:

```
makeHeatMap(getMetricMatrix(JaccardSimilarity), GROUPS, plt.cm.Blues)
makeHeatMap(getMetricMatrix(L2Similarity), GROUPS, plt.cm.Blues)
makeHeatMap(getMetricMatrix(CosineSimilarity), GROUPS, plt.cm.Blues)
```







(c) Based on the three heatmaps:

c-1. decide which of the similarity metrics seems the most reasonable, and why?

1. The JaccardSimilarity may be the best one, because the similarities bewteen different newsgroups are far more lower than the similarities bewteen the same newsgroups, this can help us to classificate articles into different newsgroups.

c-2. Are there any pairs of newsgroups that are very similar? Would you have expected these to be similar?

2. `soc.religion.christian` , `talk.religion.misc` and `alt.atheism` are similar. It is not strange to find this because these three newsgroups are all about religion.

Part 2: Dimension Reduction

(a)

a-1. Implement the baseline cosine-similarity nearest-neighbor classification system that, for any given document, finds the document with largest cosine similarity, and returns that newsgroup/label. (Do each computation using brute-force search.)

In [4]:

```
  获取    在       中的最近邻居索引（除了自己）
def nearestNeighbour(vec, vecSet, current_index, Metric):
    most_similar = -sys.maxsize
    nearest_index = 0
    ind = 0
    for v in vecSet:
          跳过自己
        if current_index != ind:
            similarity = Metric(vec, v)
            if similarity > most_similar:
                nearest_index = ind
                most_similar = similarity
        ind += 1

    return nearest_index, most_similar
```

a-2. Compute the 20 × 20 matrix whose entry (A, B) is defined by the number of articles in group A that have their nearest neighbor in group B. Plot these results in a heatmap

In [5]:

```
   获取 错入 的错误率
def getError(groupLabel_x, groupLabel_y, Metric, d, KDMATRIX):
    error = 0

      获取文章的单词向量并降6
    reduce_time_start = time.clock()
    group_x_articles = readArticle(getGroupList(groupLabel_x), KDMATRIX)
    group_y_articles = readArticle(getGroupList(groupLabel_y), KDMATRIX)
    reduce_time_end = time.clock()

    article_both = np.concatenate([group_x_articles, group_y_articles])
      计算错误率
    current_index = 0
    for article_x in article_both:
        nearest_index, _ = nearestNeighbour(article_x, article_both, curren
        if (current_index<50 and nearest_index>=50):
            error += 1
        current_index += 1
        if (current_index >= 50):
            break

    return error, reduce_time_end - reduce_time_start

   获取错误率矩阵
def getErrorMatrix(Metric, d):
    KDMATRIX = np.random.normal(0, 1, (WORD_SIZE, d))
    metric_matrix = np.zeros((CLASS_SIZE, CLASS_SIZE))
    total_reduce_time = 0
    for i in range(CLASS_SIZE):
        for j in range(CLASS_SIZE):
            if i != j:
                metric_matrix[i][j], reduce_time = getError(i, j, Metric, d
                total_reduce_time += reduce_time
    return metric_matrix, total_reduce_time
```

