Part 1

In [16]:
```
import numpy as np
import sklearn
import nltk, string
import matplotlib.pyplot as plt
```

In [4]:
```
from sklearn.datasets import fetch_20newsgroups
categories = ['comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys
              'rec.autos', 'rec.motorcycles', 'rec.sport.baseball',
dataset = fetch_20newsgroups(subset='all', categories=categories)
labels = dataset.target
from pprint import pprint
pprint(list(dataset.target_names))
```

```
['comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey']
```

Question 1:

Report the dimensions of the TF-IDF matrix you get.

In [5]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vect = TfidfVectorizer(stop_words='english',min_df=3)
X_train_tfidf = tfidf_vect.fit_transform(dataset.data) # making the
print("Shape of TF-IDF matrix: ", X_train_tfidf.shape)
```

```
Shape of TF-IDF matrix:  (7882, 27768)
```

QUESTION 2:

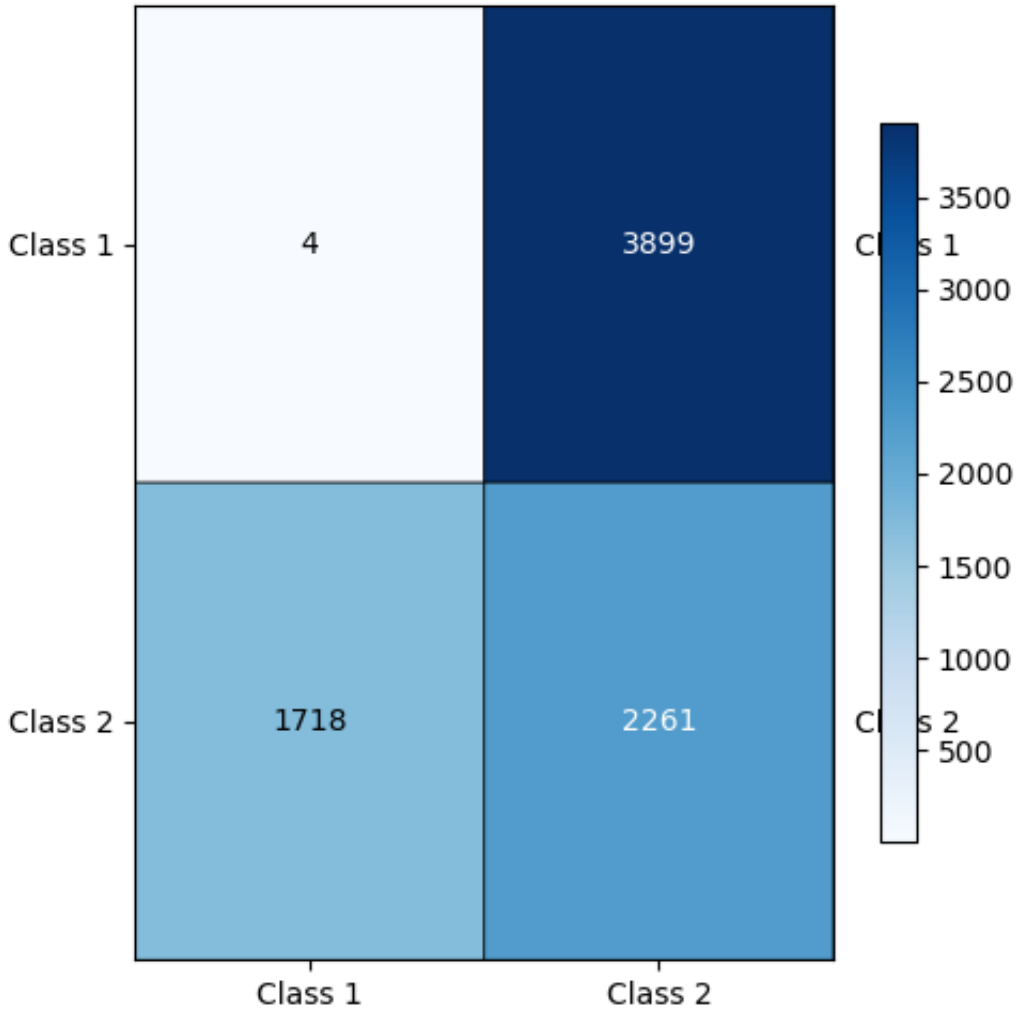
Report the contingency table of your clustering result. You may usethe provided plotmat.py to visualize the matrix. Does the contingency matrix have to be square-shaped?

A: The contigency matrix doesn't have to be square-shaped because the contingency table is a matrix which allows you to view the frequency of occurrence between the different combinations of your X and Y variables.

In [13]:
```
from sklearn.cluster import KMeans
from sklearn.metrics.cluster import contingency_matrix, homogeneity_

y_true = [int(i/4) for i in dataset.target]

km = KMeans(n_clusters=2, random_state=0, max_iter=1000, n_init=30)
y_pred = km.fit_predict(X_train_tfidf)
con_mat = contingency_matrix(y_true,y_pred)
print("Contingency table: \n", con_mat)
```

```
Contingency table:
 [[   4 3899]
 [1718 2261]]
```

In [14]:
```
import sys
sys.path.append('.')
from plotmat import plot_mat
plot_mat(con_mat,size=(5,5),xticklabels = ['Class 1','Class 2'],yti
```



Question 3

Report the 5 clustering measures explained in the introduction for K- means clustering.

In [20]:
```
print("Homogeneity: %0.3f" % homogeneity_score(y_true, y_pred))
print("Completeness : %0.3f" % completeness_score(y_true, y_pred))
print("V-measure : %0.3f" % v_measure_score(y_true, y_pred))
print("Adjusted Rand-Index : %.3f"% adjusted_rand_score(y_true, y_p
print("Adjusted Mutual Information Score : %.3f"% adjusted_mutual_i
```

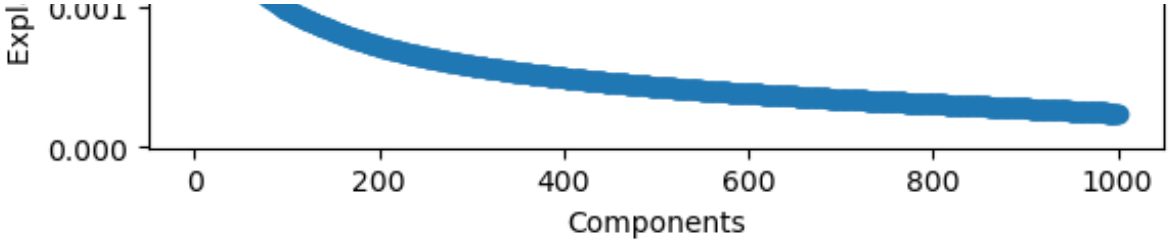
```
Homogeneity: 0.254
Completeness : 0.335
V-measure : 0.289
Adjusted Rand-Index : 0.181
Adjusted Mutual Information Score : 0.289
```

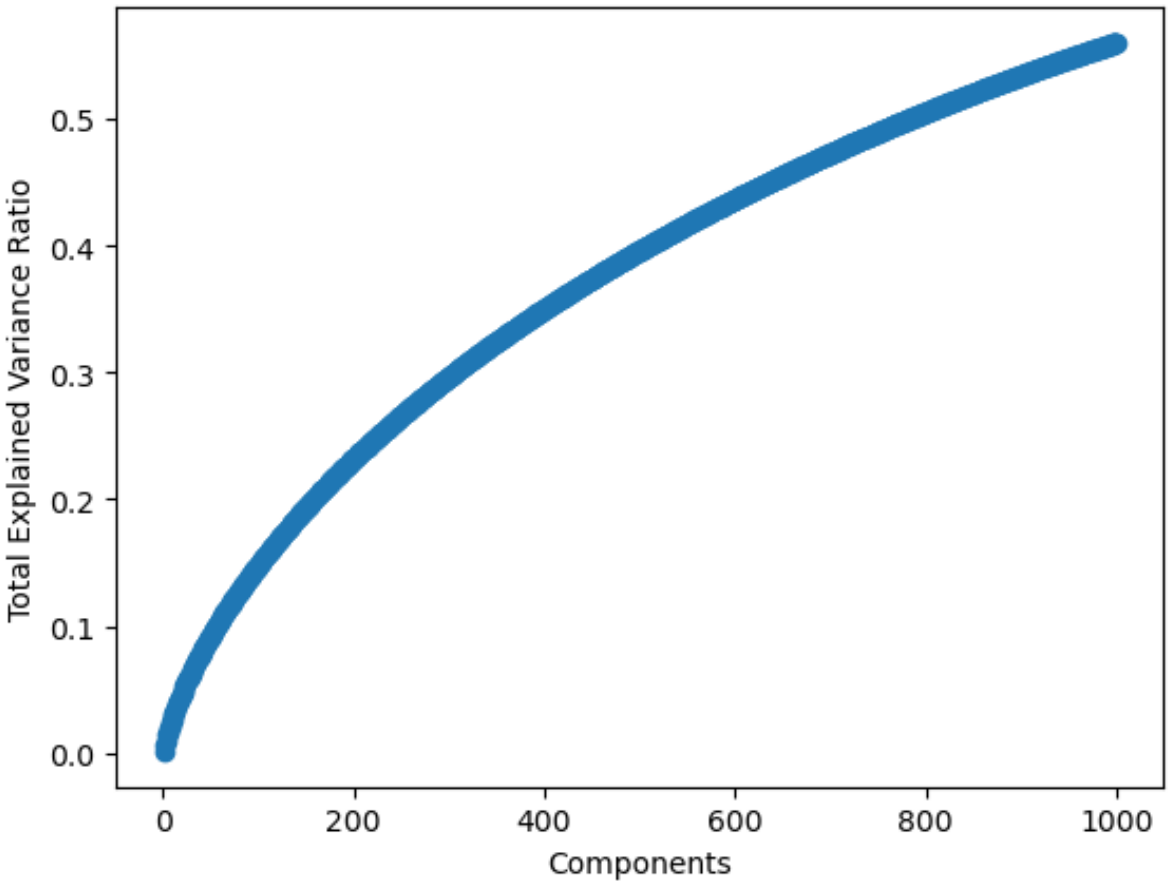
QUESTION 4:

Report the plot of the percentage of variance that the top r principle components retain v.s. r, for r = 1 to 1000.

In [8]:
```
from sklearn.decomposition import TruncatedSVD, NMF
from sklearn.utils.extmath import randomized_svd

svd = TruncatedSVD(n_components=1000)
X_train_svd = svd.fit_transform(X_train_tfidf)
plt.figure()
plt.plot(np.arange(1000)+1,sorted(svd.explained_variance_ratio_,rev
plt.scatter(np.arange(1000)+1,sorted(svd.explained_variance_ratio_,
plt.xlabel("Components"); plt.ylabel("Explained Variance Ratio per

plt.figure()
plt.plot(np.arange(1000)+1,np.cumsum(svd.explained_variance_ratio_)
plt.scatter(np.arange(1000)+1,np.cumsum(svd.explained_variance_rati
plt.xlabel("Components"); plt.ylabel("Total Explained Variance Rati
```

Out[8]:
```
Text(0, 0.5, 'Total Explained Variance Ratio')
```





QUESTION 5:

Let r be the dimension that we want to reduce the data to (i.e. n components). Try r = 1 −10, 20, 50, 100, 300, and plot the 5 measure scores v.s. r for both SVD and NMF. Report a good choice of r for SVD and NMF respectively. Note: In the choice of r, there is a trade-off between the information preservation, and better performance of k-means in lower dimensions.

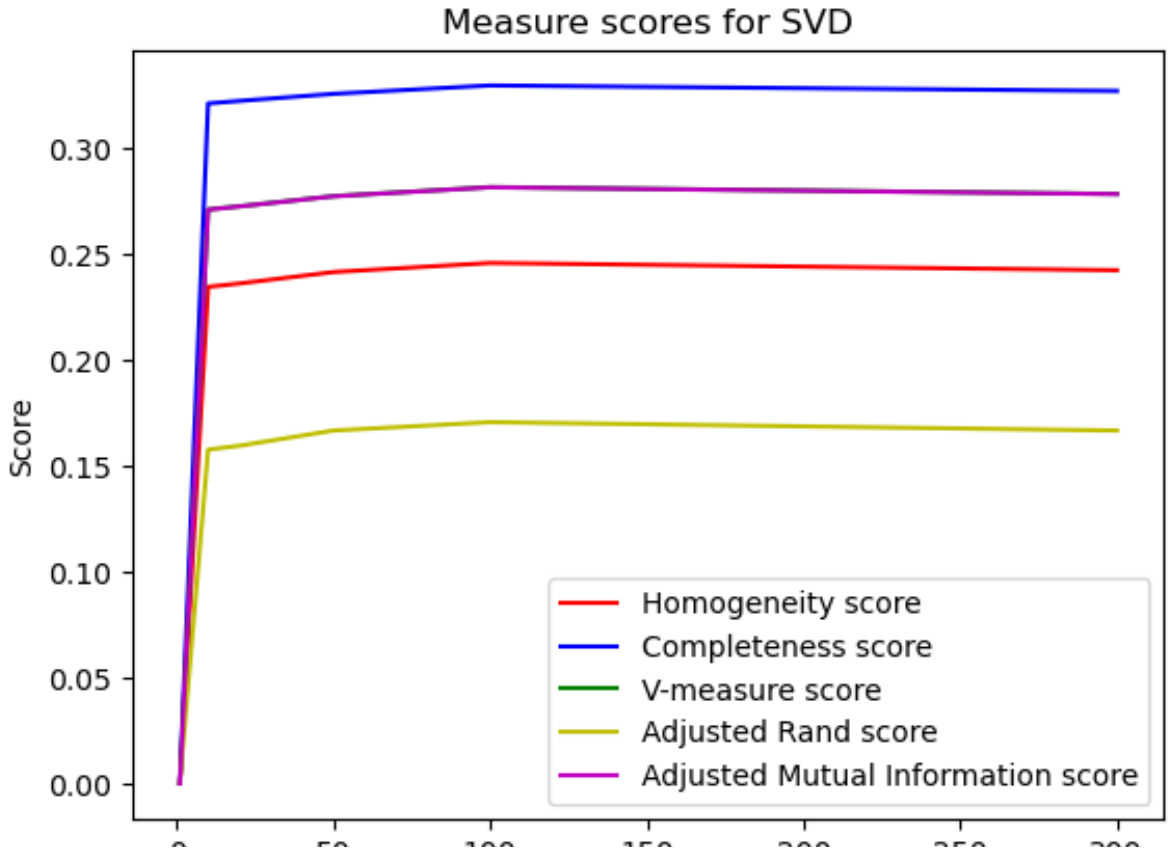
A: According to the charts below, the good choice of r for SVD and NMF is both 10.

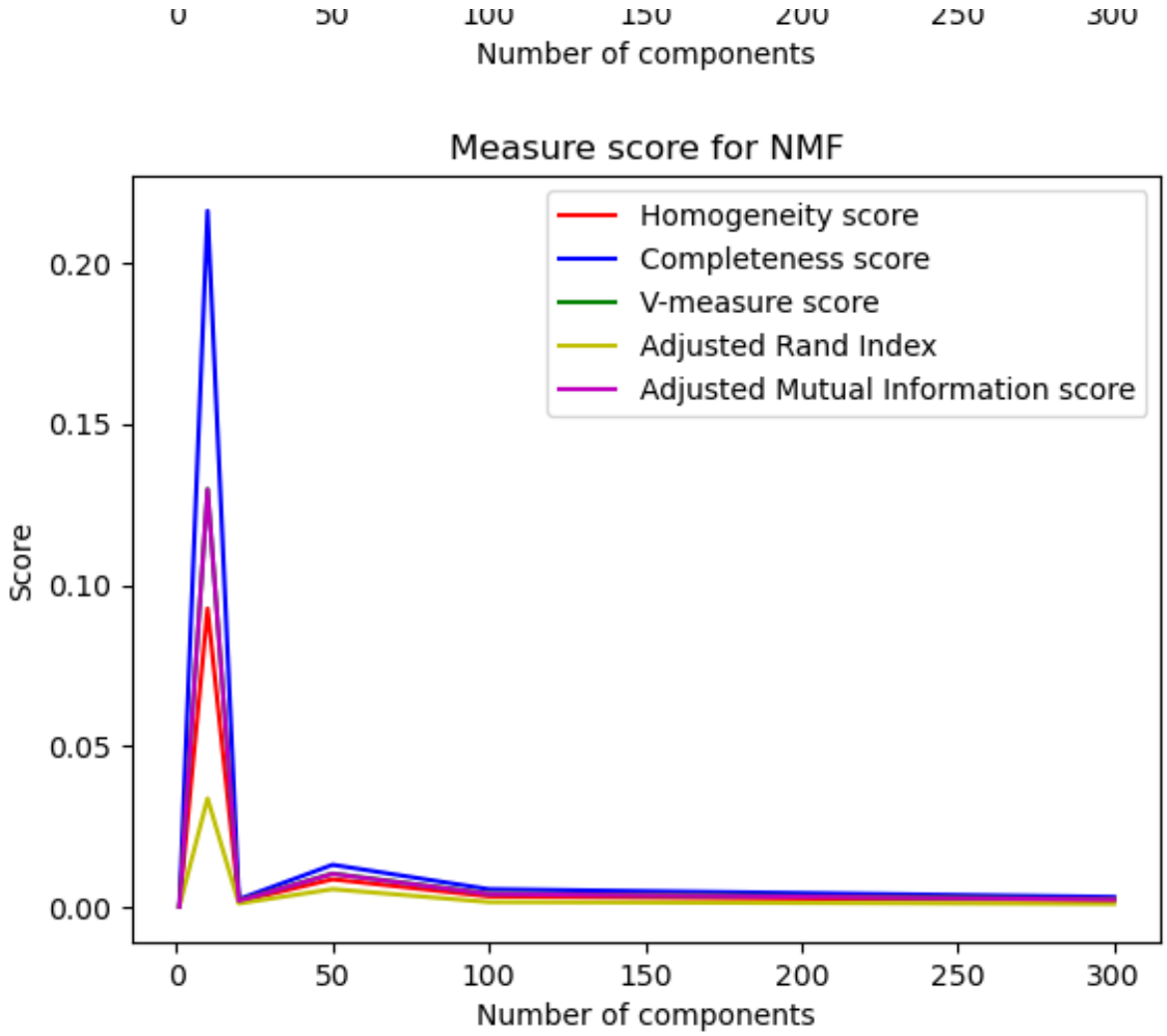
In [21]:
```
from sklearn.decomposition import NMF

r = [1,10,20,50,100,300]
hom_score = []; complt_score = []; v_score = []; adj_rand_score = [
for i in r:
```

```
    y_pred = km.fit_predict(TruncatedSVD(n_components=i,random_stat
    hom_score.append(homogeneity_score(y_true,y_pred))
    complt_score.append(completeness_score(y_true,y_pred))
    v_score.append(v_measure_score(y_true,y_pred))
    adj_rand_score.append(adjusted_rand_score(y_true,y_pred))
    adj_mut_inf_score.append(adjusted_mutual_info_score(y_true,y_pr

fig, ax = plt.subplots()
ax.plot(r,hom_score, 'r', label='Homogeneity score')
ax.plot(r, complt_score, 'b', label='Completeness score')
ax.plot(r, v_score, 'g', label='V-measure score')
ax.plot(r,adj_rand_score,'y',label='Adjusted Rand score')
ax.plot(r,adj_mut_inf_score,'m',label='Adjusted Mutual Information
ax.legend(loc='best')
plt.xlabel("Number of components"); plt.ylabel("Score"); plt.title(
print("SVD")
print('Homogeneity score: ', hom_score)
print('Completeness score:', complt_score)
print('V-measure score: ', v_score)
print('Adjusted Rand score: ', adj_rand_score)
print('Adjusted Mutual Information score: ', adj_mut_inf_score)

hom_score = []; complt_score = []; v_score = []; adj_rand_score = [
for i in r:
    y_pred = km.fit_predict(NMF(n_components=i,init='random',random
    hom_score.append(homogeneity_score(y_true,y_pred))
    complt_score.append(completeness_score(y_true,y_pred))
    v_score.append(v_measure_score(y_true,y_pred))
    adj_rand_score.append(adjusted_rand_score(y_true,y_pred))
    adj_mut_inf_score.append(adjusted_mutual_info_score(y_true,y_pr

fig, ax = plt.subplots()
ax.plot(r,hom_score, 'r', label='Homogeneity score')
ax.plot(r, complt_score, 'b', label='Completeness score')
ax.plot(r, v_score, 'g', label='V-measure score')
ax.plot(r,adj_rand_score,'y',label='Adjusted Rand Index')
ax.plot(r,adj_mut_inf_score,'m',label='Adjusted Mutual Information
ax.legend(loc='best')
plt.xlabel("Number of components"); plt.ylabel("Score"); plt.title(
print("NMF")
print(hom_score)
print(complt_score)
print(v_score)
print(adj_rand_score)
print(adj_mut_inf_score)
```

```
SVD
Homogeneity score:  [0.00029275270509964866, 0.23462501869262953,
0.23622580967999646, 0.24162603939083188, 0.2459136851918752, 0.24
2488600144672]
Completeness score: [0.000297122118721392, 0.3212187345416125, 0.3
2242397160243236, 0.3257637898975148, 0.32972145379678475, 0.32714
066941714903]
```

```
V-measure score:  [0.00029492122900643043, 0.2711767512281033, 0.2
7267481812015304, 0.27745655726190943, 0.28171670663624065, 0.2785
246026362978]
Adjusted Rand score:  [0.0003281607588681523, 0.1577989589471986,
0.15961920123127504, 0.16679707546182895, 0.1707597324152979, 0.16
679709209461066]
Adjusted Mutual Information score:  [0.00020271945487688344, 0.271
0996208286373, 0.2725979456601095, 0.2773805882185429, 0.281641363
14170246, 0.27844872472650617]

/Users/ryan/opt/anaconda3/lib/python3.9/site-packages/sklearn/deco
mposition/_nmf.py:1637: ConvergenceWarning: Maximum number of iter
ations 200 reached. Increase it to improve convergence.
  warnings.warn(

NMF
[0.0003003030178761853, 0.09267200875838809, 0.0019120402724064785
, 0.008734874759382975, 0.003444719931676562, 0.002163728856299646
]
[0.0003047688479979988, 0.21597065707166085, 0.0025441502068279777
, 0.013232995618956961, 0.005663833745154288, 0.003260895350137586
2]
[0.0003025194525487269, 0.129693246200255, 0.0021832628911060627,
0.01052341965173116, 0.004283955869143768, 0.0026013574758204493]
[0.0003390408027462719, 0.03383565814396721, 0.0012379619230262304
, 0.005683121258793998, 0.0016856298724459546, 0.00111394559908445
34]
[0.00021032082455495874, 0.12958167529017284, 0.002078940382781518
, 0.01041426109884003, 0.004170558556580682, 0.0024915538823062316
]
```





QUESTION 6:

How do you explain the non-monotonic behavior of the measures as r increases?

A: There is a non-monotonic behavior in the measures as r increases. As the number of components increases, the dimensions in which k-means needs to perform clustering increases. It is a well-known fact that k-means suffers from the curse of dimensionality because the Euclidean distance is not a good metric in high dimensions since the ratio between the nearest and farthest points approaches. This means that points in high dimensions are essentially equidistant from each other which makes it hard to perform clustering.

QUESTION 7:

Are these measures on average better than those computed in Question 3?

A: The best result in SVD is better on average in the measures than the result computed just using tf-idf. However the best result using NMF is not better on average in the measures comparing to the rresults in question 3.

QUESTION 8:

Visualize the clustering results for:

•SVD with your optimal choice of r for K-Means clustering;

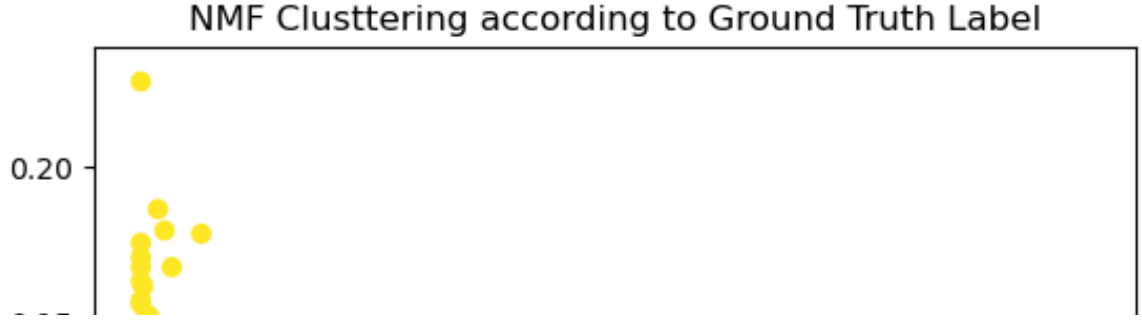
•NMF with your choice of r for K-Means clustering.

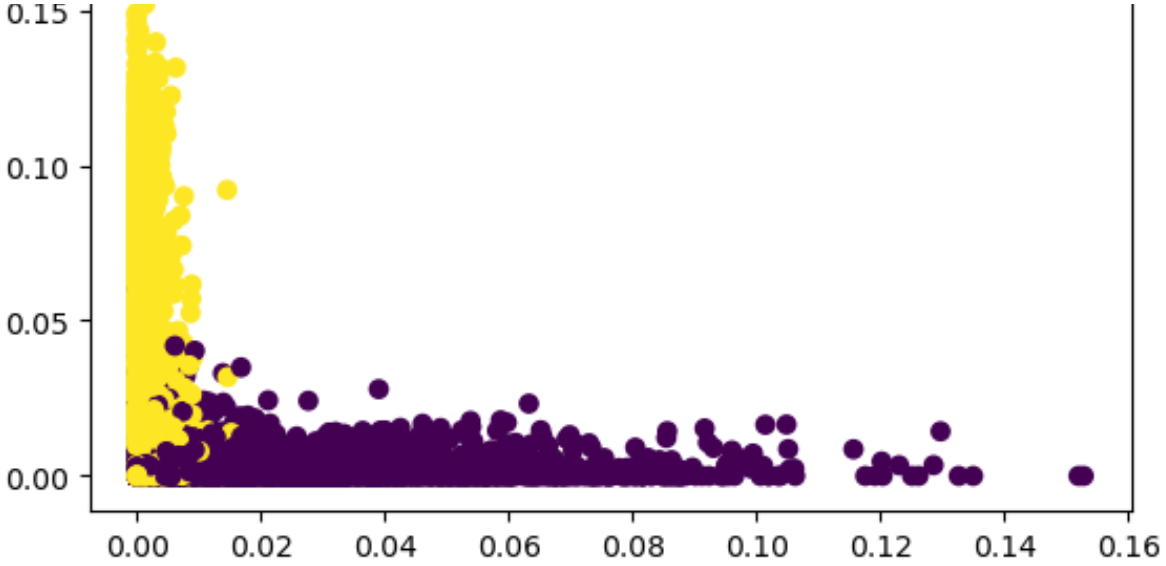
To recap, you can accomplish this by first creating the dense representations and then once again projecting these representations into a 2-D plane for visualization.

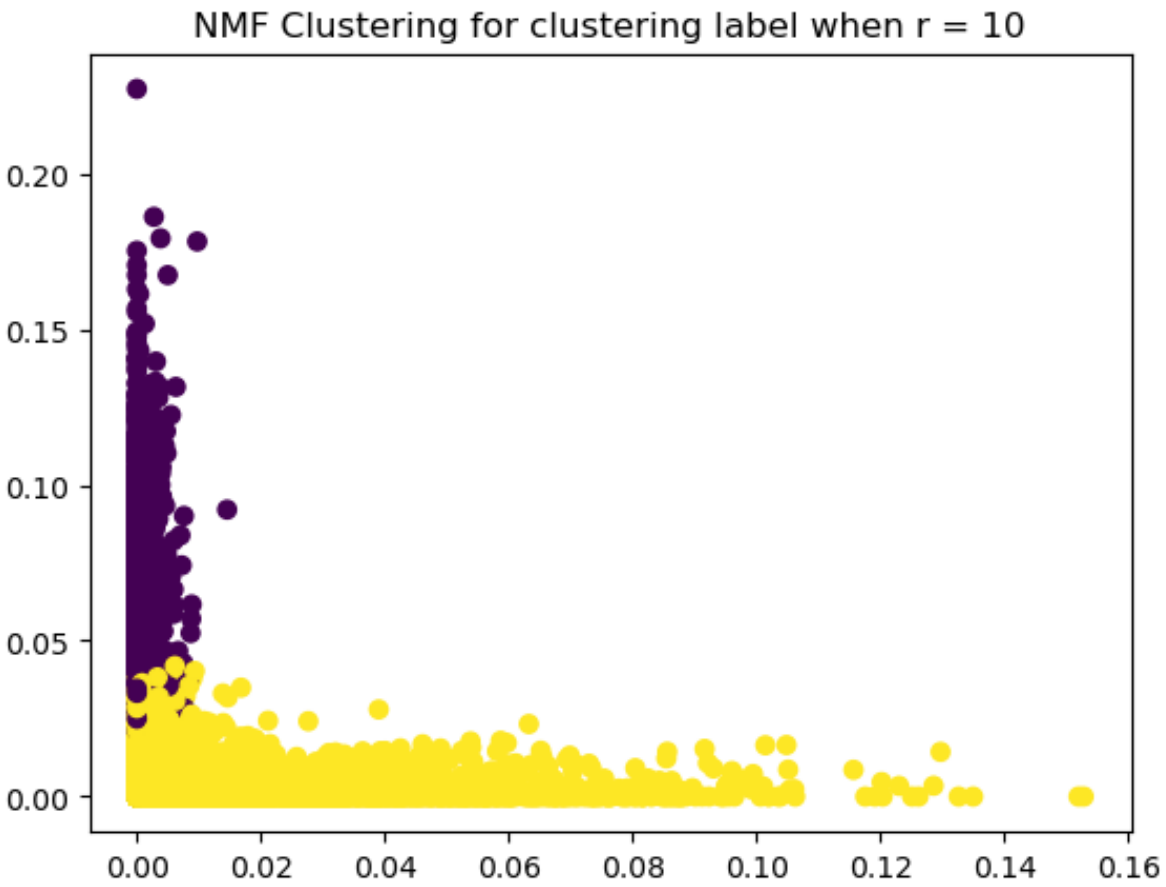
In [22]:
```
#Ground Truth plot for NMF r=10
from sklearn.decomposition import NMF
nmf_10 = NMF(n_components = 10).fit_transform(X_train_tfidf)
plt.figure()
plt.scatter(nmf_10[:,0],nmf_10[:,1],c=y_true)
plt.title("NMF Clusttering according to Ground Truth Label")
#Clustering label plot for NMF
c_nmf = km.fit_predict(nmf_10)
plt.figure()
plt.scatter(nmf_10[:,0],nmf_10[:,1], c=c_nmf)
plt.title('NMF Clustering for clustering label when r = 10')
```

```
/Users/ryan/opt/anaconda3/lib/python3.9/site-packages/sklearn/deco
mposition/_nmf.py:289: FutureWarning: The 'init' value, when 'init
=None' and n_components is less than n_samples and n_features, wil
l be changed from 'nndsvd' to 'nndsvda' in 1.1 (renaming of 0.26).
  warnings.warn(
```

Out[22]: Text(0.5, 1.0, 'NMF Clustering for clustering label when r = 10')

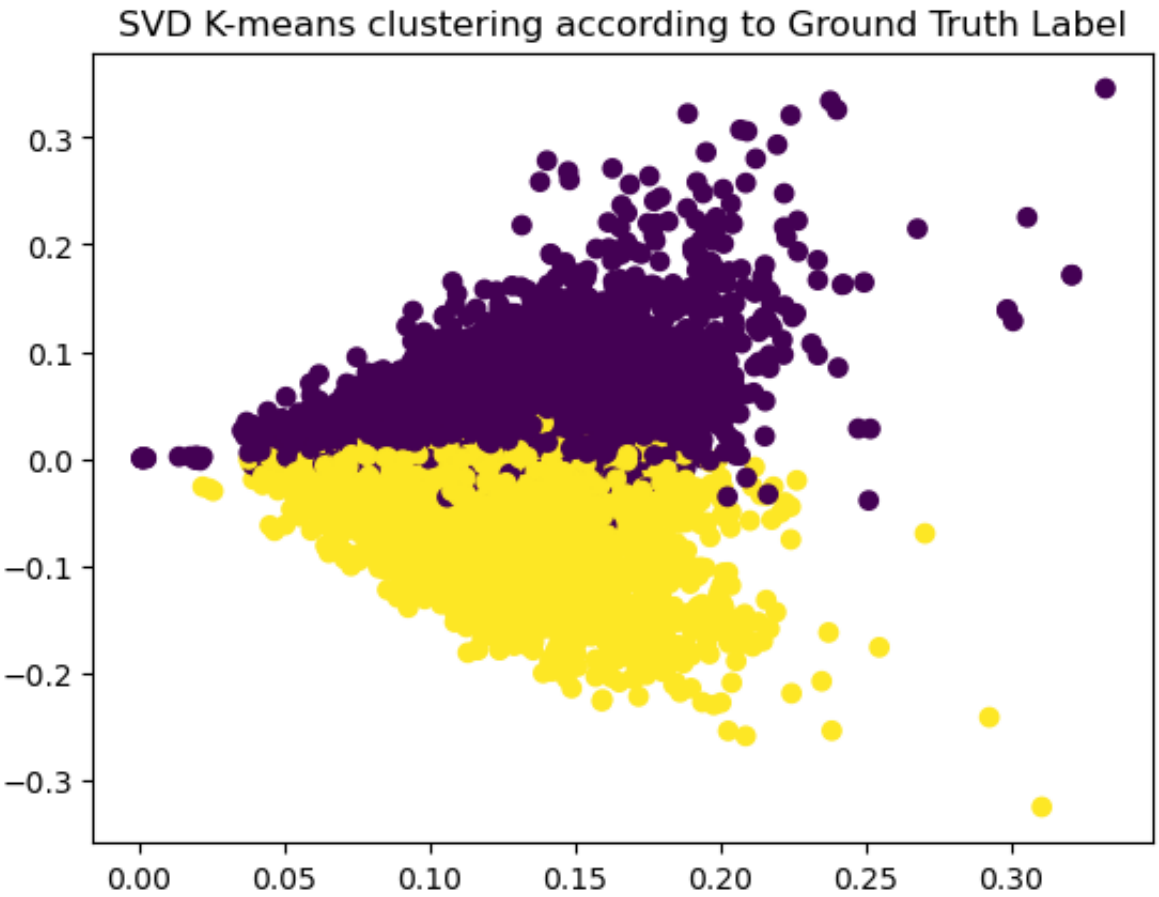



0.15
0.10
0.05
0.00
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16

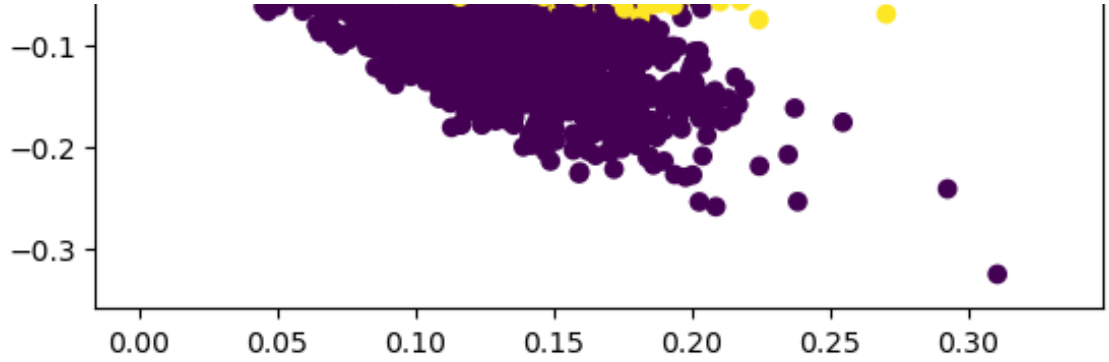

NMF Clustering for clustering label when r = 10
0.20
0.15
0.10
0.05
0.00
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16

In [23]:

```
##Ground Truth plot for SVD 10
svd_10 = TruncatedSVD(n_components = 10).fit_transform(X_train_tfid
plt.show()
plt.scatter(svd_10[:,0],svd_10[:,1], c=y_true)
plt.title("SVD K-means clustering according to Ground Truth Label")
#Cluster label plot for SVD 10
svd_label = km.fit_predict(svd_10)
plt.show()
plt.scatter(svd_10[:,0],svd_10[:,1],c=svd_label)
plt.title("SVD K-means clustering according to Clustering Label whe
```



Out[23]: Text(0.5, 1.0, 'SVD K-means clustering according to Clustering Lab
el when r = 10')



QUESTION 9:

What do you observe in the visualization? How are the data points of the two classes distributed? Is distribution of the data ideal for K-Means clustering?

A: By use of the Euclidean distance, K-means treats the data space as isotropic, from the above four plots, we can observe that there are both overlapping in both SVD and NMF, which implies that the euclidean distance of two centroid is close and the result may not be that accurate. Also the clustering shsape return by NMF and SVD is both not in 2D spherical shape, especially in the NMF.

QUESTION 10:

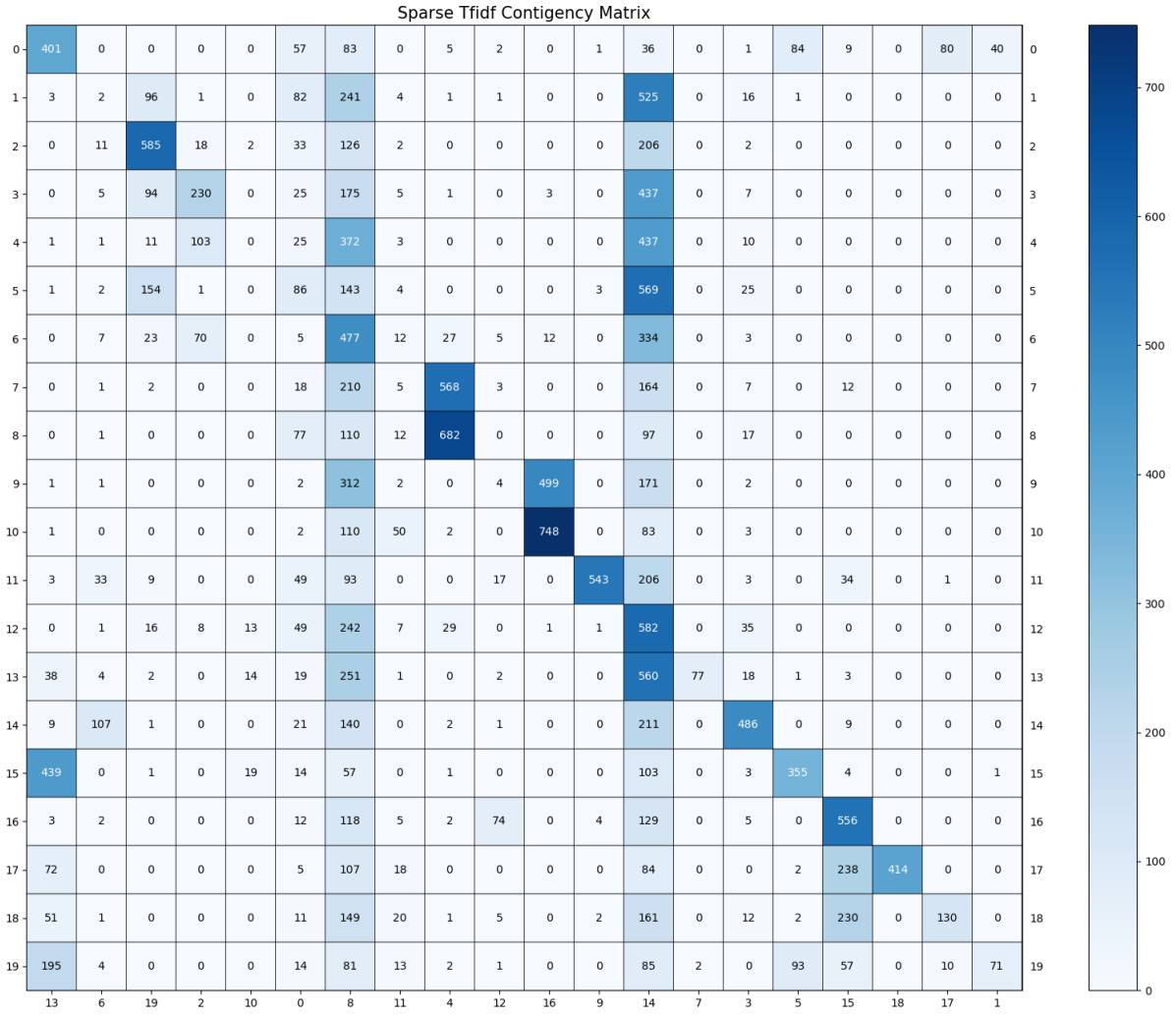
Load documents with the same configuration as in Question 1, but for ALL 20 categories. Construct the TF-IDF matrix, reduce its dimensionality using BOTH NMF and SVD (specify settings you choose and why), and perform K-Means clustering with k=20 . Visualize the contingency matrix and report the five clustering metrics (DO BOTH NMF AND SVD). There is a mismatch between cluster labels and class labels. For example, the cluster #3 may correspond to the class #8. As a result, the high-value entries of the 20 ×20 contingency matrix can be scattered around, making it messy to inspect, even if the clustering result is not bad. One can use scipy.optimize.linear_sum_assignment to identify the best-matching cluster-class pairs, and permute the columns of the contingency matrix accordingly. See below for an example:

In [8]:
```
#get data
from sklearn.decomposition import TruncatedSVD, NMF
from sklearn.utils.extmath import randomized_svd
from scipy.optimize import linear_sum_assignment
dataset_20 = fetch_20newsgroups(subset='all')
labels20 = dataset_20.target
from pprint import pprint
pprint(list(dataset_20.target_names))
#do tfidf
X20_train_tfidf = tfidf_vect.fit_transform(dataset_20.data) # makin
print("Shape of TF-IDF matrix: ", X20_train_tfidf.shape)
```

```
['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']
Shape of TF-IDF matrix:  (18846, 52295)
```

Sparse Tfidf contigency matrix

In [10]:
```
kmeans = KMeans(n_clusters=20, random_state=0, max_iter=1000, n_ini
y20_pred = kmeans.fit_predict(X20_train_tfidf)
con_mat20 = contingency_matrix(labels20,y20_pred)
rows, cols = linear_sum_assignment(con_mat20, maximize=True)
plot_mat(con_mat20[rows[:, np.newaxis], cols], xticklabels=cols, yt
print("Homogeneity : %0.3f" % homogeneity_score(labels20, y20_pred)
print("Completeness : %0.3f" % completeness_score(labels20, y20_pre
print("V-measure : %0.3f" % v_measure_score(labels20, y20_pred))
print("Adjusted Rand-Index : %.3f"% adjusted_rand_score(labels20, y
print("Adjusted Mutual Information Score : %.3f"% adjusted_mutual_i
```



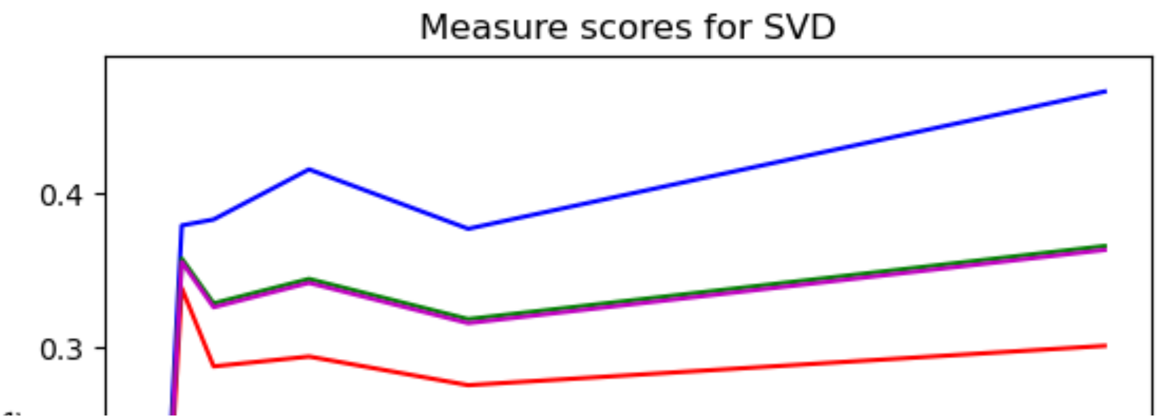

```
Homogeneity : 0.359
Completeness : 0.451
V-measure : 0.400
Adjusted Rand-Index : 0.137
Adjusted Mutual Information Score : 0.398
```

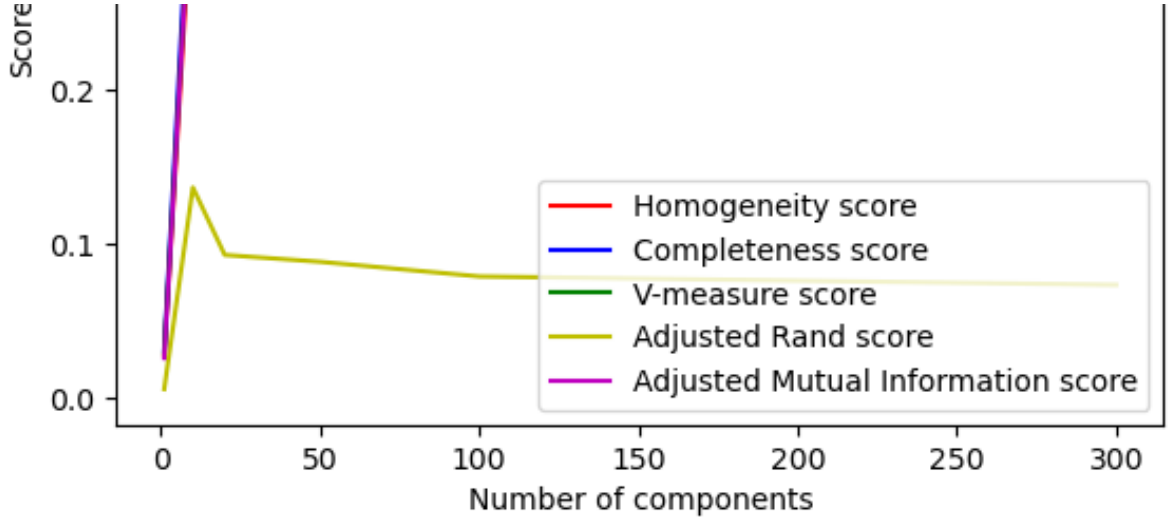
In [28]:
```
```

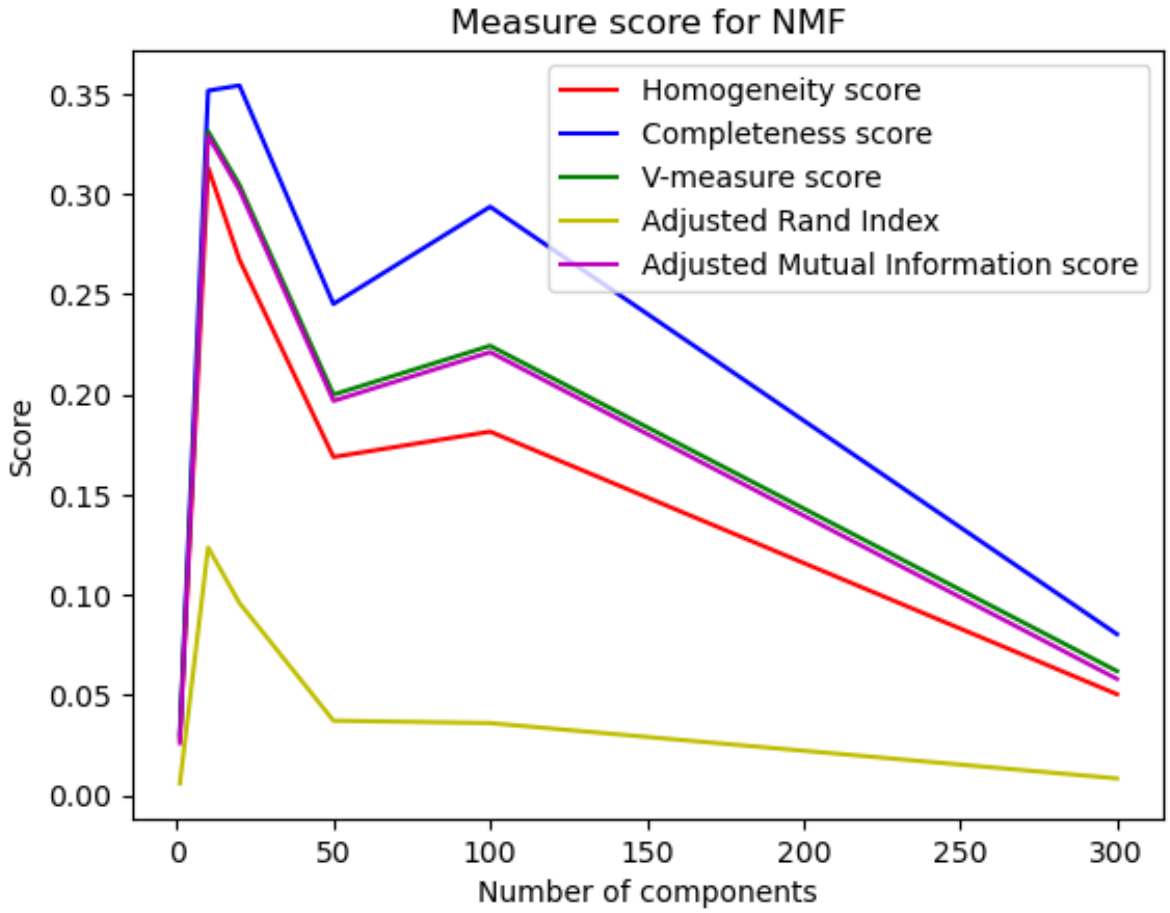
```
r = [1,10,20,50,100,300]
hom_score = []; complt_score = []; v_score = []; adj_rand_score = [
for i in r:
    y20_pred = kmeans.fit_predict(TruncatedSVD(n_components=i,rando
    hom_score.append(homogeneity_score(labels20,y20_pred))
    complt_score.append(completeness_score(labels20,y20_pred))
    v_score.append(v_measure_score(labels20,y20_pred))
    adj_rand_score.append(adjusted_rand_score(labels20,y20_pred))
    adj_mut_inf_score.append(adjusted_mutual_info_score(labels20,y2

fig, ax = plt.subplots()
ax.plot(r,hom_score, 'r', label='Homogeneity score')
ax.plot(r, complt_score, 'b', label='Completeness score')
ax.plot(r, v_score, 'g', label='V-measure score')
ax.plot(r,adj_rand_score,'y',label='Adjusted Rand score')
ax.plot(r,adj_mut_inf_score,'m',label='Adjusted Mutual Information
ax.legend(loc='best')
plt.xlabel("Number of components"); plt.ylabel("Score"); plt.title(
print("SVD")
print('Homogeneity score: ', hom_score)
print('Completeness score:', complt_score)
print('V-measure score: ', v_score)
print('Adjusted Rand score: ', adj_rand_score)
print('Adjusted Mutual Information score: ', adj_mut_inf_score)

hom_score = []; complt_score = []; v_score = []; adj_rand_score = [
for i in r:
    y20_pred = kmeans.fit_predict(NMF(n_components=i,init='random',
    hom_score.append(homogeneity_score(labels20,y20_pred))
    complt_score.append(completeness_score(labels20,y20_pred))
    v_score.append(v_measure_score(labels20,y20_pred))
    adj_rand_score.append(adjusted_rand_score(labels20,y20_pred))
    adj_mut_inf_score.append(adjusted_mutual_info_score(labels20,y2

fig, ax = plt.subplots()
ax.plot(r,hom_score, 'r', label='Homogeneity score')
ax.plot(r, complt_score, 'b', label='Completeness score')
ax.plot(r, v_score, 'g', label='V-measure score')
ax.plot(r,adj_rand_score,'y',label='Adjusted Rand Index')
ax.plot(r,adj_mut_inf_score,'m',label='Adjusted Mutual Information
ax.legend(loc='best')
plt.xlabel("Number of components"); plt.ylabel("Score"); plt.title(
print("NMF")
print(hom_score)
print(complt_score)
print(v_score)
print(adj_rand_score)
print(adj_mut_inf_score)
```