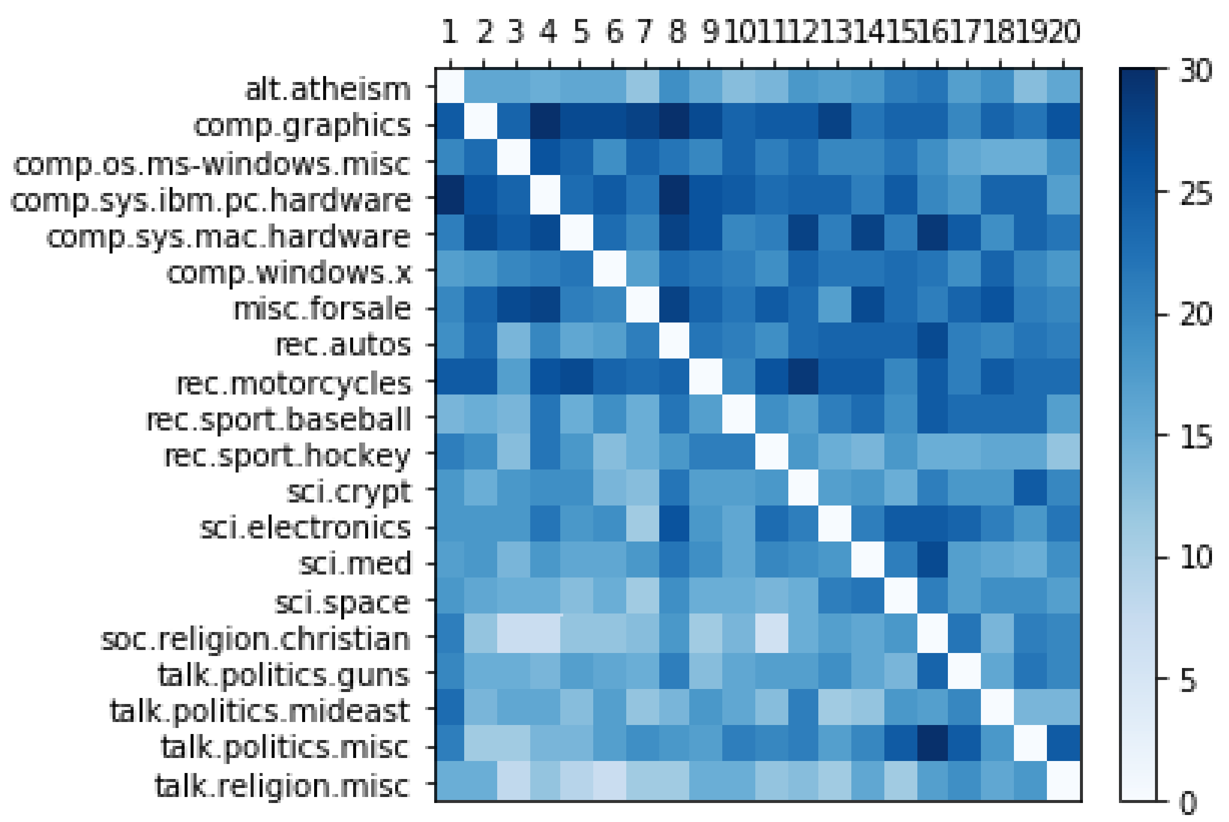
In [6]:

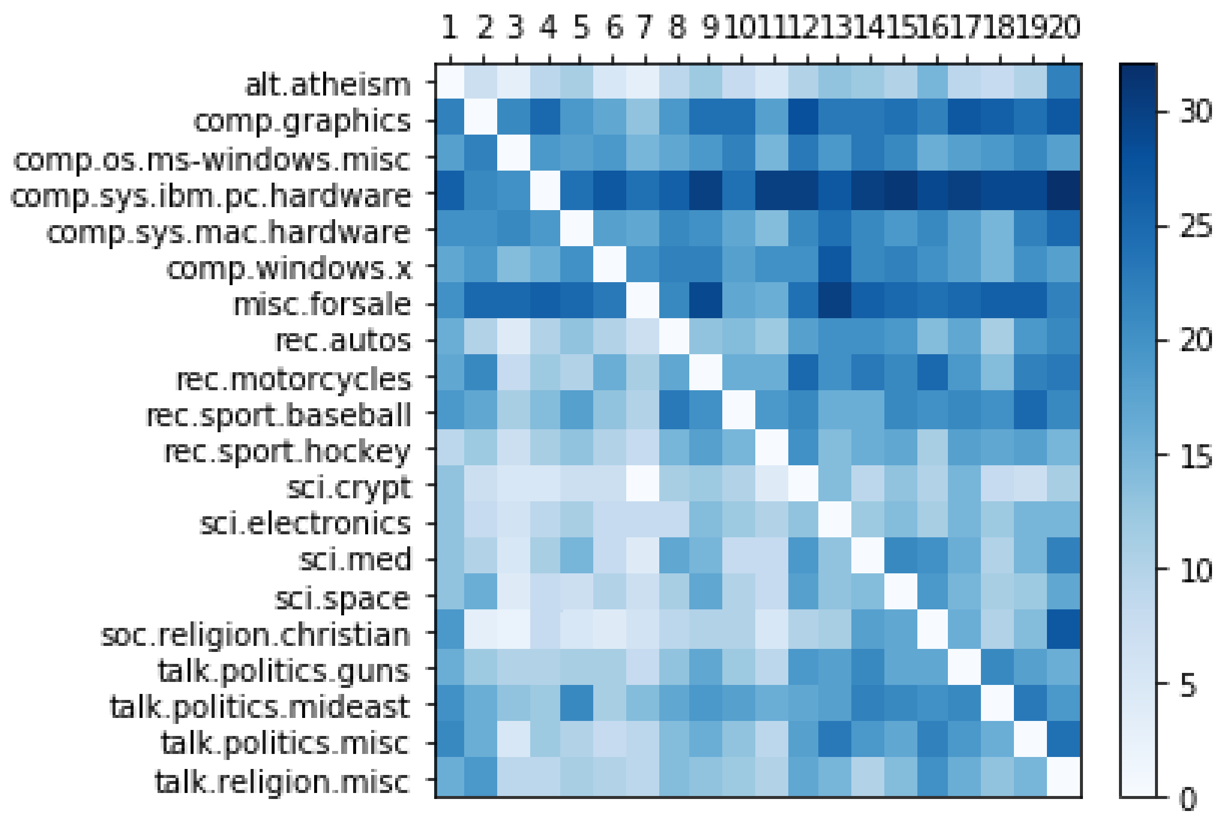
```
[illegible]
def getAvgError(Metric, d = None):
    KDMATRIX = None
    if type(d) != type(None):
        KDMATRIX = np.random.normal(0, 1, (WORD_SIZE, d))

    error_count = 0
    total_hash_time = 0
    time_start = time.clock()

    for i in range(DATA_SIZE):
        hash_time_start = time.clock()
        article = readArticle(i, KDMATRIX)
        article_group = readArticle(getGroupList(i // BATCH_SIZE), KDMATRIX
        total_hash_time += time.clock() - hash_time_start

        nearest_index, most_similar = nearestNeighbour(article, article_gro

        for j in range(CLASS_SIZE):
            if j != (i // BATCH_SIZE):
                hash_time_start = time.clock()
                article_group = readArticle(getGroupList(j), KDMATRIX)
                total_hash_time += time.clock() - hash_time_start

                index, similarity = nearestNeighbour(article, article_group

                if most_similar < similarity:
                    error_count += 1
                    break

    avg_hash_time = total_hash_time / 1000
    total_time = time.clock() - time_start
    error_rat
```

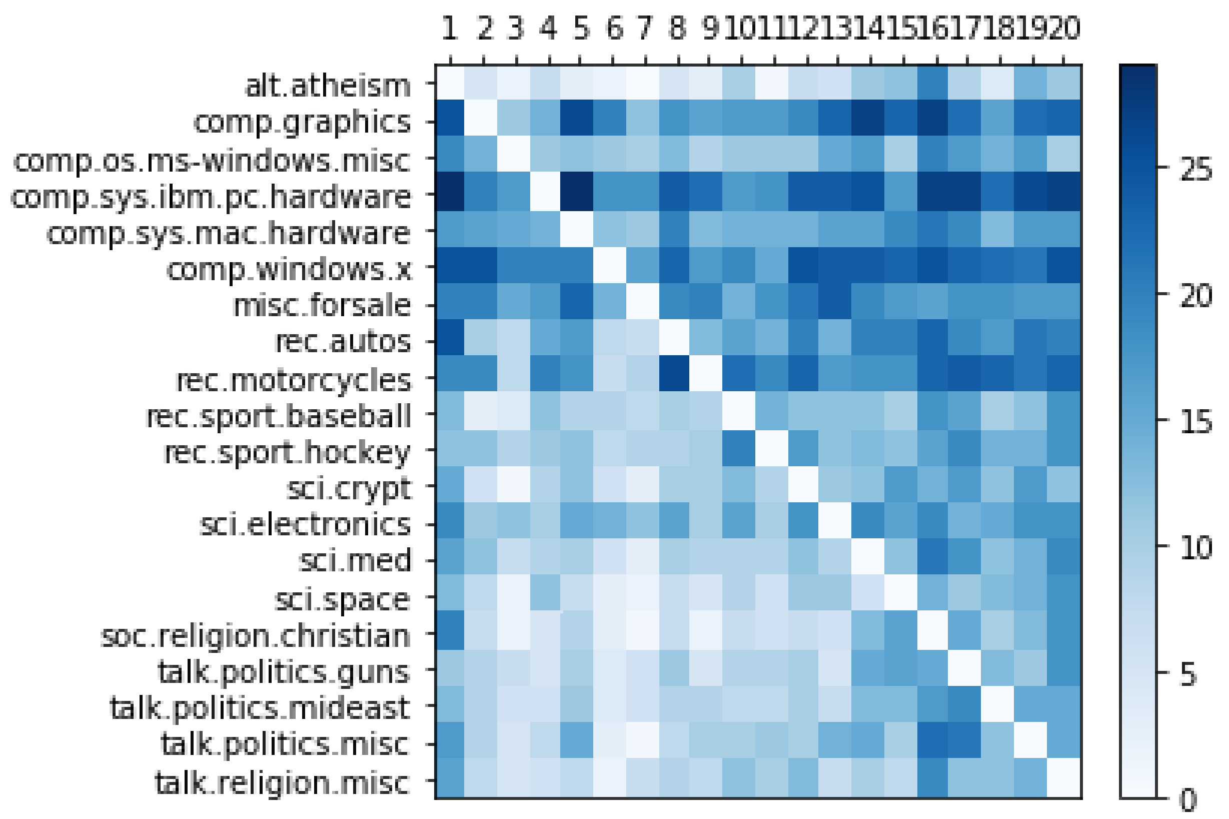
In [19]:

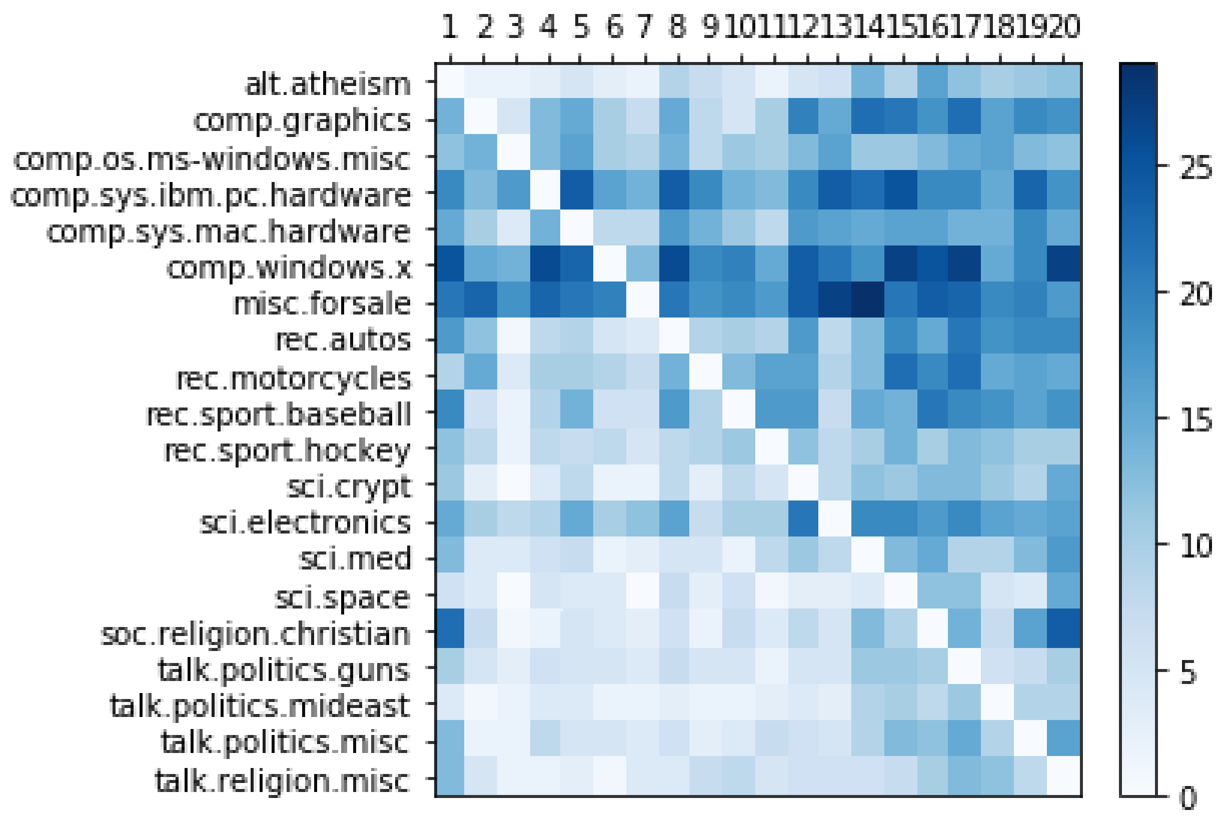
```
for d in [10,25,50,100]:
    time_begin = time.clock()
    error_matrix, reduce_dimension_time = getErrorMatrix(CosineSimilarity, 
    time_end = time.clock()

    print('d = %3d, reduce dimension_time = %6lf, total time used is %6lf' 
          %(d, reduce_dimension_time, time_end-time_begin))
    makeHeatMap(error_matrix, GROUPS, plt.cm.Blues)
```

```
d =  10, reduce dimension_time = 19.897009, total time used i
s 127.395688
d =  25, reduce dimension_time = 20.139775, total time used i
s 130.635457
d =  50, reduce dimension_time = 23.956532, total time used i
s 133.437758
d = 100, reduce dimension_time = 30.671653, total time used i
s 143.524978
```