```
SVD
Homogeneity score:  [0.0280029766961423, 0.33868411668173454, 0.28
79281718544785, 0.2942994589856629, 0.2757058935951528, 0.30124027
```

```
72300583]
Completeness score: [0.031030705737015128, 0.3793967975868234, 0.3
832889795236285, 0.41581412561515085, 0.37724085934887175, 0.46635
795988432077]
V-measure score:  [0.029439197888506435, 0.3578863236978208, 0.328
83455060593253, 0.3446599948539983, 0.3185788971564128, 0.36603992
643892314]
Adjusted Rand score:  [0.005927829387410708, 0.1368534650825502 5,
0.0930576459033265, 0.08858305613659094, 0.07908366130289018, 0.07
362118184161288]
Adjusted Mutual Information score:  [0.026112451413343993, 0.35568
197267845625, 0.3263237670102939, 0.3421392148434956, 0.3159984405
3794296, 0.36350804880170234]

/Users/ryan/opt/anaconda3/lib/python3.9/site-packages/sklearn/deco
mposition/_nmf.py:1637: ConvergenceWarning: Maximum number of iter
ations 200 reached. Increase it to improve convergence.
  warnings.warn(
/Users/ryan/opt/anaconda3/lib/python3.9/site-packages/sklearn/deco
mposition/_nmf.py:1637: ConvergenceWarning: Maximum number of iter
ations 200 reached. Increase it to improve convergence.
  warnings.warn(
/Users/ryan/opt/anaconda3/lib/python3.9/site-packages/sklearn/deco
mposition/_nmf.py:1637: ConvergenceWarning: Maximum number of iter
ations 200 reached. Increase it to improve convergence.
  warnings.warn(
/Users/ryan/opt/anaconda3/lib/python3.9/site-packages/sklearn/deco
mposition/_nmf.py:1637: ConvergenceWarning: Maximum number of iter
ations 200 reached. Increase it to improve convergence.
  warnings.warn(

NMF
[0.02799034702971921, 0.3132502923122217, 0.2673010122683917, 0.16
880957127934254, 0.18146262738358176, 0.05043671469361167]
[0.03101686964391566, 0.3516077645149955, 0.3542070001268461, 0.24
516322594036472, 0.2936876034600391, 0.08040111808325054]
[0.02942599207519214, 0.33132255488991563, 0.30467793752672073, 0.
19994501736542303, 0.22432199626305488, 0.06198770138188071 4]
[0.005921185361535245, 0.12366905003452264, 0.09598777167737527, 0
.03724263062190923, 0.03603421000888785, 0.008419083230890852]
[0.02609919213141662 4, 0.32902553876373025, 0.30207595419282024, 0
.19680606439007128, 0.2211227452502638, 0.05809335283693487]
```




Score
0.2
0.1
0.0
0
50
100
150
200
250
300
Number of components
Homogeneity score
Completeness score
V-measure score
Adjusted Rand score
Adjusted Mutual Information score

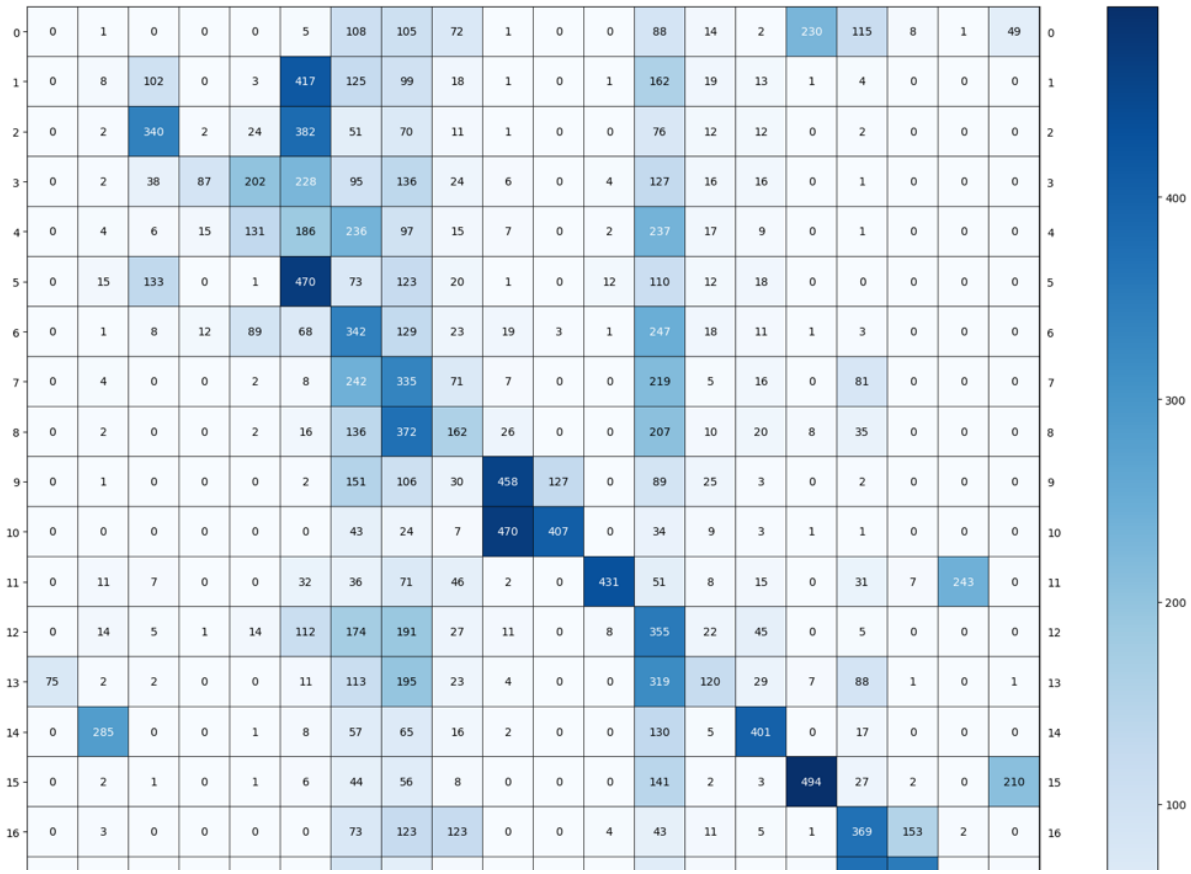

Measure score for NMF
Score
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100
150
200
250
300
Number of components
Homogeneity score
Completeness score
V-measure score
Adjusted Rand Index
Adjusted Mutual Information score

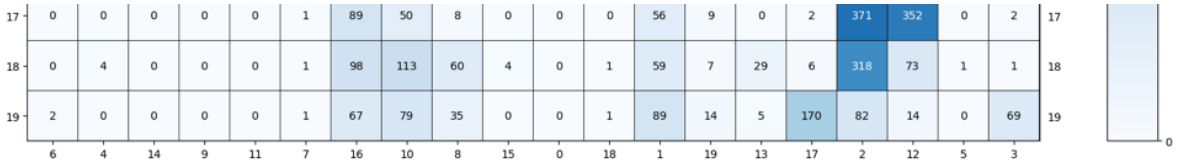
In [17]:

```
#nmf when n_components = 10
from scipy.optimize import linear_sum_assignment
nmf_data_10 = NMF(n_components = 10).fit_transform(X20_train_tfidf)
kmeans = KMeans(n_clusters=20, random_state=0, max_iter=1000, n_ini
y20_pred = kmeans.fit_predict(nmf_data_10)
con_mat20 = contingency_matrix(labels20,y20_pred)
print("NMF Contingency table: \n", con_mat20)

rows, cols = linear_sum_assignment(con_mat20, maximize=True)
plot_mat(con_mat20[rows[:, np.newaxis], cols], xticklabels=cols, yt
print("Homogeneity (NMF, best r): %0.3f" % homogeneity_score(labels
print("Completeness (NMF, best r): %0.3f" % completeness_score(labe
print("V-measure (NMF, best r): %0.3f" % v_measure_score(labels20,
print("Adjusted Rand-Index (NMF, best r): %.3f"% adjusted_rand_scor
print("Adjusted Mutual Information Score (NMF, best r): %.3f"% adju
```

```
/Users/ryan/opt/anaconda3/lib/python3.9/site-packages/sklearn/deco
mposition/_nmf.py:289: FutureWarning: The 'init' value, when 'init
=None' and n_components is less than n_samples and n_features, wil
l be changed from 'nndsvd' to 'nndsvda' in 1.1 (renaming of 0.26).
  warnings.warn(

NMF Contingency table:
 [[  0  88 115  49   1   1   0   5  72   0 105   0   8   2   0   1
108 230
    0  14]
 [  0 162   4   0   8   0   0 417  18   0  99   3   0  13 102   1
125   1
    1  19]
 [  0  76   2   0   2   0   0 382  11   2  70  24   0  12 340   1
51   0
    0  12]
 [  0 127   1   0   2   0   0 228  24  87 136 202   0  16  38   6
95   0
    4  16]
 [  0 237   1   0   4   0   0 186  15  15  97 131   0   9   6   7
236   0
    2  17]
 [  0 110   0   0  15   0   0 470  20   0 123   1   0  18 133   1
73   0
   12  12]
 [  3 247   3   0   1   0   0  68  23  12 129  89   0  11   8  19
342   1
    1  18]
 [  0 219  81   0   4   0   0   8  71   0 335   2   0  16   0   7
242   0
    0   5]
 [  0 207  35   0   2   0   0  16 162   0 372   2   0  20   0  26
136   8
    0  10]
 [127  89   2   0   1   0   0   2  30   0 106   0   0   3   0 458
151   0
    0  25]
 [407  34   1   0   0   0   0   0   7   0  24   0   0   3   0 470
```

```
43   1
    0   9]
 [  0  51  31   0  11 243   0  32  46   0  71   0   7  15   7   2
36   0
  431   8]
 [  0 355   5   0  14   0   0 112  27   1 191  14   0  45   5  11
174   0
    8  22]
 [  0 319  88   1   2   0  75  11  23   0 195   0   1  29   2   4
113   7
    0 120]
 [  0 130  17   0 285   0   0   8  16   0  65   1   0 401   0   2
57   0
    0   5]
 [  0 141  27 210   2   0   0   6   8   0  56   1   2   3   1   0
44 494
    0   2]
 [  0  43 369   0   3   2   0   0 123   0 123   0 153   5   0   0
73   1
    4  11]
 [  0  56 371   2   0   0   0   1   8   0  50   0 352   0   0   0
89   2
    0   9]
 [  0  59 318   1   4   1   0   1  60   0 113   0  73  29   0   4
98   6
    1   7]
 [  0  89  82  69   0   0   2   1  35   0  79   0  14   5   0   0
67 170
    1  14]]
```



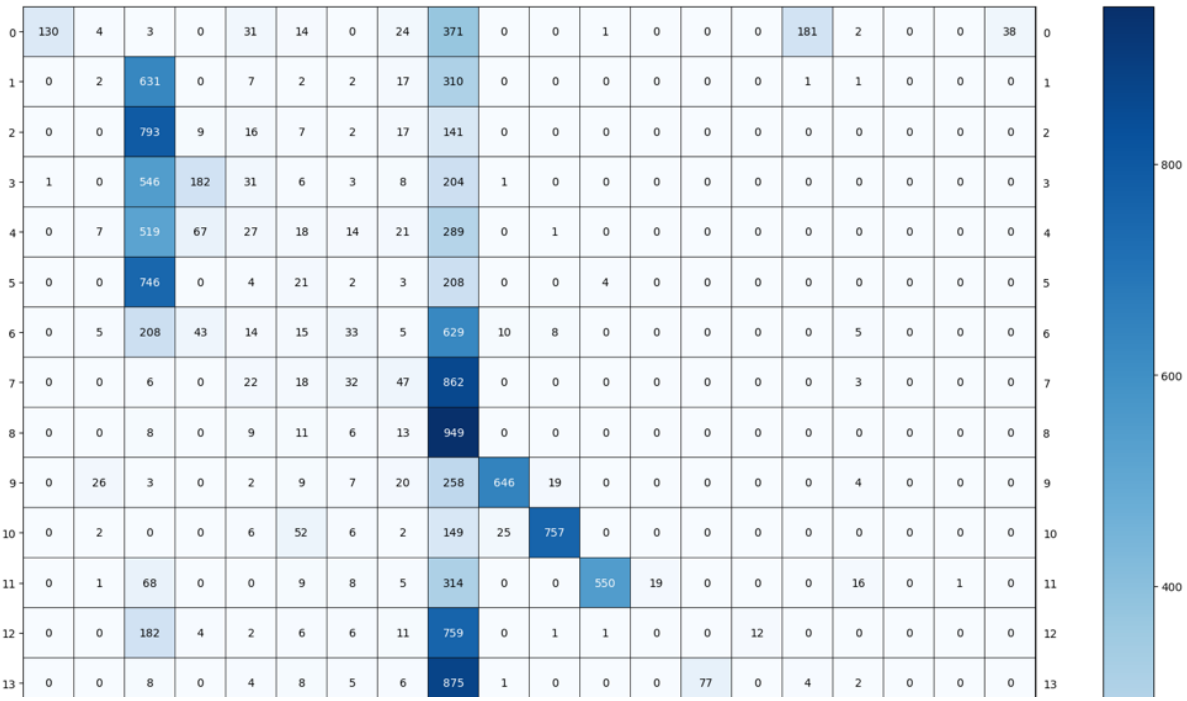


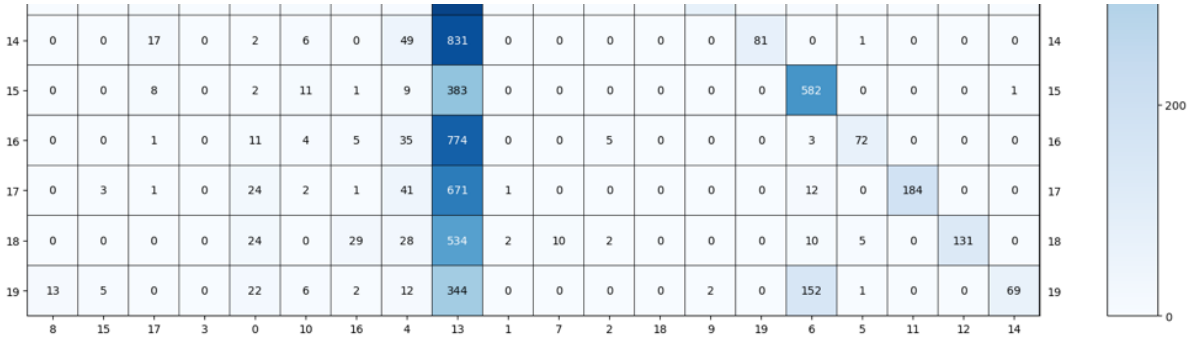

```
Homogeneity (NMF, best r): 0.317
Completeness (NMF, best r): 0.357
V-measure (NMF, best r): 0.335
Adjusted Rand-Index (NMF, best r): 0.124
Adjusted Mutual Information Score (NMF, best r): 0.333
```

In [18]:
```
#svd when n_components = 300
SVD_data_300 = TruncatedSVD(n_components = 300).fit_transform(X20_t
kmeans = KMeans(n_clusters=20, random_state=0, max_iter=1000, n_ini
y20_pred = kmeans.fit_predict(SVD_data_300)
con_mat20 = contingency_matrix(labels20,y20_pred)
print("SVD Contingency table: \n", con_mat20)
#con_mat20 = confusion_matrix(labels20, y20_pred)
rows, cols = linear_sum_assignment(con_mat20, maximize=True)
plot_mat(con_mat20[rows[:, np.newaxis], cols], xticklabels=cols, yt
print("Homogeneity (SVD, best r): %0.3f" % homogeneity_score(labels
print("Completeness (SVD, best r): %0.3f" % completeness_score(labe
print("V-measure (SVD, best r): %0.3f" % v_measure_score(labels20,
print("Adjusted Rand-Index (SVD, best r): %.3f"% adjusted_rand_scor
print("Adjusted Mutual Information Score (SVD, best r): %.3f"% adju
```

```
SVD Contingency table:
 [[ 31   0   1   0  24   2 181   0 130   0  14   0   0 371  38   4
0   3
    0   0]
 [  7   0   0   0  17   1   1   0   0   0   2   0   0 310   0   2
2 631
    0   0]
 [ 16   0   0   9  17   0   0   0   0   0   7   0   0 141   0   0
2 793
    0   0]
 [ 31   1   0 182   8   0   0   0   1   0   6   0   0 204   0   0
3 546
    0   0]
 [ 27   0   0  67  21   0   0   1   0   0  18   0   0 289   0   7
14 519
    0   0]
 [  4   0   4   0   3   0   0   0   0   0  21   0   0 208   0   0
2 746
    0   0]
 [ 14  10   0  43   5   5   0   8   0   0  15   0   0 629   0   5
33 208
    0   0]
 [ 22   0   0   0  47   3   0   0   0   0  18   0   0 862   0   0
32   6
    0   0]
 [  9   0   0   0  13   0   0   0   0   0  11   0   0 949   0   0
6   8
```

```
    0   0]
 [  2 646   0   0  20   4   0  19   0   0   9   0   0 258   0  26
7   3
    0   0]
 [  6  25   0   0   2   0   0 757   0   0  52   0   0 149   0   2
6   0
    0   0]
 [  0   0 550   0   5  16   0   0   0   0   9   0   1 314   0   1
8  68
   19   0]
 [  2   0   1   4  11   0   0   1   0   0   6   0   0 759   0   0
6 182
    0  12]
 [  4   1   0   0   6   2   4   0   0  77   8   0   0 875   0   0
5   8
    0   0]
 [  2   0   0   0  49   1   0   0   0   0   6   0   0 831   0   0
0  17
    0  81]
 [  2   0   0   0   9   0 582   0   0   0  11   0   0 383   1   0
1   8
    0   0]
 [ 11   0   5   0  35  72   3   0   0   0   4   0   0 774   0   0
5   1
    0   0]
 [ 24   1   0   0  41   0  12   0   0   0   2 184   0 671   0   3
1   1
    0   0]
 [ 24   2   2   0  28   5  10  10   0   0   0   0 131 534   0   0
29   0
    0   0]
 [ 22   0   0   0  12   1 152   0  13   2   6   0   0 344  69   5
2   0
    0   0]]
```



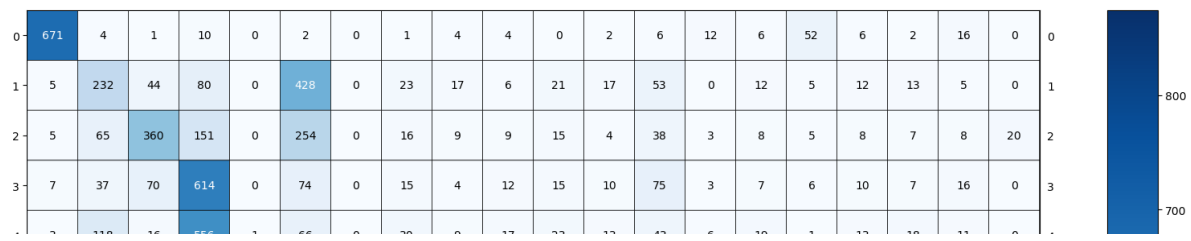


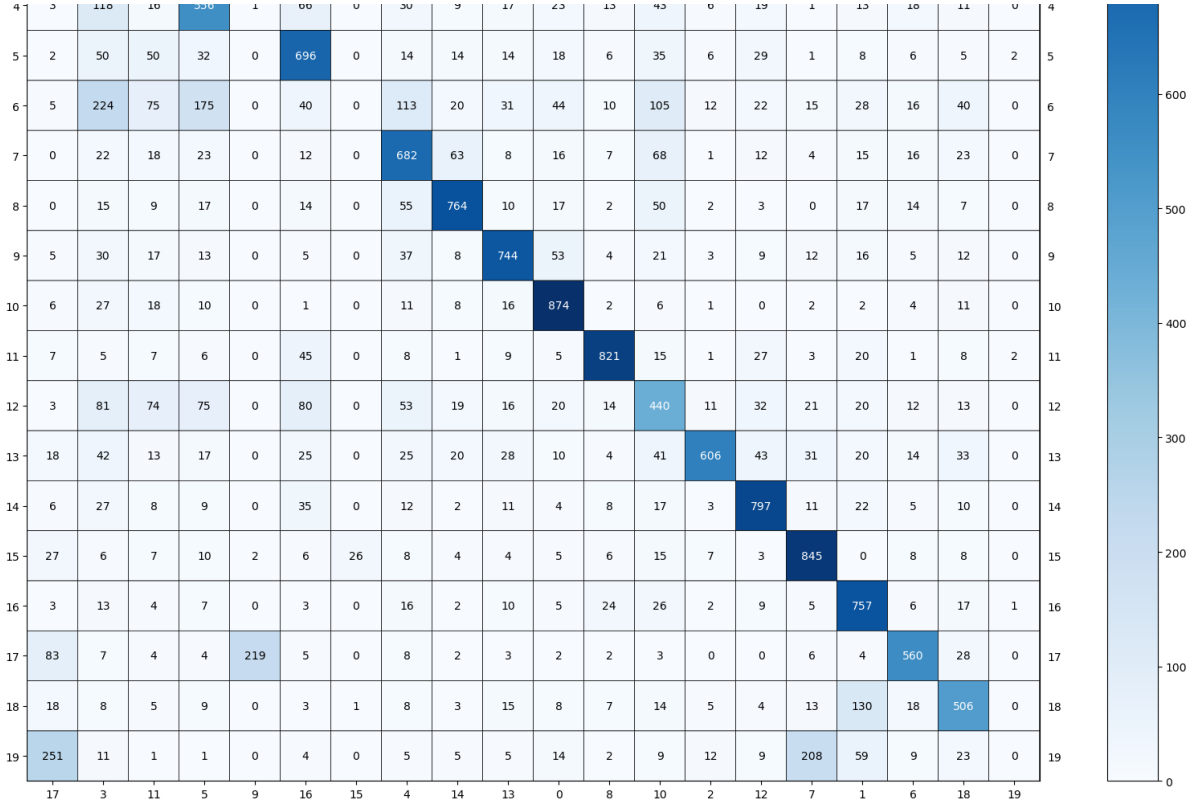

```
Homogeneity (SVD, best r): 0.284
Completeness (SVD, best r): 0.494
V-measure (SVD, best r): 0.361
Adjusted Rand-Index (SVD, best r): 0.076
Adjusted Mutual Information Score (SVD, best r): 0.358
```

Question 11

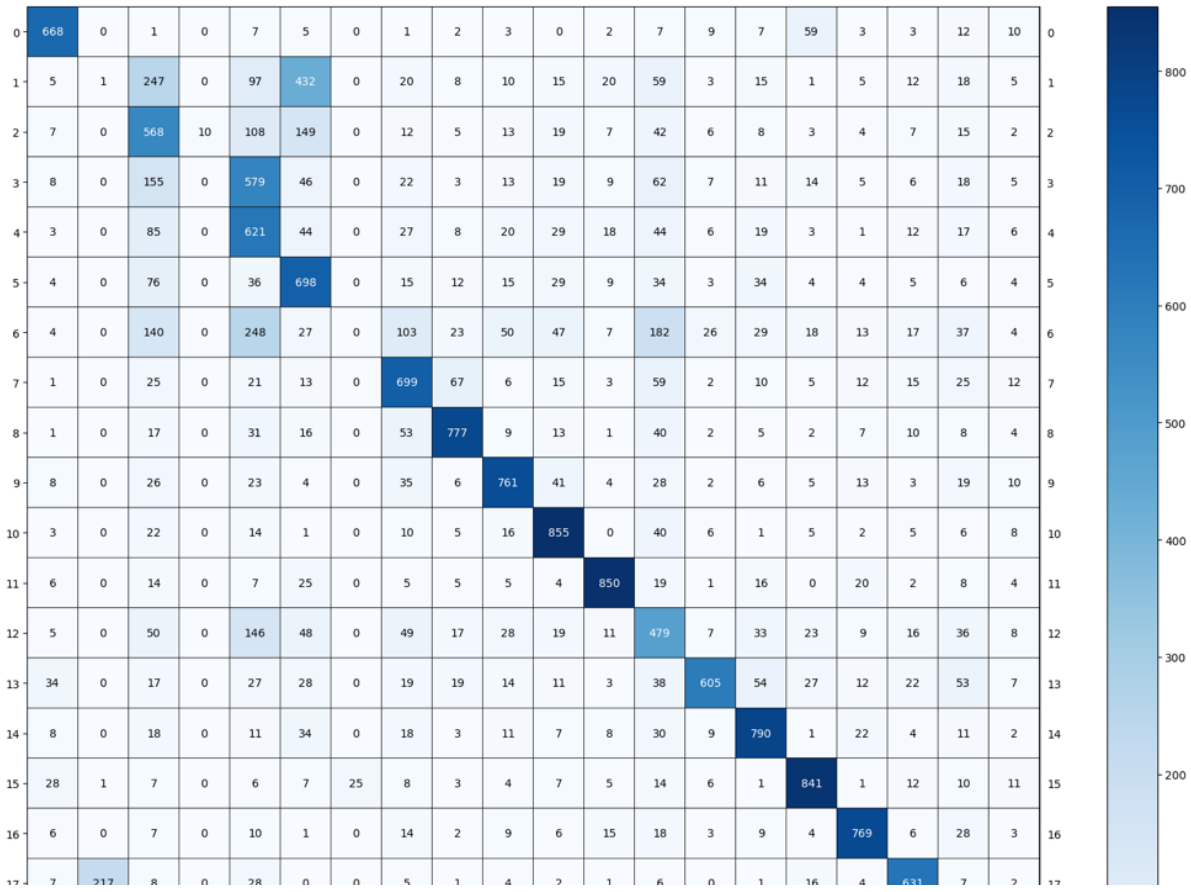
Reduce the dimension of your dataset with UMAP. Consider the following settings: n components = [5, 20, 200], metric = ”cosine” vs. ”euclidean”. If ”cosine” metric fails, please look at the FAQ at the end of this spec. Report the permuted contingency matrix and the five clustering evaluation metrics for the different combinations (6 combinations).

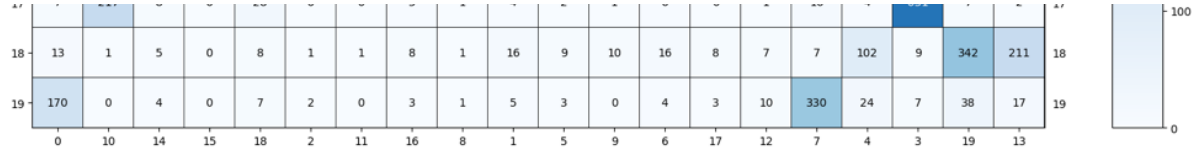
In [26]:
```
#UMAP to reduce dementionality
#!pip install umap-learn