In [24]:

```
for d in [5,10,15,20,25,50,100]:
    avgError, hash_time, total_time = getAvgError(CosineSimilarity, d)
    print('d = %4d, average error rate = %3lf, hash time = %6lf, total time
          %(d, avgError, hash_time, total_time))
```

```
d =    5, average error rate = 0.925000, hash time = 127.2750
24, total time = 144.771924
d =   10, average error rate = 0.859000, hash time = 175.8397
36, total time = 198.435276
d =   15, average error rate = 0.819000, hash time = 221.0182
95, total time = 246.386997
d =   20, average error rate = 0.797000, hash time = 231.9303
10, total time = 259.262133
d =   25, average error rate = 0.779000, hash time = 246.0126
17, total time = 274.947280
d =   50, average error rate = 0.706000, hash time = 333.1169
95, total time = 365.227891
d =  100, average error rate = 0.665000, hash time = 453.6302
71, total time = 489.434515
```

In [7]:

```
avgError, hash_time, total_time = getAvgError(CosineSimilarity)
print('original dimension runing time:')
print('average error rate = %3lf, hash time = %6lf, total time = %6lf' \
      %(avgError, hash_time, total_time))
```

```
original dimension runing time:
average error rate = 0.544000, hash time = 131.721036, total
time = 530.826413
```

From the result above, we can see that the more dimension we reduce, the less time it takes, and has higher error rate. This is because the demension reduction will lose the information of original data.

a-3. What is the average classification error (i.e., what fraction of the 1000 articles have the same newsgroup/label as their closest neighbor)?

(b) Your plots for Part 1(b) were symmetric—why is the matrix in (a) not symmetric?

Because for part1(b), the matrixes represent the similarity between two newsgroup, the similarity between A and B or the similarity between B and A have the same meaning. But in Part 2(a), the matrix represent the classification error, matrix(A, B) means which ought to be in A but in B. It is not the same as matrix(B, A) since the nearest neighbour of A_i is B_j doesn't illustarte that the nearest neighbour of B_j must be A_i

(c) Implement the random projection dimension reduction function and plot the nearest-neighbor visualization as in part (a) for cosine similarity and d = 10, 25, 50, 100.

What is the average classification error for each of these settings?

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

While feature dimension increases, running time to reduce dimension increasing exponentially. We can see that when dimension is 100, the running time to reduce dimension is nearly 4 times longer than to calculate the distance between each two vectors. We conclude that dimension equals to 25 is the comparable result to the original dataset since the wrong rate decrease dramatically comparing with dimension equals to 10 and its running time to reduce dimensino is not extremely expensive like dimension 50 and 100.

(d) What is the time it takes to reduce the dimensionality of the data? Suppose you are trying to build a very fast article classification system, and have an enormous dataset of n labeled tweets/articles. 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)?

The time without dimension reduction: $n^2 k$

The time with dimension reduction: $nkd + n^2 d$

The reduce time is: $n^2 k - nkd - n^2 d$

Only when $d < \frac{nk}{k+n}$, using the dimensionality reduction is useful to reduce time.

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

Because of the sparsity of the data, we could use partion sum to reduce the dimension. For example, we can make $D_j = \sum_{i=B_j}^{B_{j+1}-1} K_i,\ 1 = B_1 < B_2 < \cdots < B_{d+1} = K + 1,\ j = 1, 2, \ldots, d$ to map K into D, in this case, we can reduce the dimension and keep the position information of K(the zeros are still zeros).

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? [For this part, we expect a theoretical analysis—you do not need to implement these algorithms and measure their runtimes empirically.]

Only computing the times of multipy: $k \approx 60,000, d = 50, n = 1,000$ the reduce time is:
$n^2 k - nkd - n^2 d = n^2 (k - d) - nkd = 56,950,000,000$

Part 3: Locality Sensitive Hashing

(a) Suppose is the row of M, if and only if x, y belong to the same hyperspace partitioned by the normal of , , probability is

In [22]:

```
l = 128
UNIT_VECTOR = np.zeros(21)

for i in range(21):
    UNIT_VECTOR[i] = 2**i

for d in range(5, 21, 5):

    H = np.random.normal(0, 1, (l, d, WORD_SIZE))
    BUKETS = np.zeros((l, DATA_SIZE))

    time_begin = time.clock()

    for i in range(DATA_SIZE):
        BUKETS[:, i] = ((np.sign(H.dot(readArticle(i))) + 1) / 2).dot(UNIT_

    time_put_in_bukets = time.clock()



    error = 0
    colides_times = 0
    for i in range(DATA_SIZE):


        colides = []
        for j in range(l):

            _, colide = np.where(BUKETS==BUKETS[j, i])
            colides = np.concatenate([colides, colide])

        colides = np.unique(colides)
        colides_times += len(colides)-1


        similar_index = 0
        most_similarity = -sys.maxsize
        for c in colides:
            c = int(c)
            if (c != i):
                similarity = CosineSimilarity(readArticle(i), readArticle(c
                if similarity > most_similarity:
                    most_similarity = similarity
                    similar_index = c

        if LABELS[similar_index] != LABELS[i]:

            error += 1

    time_end = time.clock()
    print("d =%3d, Time to hash: %8lf, Total Time: %8lf, avg error rate: %8
          % (d, time_put_in_bukets-time_begin, time_end-time_begin, error/1
```

```
d =  5, Time to hash: 69.979264, Total Time: 1134.045402, avg
error rate: 0.544000, avg collision =  999
d = 10, Time to hash: 128.132905, Total Time: 431.752395, avg
error rate: 0.544000, avg collision =  999
d = 15, Time to hash: 186.038179, Total Time: 355.624006, avg
```

```
error rate: 0.586000, avg collision =  481
d = 20, Time to hash: 255.853117, Total Time: 309.306317, avg
error rate: 0.700000, avg collision =   41
```

(e)

When dimension grows, it takes more time to hash both in part2 and part3 and get higher correct rate. In part2, the correct rate increases dramatically at first while the growth rate of correction drops with the increasing dimension. In part3, the correct rate increases little at first while the growth rate of correction rockets in some range of dimension and then the growth rate drops down. Combine the growing properties of these two parts, we may get a better performance.

There are two ways to combine. One is to reduce dimension before hashing. The other one is hashing at first, and count the similarity with dimension reduction. Though we may not use all the original data in LSH, we should still reduce the dimension before hash as the latter one above, since the time cost of dimension reduction is so large while the time for LSH is little (the major time cost is to compare the data on original demision).

In order to engage the acceptable correct rate and minimize running time, we choose the hash dimension as 15 and the result demision reduction as 100. Calculating the hash values by LSH, we choose the colide items for the input vector and compare the reduced demision similarity among them