import umap.umap_ as umap
n_components = [5,20,200]
metric = ["cosine", "euclidean"]
for i in n_components:
    for j in metric:
        umap20 = umap.UMAP(n_components=i, metric=j).fit_transform(
        kmeans = KMeans(n_clusters=20, random_state=0, max_iter=100
        y20_pred = kmeans.fit_predict(umap20)
        con_mat20 = contingency_matrix(labels20,y20_pred)
        rows, cols = linear_sum_assignment(con_mat20, maximize=True
        plot_mat(con_mat20[rows[:, np.newaxis], cols], xticklabels=
        print("n_components: ", i, "metric: ", j)
        print("Homogeneity : %0.3f" % homogeneity_score(labels20, y
        print("Completeness : %0.3f" % completeness_score(labels20,
        print("V-measure : %0.3f" % v_measure_score(labels20, y20_p
        print("Adjusted Rand-Index : %.3f"% adjusted_rand_score(lab
        print("Adjusted Mutual Information Score : %.3f"% adjusted_
```



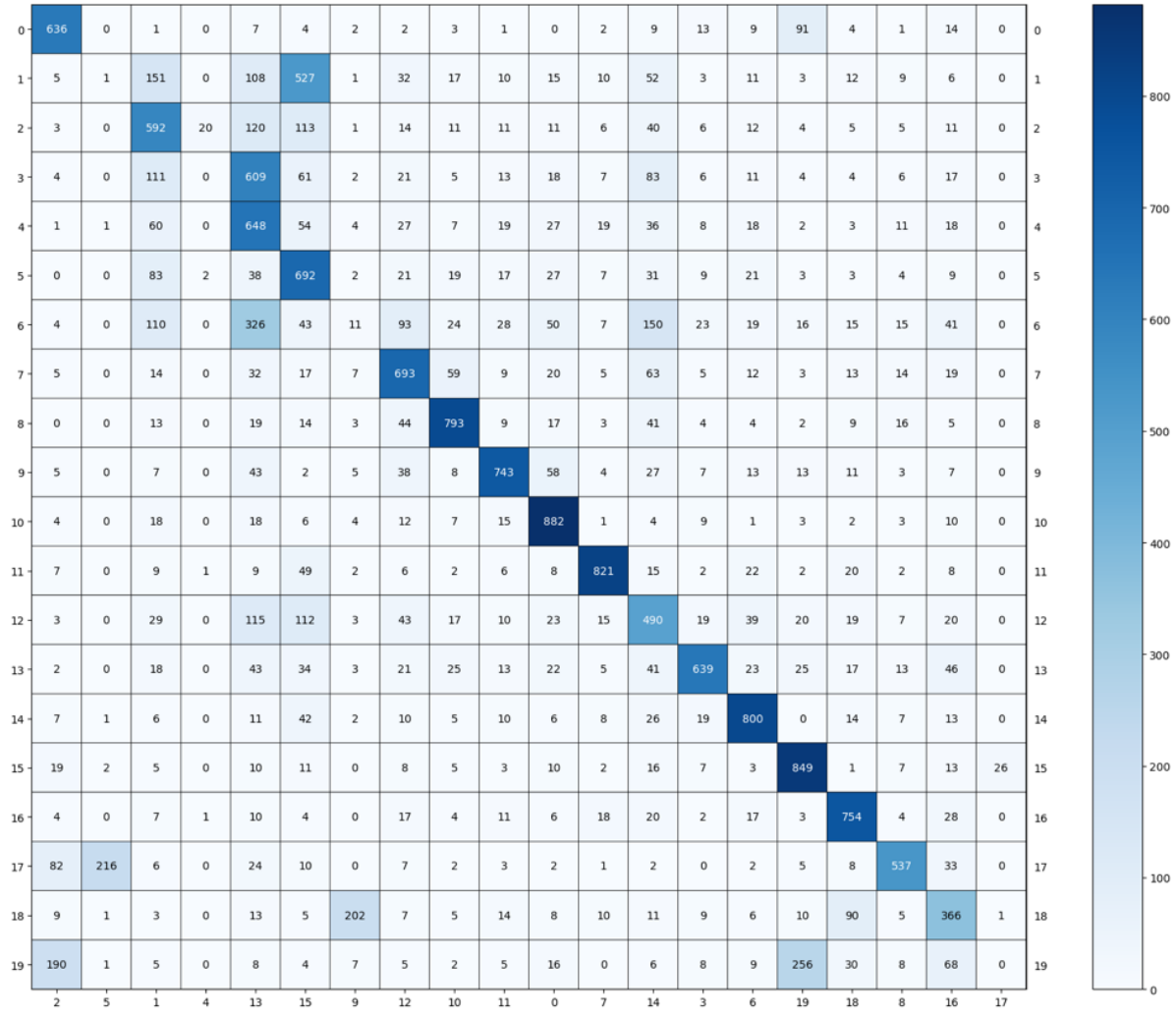


n_components: 5 metric: cosine
Homogeneity : 0.497
Completeness : 0.523
V-measure : 0.509
Adjusted Rand-Index : 0.399
Adjusted Mutual Information Score : 0.508

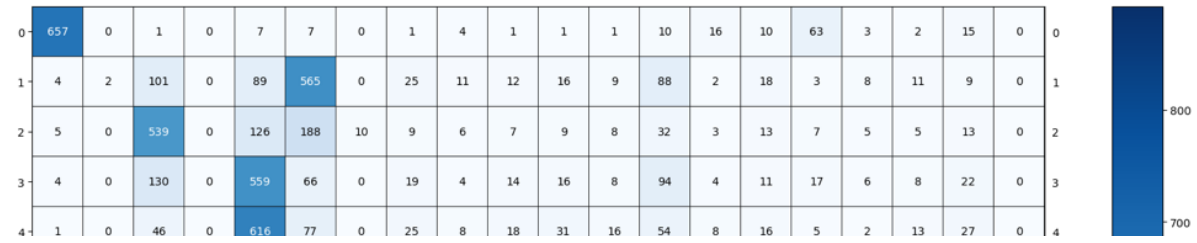


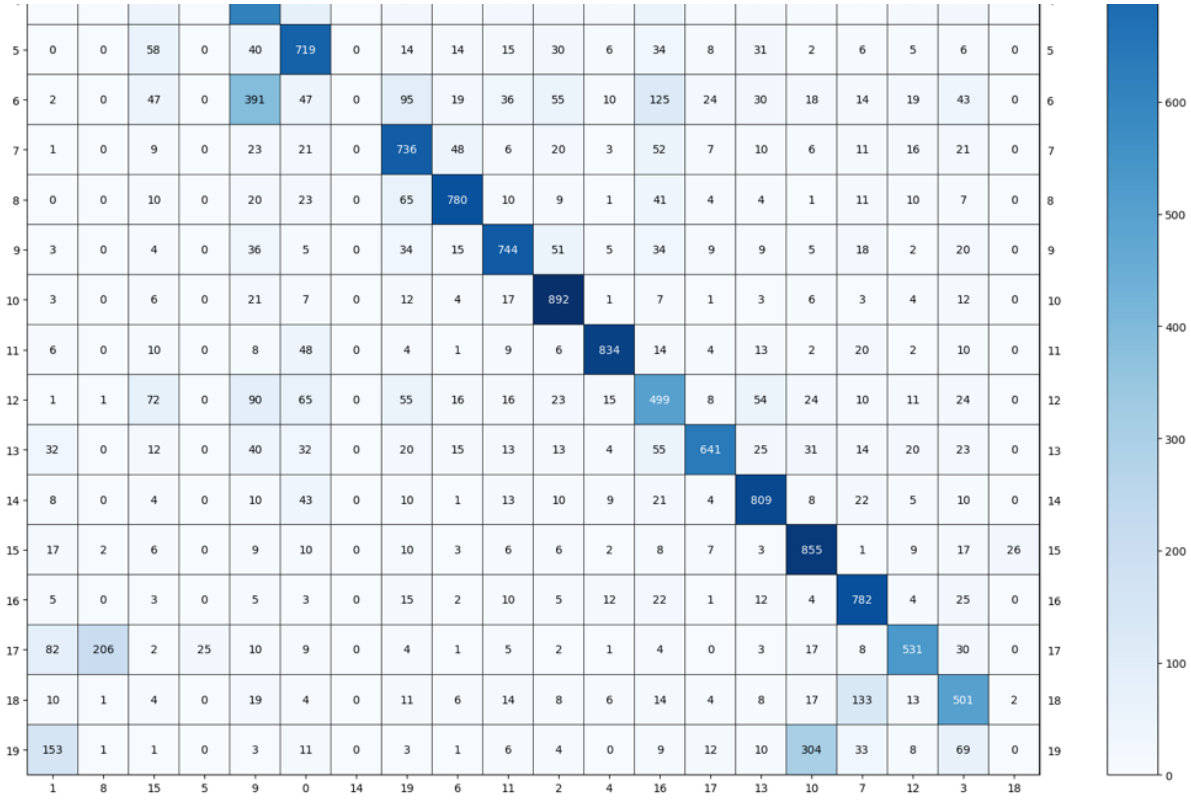


n_components: 5 metric: euclidean
Homogeneity : 0.507
Completeness : 0.539
V-measure : 0.523
Adjusted Rand-Index : 0.406
Adjusted Mutual Information Score : 0.521

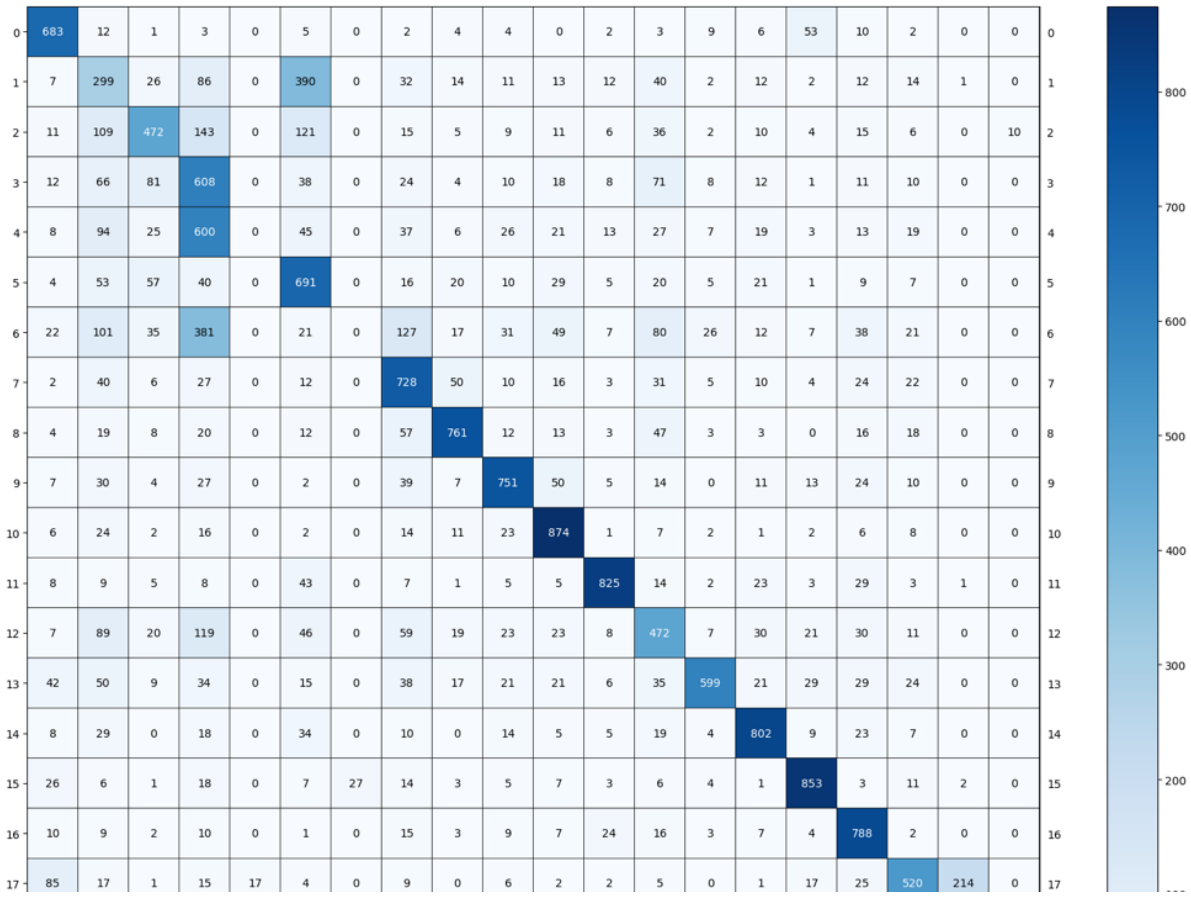


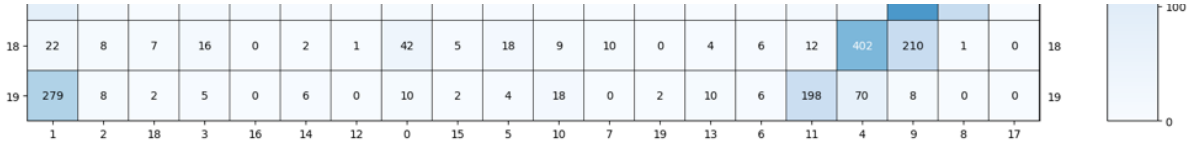
n_components: 20 metric: cosine
Homogeneity : 0.504
Completeness : 0.537
V-measure : 0.520
Adjusted Rand-Index : 0.401
Adjusted Mutual Information Score : 0.518



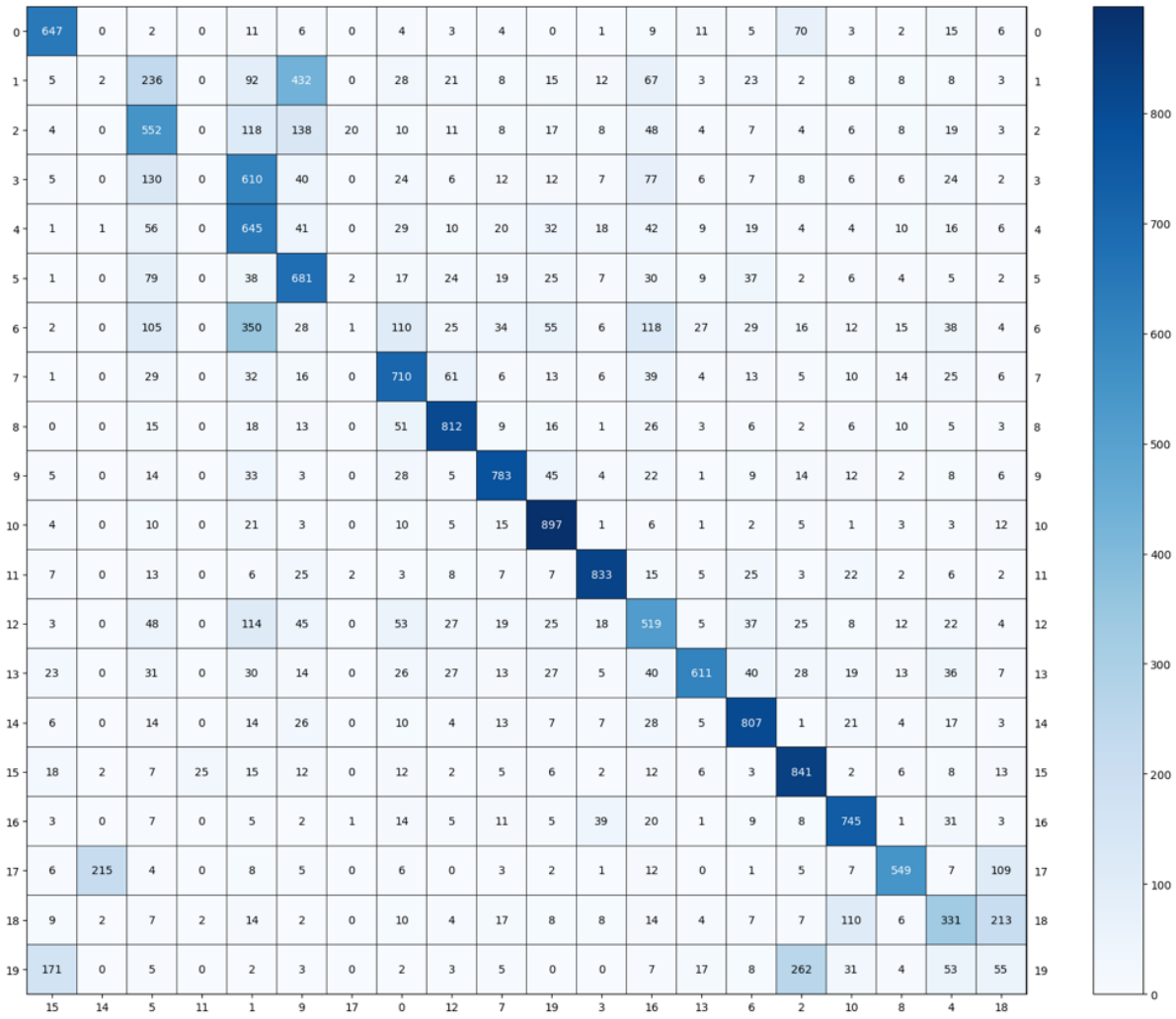


n_components: 20 metric: euclidean
Homogeneity : 0.507
Completeness : 0.547
V-measure : 0.526
Adjusted Rand-Index : 0.408
Adjusted Mutual Information Score : 0.524





n_components: 200 metric: cosine
Homogeneity : 0.492
Completeness : 0.530
V-measure : 0.510
Adjusted Rand-Index : 0.391
Adjusted Mutual Information Score : 0.508



n_components: 200 metric: euclidean
Homogeneity : 0.509
Completeness : 0.540
V-measure : 0.524
Adjusted Rand-Index : 0.409
Adjusted Mutual Information Score : 0.523

QUESTION 13:

So far, we have attempted K-Means clustering with 4 different representation learning techniques (sparse TF-IDF representation, PCA-reduced, NMF-reduced, UMAP-reduced). Compare and contrast the clustering results across the 4 choices, and suggest an approach that is best for the K-Means clustering task on the 20-class text data. Choose any choice of clustering metrics for your comparison.

A: According to the measure score results above, we can observe that it has the best performance on average when using the UMAP reduction using euclidean as metric and set the n_components to 20.

QUESTION 14:

Use UMAP to reduce the dimensionality properly, and perform Agglom- erative clustering with n_clusters=20 . Compare the performance of “ward” and “single” linkage criteria. Report the five clustering evaluation metrics for each case.

In [12]:
```
#BEST n_components = 20, metric = euclidean, linkage = ward
from sklearn.cluster import AgglomerativeClustering
umap20 = umap.UMAP(n_components=20, metric='euclidean').fit_transfo
agg_cluster = AgglomerativeClustering(n_clusters = 20, linkage = 'w
y20_pred = agg_cluster.fit(umap20)
print("n_clusters: 20,  linkage: ward")
print("Homogeneity : %0.3f" % homogeneity_score(labels20, y20_pred.
print("Completeness : %0.3f" % completeness_score(labels20, y20_pre
print("V-measure : %0.3f" % v_measure_score(labels20, y20_pred.labe
print("Adjusted Rand-Index : %.3f"% adjusted_rand_score(labels20, y
print("Adjusted Mutual Information Score : %.3f"% adjusted_mutual_i
#BEST n_components = 20, metric = euclidean, linkage = ward
X20_train_tfidf = tfidf_vect.fit_transform(dataset_20.data)
agg_cluster = AgglomerativeClustering(n_clusters = 20, linkage = 's
y20_pred = agg_cluster.fit(umap20)
print("\n""n_clusters: 20,  linkage: single")
print("Homogeneity : %0.3f" % homogeneity_score(labels20, y20_pred.
print("Completeness : %0.3f" % completeness_score(labels20, y20_pre
print("V-measure : %0.3f" % v_measure_score(labels20, y20_pred.labe
print("Adjusted Rand-Index : %.3f"% adjusted_rand_score(labels20, y
print("Adjusted Mutual Information Score : %.3f"% adjusted_mutual_i
```

```
n_clusters: 20,  linkage: ward
Homogeneity : 0.481
Completeness : 0.524
V-measure : 0.502
Adjusted Rand-Index : 0.365
Adjusted Mutual Information Score : 0.500

n_clusters: 20,  linkage: single
Homogeneity : 0.022
Completeness : 0.328
V-measure : 0.042
Adjusted Rand-Index : 0.001
Adjusted Mutual Information Score : 0.036
```

QUESTION 15:

Apply HDBSCAN on UMAP-transformed 20-category data. Use min_cluster_size=100 . Vary the min cluster size among 20, 100, 200 and report your findings in terms of the five clustering evaluation metrics - you will plot the best contingency matrix in the next question. Feel free to try modifying other parameters in HDBSCAN to get better performance.

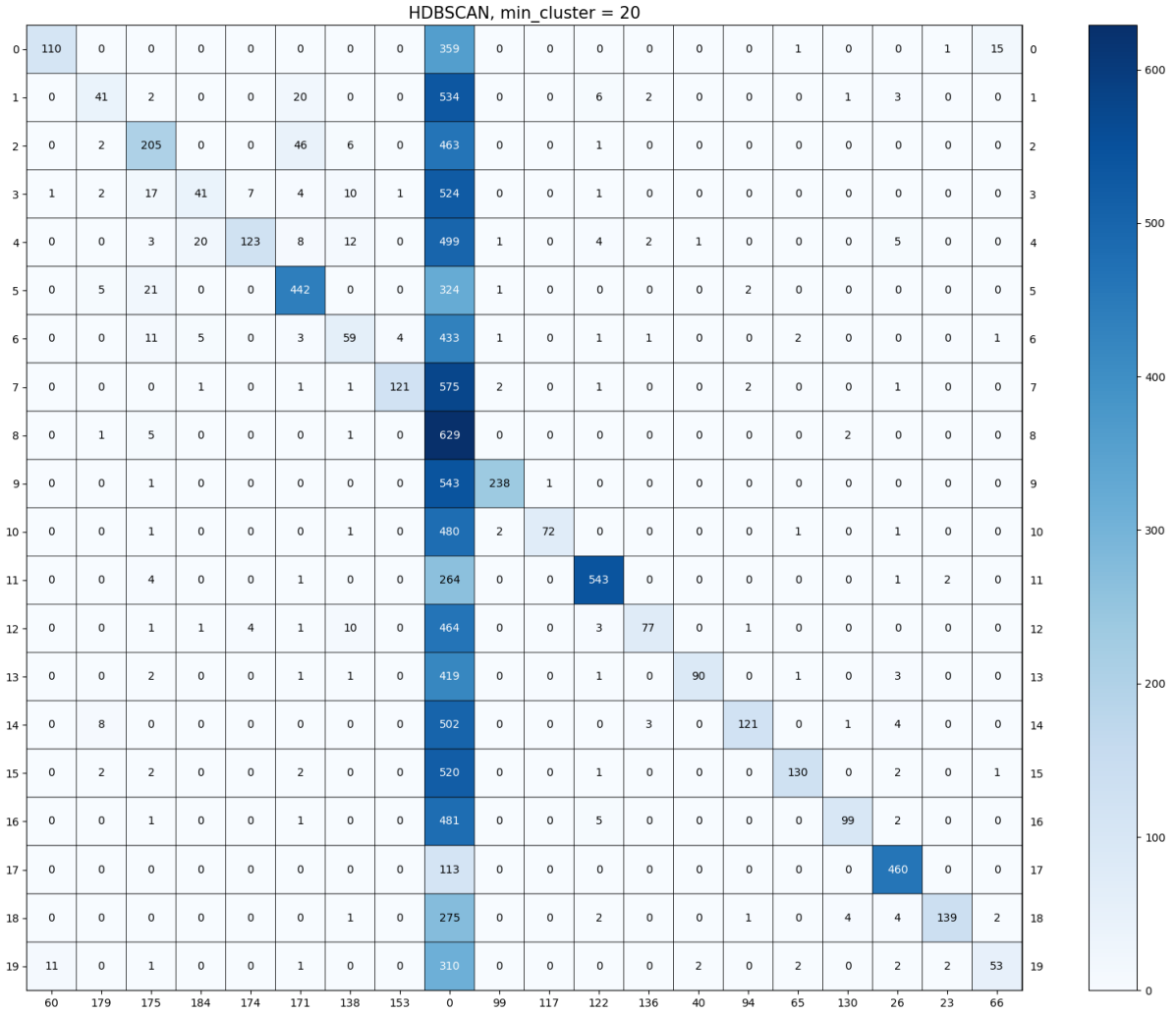
In [13]:
```
import hdbscan
size=[20,100,200]
for i in size:
    umap20 = umap.UMAP(n_components=20, metric='euclidean').fit_tra
    hdb = hdbscan.HDBSCAN(min_cluster_size = i)
    y20_pred = hdb.fit(umap20)
    print("\n""hdbscan,min_cluster_size: ", i)
    print("Homogeneity : %0.3f" % homogeneity_score(labels20, y20_p
    print("Completeness : %0.3f" % completeness_score(labels20, y20
    print("V-measure : %0.3f" % v_measure_score(labels20, y20_pred.
    print("Adjusted Rand-Index : %.3f"% adjusted_rand_score(labels2
    print("Adjusted Mutual Information Score : %.3f"% adjusted_mutu
```

```
hdbscan,min_cluster_size:  20
Homogeneity : 0.422
Completeness : 0.376
V-measure : 0.398
Adjusted Rand-Index : 0.058
Adjusted Mutual Information Score : 0.380

hdbscan,min_cluster_size:  100
Homogeneity : 0.014
Completeness : 0.488
V-measure : 0.027
Adjusted Rand-Index : 0.001
Adjusted Mutual Information Score : 0.026

hdbscan,min_cluster_size:  200
Homogeneity : 0.013
Completeness : 0.455
V-measure : 0.026
Adjusted Rand-Index : 0.001
Adjusted Mutual Information Score : 0.025
```

QUESTION 16:

Contingency matrix Plot the contingency matrix for the best clustering model from Question 15. How many clusters are given by the model? What does “-1” mean for the clustering labels? Interpret the contingency matrix considering the answer to these questions.

In [14]:
```
umap20 = umap.UMAP(n_components=20, metric='euclidean').fit_transfo
hdb = hdbscan.HDBSCAN(min_cluster_size = 20)
y20_pred = hdb.fit(umap20)
con_mat20 = contingency_matrix(labels20,y20_pred.labels_)
rows, cols = linear_sum_assignment(con_mat20, maximize=True)
plot_mat(con_mat20[rows[:, np.newaxis], cols], xticklabels=cols, yt
```



QUESTION 17:

Based on your experiments, which dimensionality reduction technique and clus-tering methods worked best together for 20-class text data and why? Follow the table below. If UMAP takes too long to converge, consider running it once and saving the intermediate results ina pickle file.Hint: DBSCAN and HDBSCAN do not accept the number of clusters as an input parameter. So pay close attention to how the different clustering metrics are being computed for these methods.

*Running the cells below takes too long for every combinations. However I use the testing results of the previous questiond to assume that using dimentionality reduction:umap.UMAP(n_components=200, metric='euclidean') and clustering method:KMeans(n_clusters=20, random_state=0, max_iter=1000, n_init=30) is the best performance. And the following is the result:

Homogeneity : 0.515 Completeness : 0.535 V-measure : 0.525 Adjusted Rand-Index : 0.418 Adjusted Mutual Information Score : 0.523

In [10]:
```
umap200 = umap.UMAP(n_components=200, metric='euclidean').fit_trans
kmeans = KMeans(n_clusters=20, random_state=0, max_iter=1000, n_ini
y20_pred = kmeans.fit_predict(umap200)
print("Homogeneity : %0.3f" % homogeneity_score(labels20, y20_pred)
print("Completeness : %0.3f" % completeness_score(labels20, y20_pre
print("V-measure : %0.3f" % v_measure_score(labels20, y20_pred))
print("Adjusted Rand-Index : %.3f"% adjusted_rand_score(labels20, y
print("Adjusted Mutual Information Score : %.3f"% adjusted_mutual_i
```

```
OMP: Info #276: omp_set_nested routine deprecated, please use omp_
set_max_active_levels instead.

Homogeneity : 0.515
Completeness : 0.535
V-measure : 0.525
Adjusted Rand-Index : 0.418
Adjusted Mutual Information Score : 0.523
```

Normal way but runtime too long

In [9]:
```
import pandas as pd
df = pd.DataFrame(columns = ['Dimentionality Reduction','Clustering
X20_train_tfidf = tfidf_vect.fit_transform(dataset_20.data)
```

In [6]:
```
import numpy as np
import sklearn
import nltk, string
import matplotlib.pyplot as plt
import pandas as pd
import hdbscan
from sklearn.cluster import KMeans
from sklearn.metrics.cluster import contingency_matrix, homogeneity
from sklearn.cluster import AgglomerativeClustering
import umap.umap_ as umap
from sklearn.decomposition import TruncatedSVD, NMF
from sklearn.utils.extmath import randomized_svd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.datasets import fetch_20newsgroups
```

In [ ]:
```
reduce_dim = [TruncatedSVD(n_components=5, random_state=42),
              TruncatedSVD(n_components=20, random_state=42),
              TruncatedSVD(n_components=200, random_state=42),
              umap.UMAP(n_components=5, metric='cosine'),
              umap.UMAP(n_components=20, metric='cosine'),
              umap.UMAP(n_components=200, metric='cosine'),
              NMF(n_components=5, init='random', random_state=42),
              NMF(n_components=20, init='random', random_state=42)
              NMF(n_components=200, init='random', random_state=42

clustering = [KMeans( max_iter=1000, n_clusters=10, n_init=30, rand
               KMeans( max_iter=1000, n_clusters=20, n_init=30, ra
               KMeans( max_iter=1000, n_clusters=50, n_init=30, ra
               AgglomerativeClustering(n_clusters=20),
               hdbscan.HDBSCAN(min_cluster_size=100),
               hdbscan.HDBSCAN(min_cluster_size=200)]
#for none
for j in range(len(clustering)):
        cluster = clustering[j]
        result = cluster.fit_predict(X20_train_tfidf.toarray())
        hom = homogeneity_score(dataset_20.target, result)
        com = completeness_score(dataset_20.target, result)
        measure = v_measure_score(dataset_20.target, result)
        rand = adjusted_rand_score(dataset_20.target, result)
        mutual = adjusted_mutual_info_score(dataset_20.target, resu
        average = 0.2*(hom + com + measure + rand + mutual)
        df.loc[len(df.index)] = ['None',str(clustering[j]), hom, co
        print("test: ", j)
#for other dimensionality reduction
for i in range(3):
    for j in range(len(clustering)):

        reduce = reduce_dim[i]
        red_vec = reduce.fit_transform(X20_train_tfidf)
        cluster = clustering[j]
        result = cluster.fit_predict(red_vec)
        hom = homogeneity_score(dataset_20.target, result)
        com = completeness_score(dataset_20.target, result)
        measure = v_measure_score(dataset_20.target, result)
        rand = adjusted_rand_score(dataset_20.target, result)
        mutual = adjusted_mutual_info_score(dataset_20.target, resu
        average = 0.2*(hom + com + measure + rand + mutual)
        df.loc[len(df.index)] = [str(reduce_dim[i]),str(clustering[
        print("test: ", i , j )
```

QUESTION 18:

Extra credit: If you can find creative ways to further enhance the clustering performance, report your method and the results you obtain.

A: To enhance the performance, I look up online and found a better clustering metthods that is similar to k-means, which is called K-medoids. The k-medoids algorithm is a clustering algorithm related to the k-means algorithm and the medoidshift algorithm. Both the k-means and k-medoids algorithms are partitional (breaking the dataset up into groups). K-means attempts to minimize the total squared error, while k-medoids minimizes the sum of dissimilarities between points labeled to be in a cluster and a point designated as the center of that cluster. In contrast to the k-means algorithm, k-medoids chooses datapoints as centers (medoids or exemplars).It could be more robust to noise and outliers as compared to k-means because it minimizes a sum of general pairwise dissimilarities instead of a sum of squared Euclidean distances.

The following are the results, we may see that using kmeans and k-medoids barely has no difference on performance in this dataset, which is both pretty decent.

In [42]:
```
from sklearn_extra.cluster import KMedoids
umap200 = umap.UMAP(n_components=200, metric='euclidean').fit_trans
kmedoids = KMedoids (n_clusters = 20, random_state=0, max_iter=1000
print("Homogeneity : %0.3f" % homogeneity_score(labels20, kmedoids.
print("Completeness : %0.3f" % completeness_score(labels20, kmedoid
print("V-measure : %0.3f" % v_measure_score(labels20, kmedoids.labe
print("Adjusted Rand-Index : %.3f"% adjusted_rand_score(labels20, k
print("Adjusted Mutual Information Score : %.3f"% adjusted_mutual_i
```

```
Homogeneity : 0.501
Completeness : 0.526
V-measure : 0.513
Adjusted Rand-Index : 0.411
Adjusted Mutual Information Score : 0.511
```

PART 2

In [24]:
```
import torch
import torch.nn as nn
from torchvision import transforms, datasets
from torch.utils.data import DataLoader, TensorDataset
import numpy as np
import matplotlib.pyplot as plt

from tqdm import tqdm
import requests
import os
import tarfile

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix, adjusted_rand_score,
from sklearn.pipeline import Pipeline
from sklearn.base import TransformerMixin
```

Type *Markdown* and LaTeX: α^2

In [25]:
```
filename = './flowers_features_and_labels.npz'

if os.path.exists(filename):
    file = np.load(filename)
    f_all, y_all = file['f_all'], file['y_all']

else:
    if not os.path.exists('./flower_photos'):
        # download the flowers dataset and extract its images
        url = 'http://download.tensorflow.org/example_images/flower
        with open('./flower_photos.tgz', 'wb') as file:
            file.write(requests.get(url).content)
        with tarfile.open('./flower_photos.tgz') as file:
            file.extractall('./')
        os.remove('./flower_photos.tgz')

    class FeatureExtractor(nn.Module):
        def __init__(self):
            super().__init__()

            vgg = torch.hub.load('pytorch/vision:v0.10.0', 'vgg16',

            # Extract VGG-16 Feature Layers
            self.features = list(vgg.features)
            self.features = nn.Sequential(*self.features)
            # Extract VGG-16 Average Pooling Layer
```

```
            self.pooling = vgg.avgpool
            # Convert the image into one-dimensional vector
            self.flatten = nn.Flatten()
            # Extract the first part of fully-connected layer from
            self.fc = vgg.classifier[0]

        def forward(self, x):
            # It will take the input 'x' until it returns the featu
            out = self.features(x)
            out = self.pooling(out)
            out = self.flatten(out)
            out = self.fc(out)
            return out

    # Initialize the model
    assert torch.cuda.is_available()
    feature_extractor = FeatureExtractor().cuda().eval()

    dataset = datasets.ImageFolder(root='./flower_photos',
                                   transform=transforms.Compose([tr
                                                                 tr
                                                                 tr
                                                                 tr
    dataloader = DataLoader(dataset, batch_size=64, shuffle=True)

    # Extract features and store them on disk
    f_all, y_all = np.zeros((0, 4096)), np.zeros((0,))
    for x, y in tqdm(dataloader):
        with torch.no_grad():
            f_all = np.vstack([f_all, feature_extractor(x.cuda()).c
            y_all = np.concatenate([y_all, y])
    np.savez(filename, f_all=f_all, y_all=y_all)
```

QUESTION 19:

In a brief paragraph discuss: If the VGG network is trained on a dataset with perhaps totally different classes as targets, why would one expect the features derived from such a network to have discriminative power for a custom dataset?

A: We use the pre-trained model's architecture to create a new dataset from our input images in this approach. We'll import the Convolutional and Pooling layers but leave out the "top portion" of the model (the Fully-Connected layer).Recall that VGG16 has been trained on millions of images. Its convolutional layers and trained weights can detect generic features. In other words, we use the patterns that the NN found to be useful to classify images of a given problem to classify a completely different problem without retraining that part of the network.

QUESTION 20:

In a brief paragraph explain how the helper code base is performing feature extraction.

A: The helper code uses a pretrained VGG16 modeled. Where VGG16 refers to a VGG model with 16 weight layers. It first resize the image into 224 x 224 for VGG16 to extract features. The steps of extracting features includes extract VGG-16 Feature Layers, extract VGG-16 Average Pooling Layer, converting the image into one-dimensional vector, then extract the first part of fully-connected layer from VGG16.

QUESTION 21:

How many pixels are there in the original images? How many features does the VGG network extract per image; i.e what is the dimension of each feature vector for an image sample?

A: There pixels in the original images are not identical, however the images were resize into the same shape 224 x 224 . And there are 4096 features extracted per image with the VGG network.

In [3]:
```
print(f_all.shape, y_all.shape)
num_features = f_all.shape[1]
```

```
(3670, 4096) (3670,)
```

QUESTION 22:

Are the extracted features dense or sparse? (Compare with sparse TF-IDF features in text.)

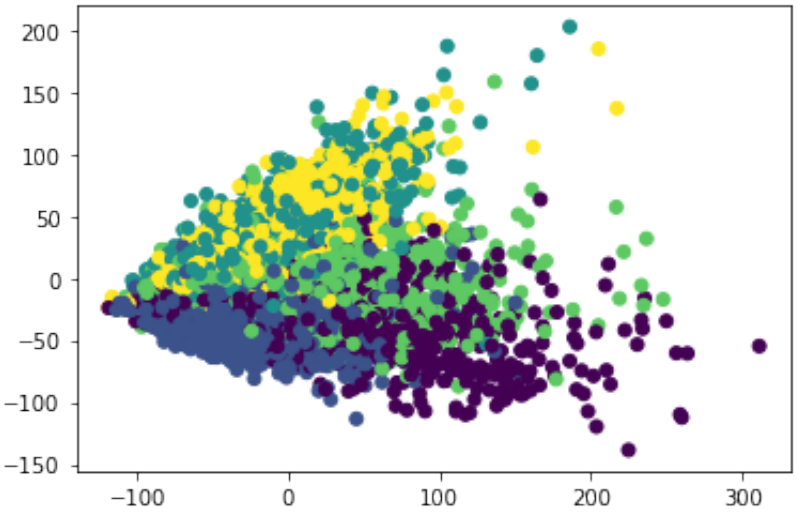
A: Comparing to the sparse TF-IDF features, the extracted feature is dense since there are no nonzero in the entities.

In [4]:
```
f_all.size - np.count_nonzero(f_all)
```

Out[4]:
```
0
```

In [5]:
```
f_pca = PCA(n_components=2).fit_transform(f_all)
plt.scatter(*f_pca.T, c=y_all)
```

Out[5]: <matplotlib.collections.PathCollection at 0x7f27d6bccf40>



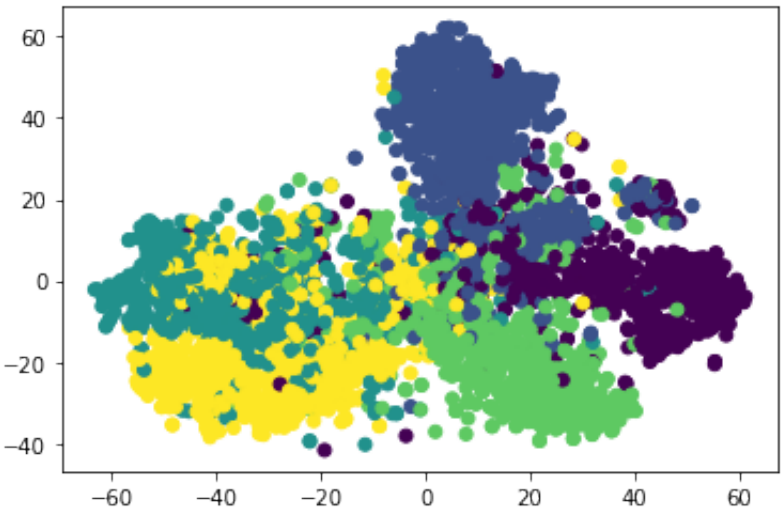
QUESTION 23:

In order to inspect the high-dimensional features, t-SNE is a popular off-the-shelf choice for visualizing Vision features. Map the features you have extracted onto 2 dimensions with t-SNE. Then plot the mapped feature vectors along x and y axes. Color-code the data points with ground-truth labels. Describe your observation.

A: According to the plot using t-SNE below, we can see that the digits are more clearly clustered in their own sub groups . If we used a clustering algorithm to pick out the separate clusters, we could probably quite accurately assign new points to a label. This is already a significant improvement over the PCA visualization we used earlier.

In [6]:
```
from sklearn.manifold import TSNE
f_tsne = TSNE(n_components=2, learning_rate='auto', init='random').
plt.scatter(*f_tsne.T, c=y_all)
```

Out[6]:
```
<matplotlib.collections.PathCollection at 0x7f27d6107b80>
```



MLP Classifier

In [13]:
```
class MLP(torch.nn.Module):
    def __init__(self, num_features):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(num_features, 1280),
            nn.ReLU(True),
            nn.Linear(1280, 640),
            nn.ReLU(True),
            nn.Linear(640, 5),
            nn.LogSoftmax(dim=1)
        )
        self.cuda()


    def forward(self, X):
        return self.model(X)

    def train(self, X, y):
        X = torch.tensor(X, dtype=torch.float32, device='cuda')
        y = torch.tensor(y, dtype=torch.int64, device='cuda')

        self.model.train()

        criterion = nn.NLLLoss()
        optimizer = torch.optim.Adam(self.parameters(), lr=1e-3, we

        dataset = TensorDataset(X, y)
```

```
        dataloader = DataLoader(dataset, batch_size=128, shuffle=Tr

        for epoch in tqdm(range(100)):
            for (X_, y_) in dataloader:

              optimizer.zero_grad()
              outputs = self(X_)

              loss = criterion(outputs, y_)
              loss.backward()
              optimizer.step()

        return self

    def eval(self, X_test, y_test):
        ###################################
        # you should implement this part #
        ###################################
        X = torch.tensor(X_test, dtype=torch.float32, device='cuda'
        y = torch.tensor(y_test, dtype=torch.int64, device='cuda')

        criterion = nn.NLLLoss()
        dataset = TensorDataset(X, y)

        dataset = TensorDataset(X, y)
        dataloader = DataLoader(dataset, batch_size=100, shuffle=Tr

        total = 0
        corr = 0

        with torch.no_grad():
          for (X,y) in dataloader :
            output = self.model(X)
            _,pred = torch.max(output,1)
            total += y.size(0)
            corr  += (pred==y).sum().item()

        accuracy = (corr/total)*100

        return accuracy
```

Autoencoder

In [22]:

```
class Autoencoder(torch.nn.Module, TransformerMixin):
    def __init__(self, n_components):
        super().__init__()
        self.n_components = n_components
        self.n_features = None  # to be determined with data
        self.encoder = None
        self.decoder = None

    def _create_encoder(self):
```

```
    def _create_encoder(self):
        return nn.Sequential(
            nn.Linear(4096, 1280),
            nn.ReLU(True),
            nn.Linear(1280, 640),
            nn.ReLU(True), nn.Linear(640, 120), nn.ReLU(True), nn.L

    def _create_decoder(self):
        return nn.Sequential(
            nn.Linear(self.n_components, 120),
            nn.ReLU(True),
            nn.Linear(120, 640),
            nn.ReLU(True),
            nn.Linear(640, 1280),
            nn.ReLU(True), nn.Linear(1280, 4096))

    def forward(self, X):
        encoded = self.encoder(X)
        decoded = self.decoder(encoded)
        return decoded

    def fit(self, X):
        X = torch.tensor(X, dtype=torch.float32, device='cuda')
        self.n_features = X.shape[1]
        self.encoder = self._create_encoder()
        self.decoder = self._create_decoder()
        self.cuda()
        self.train()

        criterion = nn.MSELoss()
        optimizer = torch.optim.Adam(self.parameters(), lr=1e-3, we

        dataset = TensorDataset(X)
        dataloader = DataLoader(dataset, batch_size=128, shuffle=Tr

        for epoch in tqdm(range(100)):
            for (X_,) in dataloader:
                X_ = X_.cuda()
                # ===================forward=====================
                output = self(X_)
                loss = criterion(output, X_)
                # ===================backward====================
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

        return self

    def transform(self, X):
        X = torch.tensor(X, dtype=torch.float32, device='cuda')
        self.eval()
        with torch.no_grad():
            return self.encoder(X).cpu().numpy()
```

In [26]:
```
from sklearn.cluster import AgglomerativeClustering
from sklearn.decomposition import TruncatedSVD, NMF
from sklearn.utils.extmath import randomized_svd
!pip install umap
!pip install umap-learn
import umap.umap_ as umap
!pip install hdbscan
import hdbscan
import pandas as pd
```

```
Looking in indexes: https://pypi.org/simple,
(https://pypi.org/simple,) https://us-python.pkg.dev/colab-wheels/
public/simple/ (https://us-python.pkg.dev/colab-wheels/public/simp
le/)
Requirement already satisfied: umap in /usr/local/lib/python3.8/di
st-packages (0.1.1)
Looking in indexes: https://pypi.org/simple,
(https://pypi.org/simple,) https://us-python.pkg.dev/colab-wheels/
public/simple/ (https://us-python.pkg.dev/colab-wheels/public/simp
le/)
Requirement already satisfied: umap-learn in /usr/local/lib/python
3.8/dist-packages (0.5.3)
Requirement already satisfied: pynndescent>=0.5 in /usr/local/lib/
python3.8/dist-packages (from umap-learn) (0.5.8)
Requirement already satisfied: numba>=0.49 in /usr/local/lib/pytho
n3.8/dist-packages (from umap-learn) (0.56.4)
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/li
b/python3.8/dist-packages (from umap-learn) (1.0.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.8/di
st-packages (from umap-learn) (4.64.1)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python
3.8/dist-packages (from umap-learn) (1.7.3)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/pytho
n3.8/dist-packages (from umap-learn) (1.21.6)
Requirement already satisfied: llvmlite<0.40,>=0.39.0dev0 in /usr/
local/lib/python3.8/dist-packages (from numba>=0.49->umap-learn) (
0.39.1)
Requirement already satisfied: importlib-metadata in /usr/local/li
b/python3.8/dist-packages (from numba>=0.49->umap-learn) (6.0.0)
Requirement already satisfied: setuptools in /usr/local/lib/python
3.8/dist-packages (from numba>=0.49->umap-learn) (57.4.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/pyth
on3.8/dist-packages (from pynndescent>=0.5->umap-learn) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/
lib/python3.8/dist-packages (from scikit-learn>=0.22->umap-learn)
(3.1.0)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3
.8/dist-packages (from importlib-metadata->numba>=0.49->umap-learn
) (3.12.0)
Looking in indexes: https://pypi.org/simple,
(https://pypi.org/simple,) https://us-python.pkg.dev/colab-wheels/
public/simple/ (https://us-python.pkg.dev/colab-wheels/public/simp
```

```
le/)
Requirement already satisfied: hdbscan in /usr/local/lib/python3.8
/dist-packages (0.8.29)
Requirement already satisfied: joblib>=1.0 in /usr/local/lib/pytho
n3.8/dist-packages (from hdbscan) (1.2.0)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python
3.8/dist-packages (from hdbscan) (1.7.3)
Requirement already satisfied: scikit-learn>=0.20 in /usr/local/li
b/python3.8/dist-packages (from hdbscan) (1.0.2)
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/pytho
n3.8/dist-packages (from hdbscan) (1.21.6)
Requirement already satisfied: cython>=0.27 in /usr/local/lib/pyth
on3.8/dist-packages (from hdbscan) (0.29.33)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/
lib/python3.8/dist-packages (from scikit-learn>=0.20->hdbscan) (3.
1.0)
```

#QUESTION 24: Report the best result (in terms of rand score) within the table below. For HDBSCAN, introduce a conservative parameter grid over min cluster size and min samples.

Here we set min_cluster_size and min_samples to 5

A: According to the results below we could find out that the top 3 best results are the followings

1.UMAP(metric='cosine', n_components=50), KMeans(max_iter=1000, n_clusters=5, n_init=30, random_state=0)

Adjusted Rand Score: 0.465058606771409 87

2.UMAP(metric='cosine', n_components=50), AgglomerativeClustering(n_clusters=5)

Adjusted Rand Score: 0.4484857156148361

3. Autoencoder(50) AgglomerativeClustering(n_clusters=5)

Adjusted Rand Score: 0.276835

In [12]:

```
df = pd.DataFrame(columns = ['Dimentionality Reduction','Clustering
reduce_dim = [TruncatedSVD(n_components=50, random_state=42),
              umap.UMAP(n_components=50, metric='cosine'),
              Autoencoder(50)]

clustering = [KMeans( max_iter=1000, n_clusters=5, n_init=30, rando
               AgglomerativeClustering(n_clusters=5),
               hdbscan.HDBSCAN(min_cluster_size = 5,min_samples =
for j in range(len(clustering)):
        cluster = clustering[j]
        result = cluster.fit_predict(f_all)
        rand = adjusted_rand_score(y_all, result)
        df.loc[len(df.index)] = ['None',str(clustering[j]), rand]
        print("None, "+str(clustering[j]))
        print("Adjusted Rand Score: ", rand)

for i in range(3):
    for j in range(len(clustering)):

        reduce = reduce_dim[i]
        red_vec = reduce.fit_transform(f_all)
        cluster = clustering[j]
        result = cluster.fit_predict(red_vec)
        rand = adjusted_rand_score(y_all, result)
        df.loc[len(df.index)] = [ str(reduce_dim[i]) ,str(clusterin
        print(str(reduce_dim[i])+", "+str(clustering[j]))
        print("Adjusted Rand Score: ", rand)
```

```
None, KMeans(max_iter=1000, n_clusters=5, n_init=30, random_state=
0)
Adjusted Rand Score:  0.19468200056257093
None, AgglomerativeClustering(n_clusters=5)
Adjusted Rand Score:  0.18855278251971858
None, HDBSCAN(min_samples=5)
Adjusted Rand Score:  0.006705947729476718
TruncatedSVD(n_components=50, random_state=42), KMeans(max_iter=10
00, n_clusters=5, n_init=30, random_state=0)
Adjusted Rand Score:  0.19124776241548638
TruncatedSVD(n_components=50, random_state=42), AgglomerativeClust
ering(n_clusters=5)
Adjusted Rand Score:  0.25287434471967624
TruncatedSVD(n_components=50, random_state=42), HDBSCAN(min_sample
s=5)
Adjusted Rand Score:  0.005231761020183455
UMAP(angular_rp_forest=True, metric='cosine', n_components=50, tqd
m_kwds={'bar_format': '{desc}: {percentage:3.0f}%| {bar} {n_fmt}/{
total_fmt} [{elapsed}]', 'desc': 'Epochs completed', 'disable': Tr
ue}), KMeans(max_iter=1000, n_clusters=5, n_init=30, random_state=
0)
Adjusted Rand Score:  0.46505860677140987
UMAP(angular_rp_forest=True, metric='cosine', n_components=50, tqd
m_kwds={'bar_format': '{desc}: {percentage:3.0f}%| {bar} {n_fmt}/{
total_fmt} [{elapsed}]', 'desc': 'Epochs completed', 'disable': Tr
ue}), AgglomerativeClustering(n_clusters=5)
```

```
Adjusted Rand Score:  0.4484857156148361
UMAP(angular_rp_forest=True, metric='cosine', n_components=50, tqd
m_kwds={'bar_format': '{desc}: {percentage:3.0f}%| {bar} {n_fmt}/{
total_fmt} [{elapsed}]', 'desc': 'Epochs completed', 'disable': Tr
ue}), HDBSCAN(min_samples=5)
Adjusted Rand Score:  0.09494009559863244

100%|██████████| 100/100 [00:22<00:00,  4.47it/s]

Autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=4096, out_features=1280, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=1280, out_features=640, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=640, out_features=120, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=120, out_features=50, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=50, out_features=120, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=120, out_features=640, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=640, out_features=1280, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=1280, out_features=4096, bias=True)
  )
), KMeans(max_iter=1000, n_clusters=5, n_init=30, random_state=0)
Adjusted Rand Score:  0.21392776971951044

100%|██████████| 100/100 [00:22<00:00,  4.47it/s]

Autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=4096, out_features=1280, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=1280, out_features=640, bias=True)

    (3): ReLU(inplace=True)
    (4): Linear(in_features=640, out_features=120, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=120, out_features=50, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=50, out_features=120, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=120, out_features=640, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=640, out_features=1280, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=1280, out_features=4096, bias=True)
  )
), AgglomerativeClustering(n_clusters=5)
Adjusted Rand Score:  0.27683501986665715
```

```
Adjusted Rand Score:  [illegible]

100%|██████████| 100/100 [00:22<00:00,  4.46it/s]

Autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=4096, out_features=1280, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=1280, out_features=640, bias=True)

    (3): ReLU(inplace=True)
    (4): Linear(in_features=640, out_features=120, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=120, out_features=50, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=50, out_features=120, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=120, out_features=640, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=640, out_features=1280, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=1280, out_features=4096, bias=True)
  )
), HDBSCAN(min_samples=5)
Adjusted Rand Score:  0.023707912767962632
```

In [15]:
```
df.sort_values(by='Adjusted Rand score',ascending=False)
```

Out[15]:

	Dimentionality Reduction	Clustering	Adjusted Rand score
6	UMAP(angular_rp_forest=True, metric='cosine', ...	KMeans(max_iter=1000, n_clusters=5, n_init=30,...	0.465059
7	UMAP(angular_rp_forest=True, metric='cosine', ...	AgglomerativeClustering(n_clusters=5)	0.448486
10	Autoencoder(\n (encoder): Sequential(\n (0...	AgglomerativeClustering(n_clusters=5)	0.276835
4	TruncatedSVD(n_components=50, random_state=42)	AgglomerativeClustering(n_clusters=5)	0.252874
9	Autoencoder(\n (encoder): Sequential(\n (0...	KMeans(max_iter=1000, n_clusters=5, n_init=30,...	0.213928
0	None	KMeans(max_iter=1000, n_clusters=5, n_init=30,...	0.194682
3	TruncatedSVD(n_components=50, random_state=42)	KMeans(max_iter=1000, n_clusters=5, n_init=30,...	0.191248
1	None	AgglomerativeClustering(n_clusters=5)	0.188553
8	UMAP(angular_rp_forest=True, metric='cosine', ...	HDBSCAN(min_samples=5)	0.094940
11	Autoencoder(\n (encoder): Sequential(\n (0...	HDBSCAN(min_samples=5)	0.023708
2	None	HDBSCAN(min_samples=5)	0.006706
5	TruncatedSVD(n_components=50, random_state=42)	HDBSCAN(min_samples=5)	0.005232

In [19]:
```
f_all[1].shape
```

Out[19]:
```
(4096,)
```

#QUESTION 25: Report the test accuracy of the MLP classifier on the original VGG features. Report the same when using the reduced-dimension features (you have freedom in choosing the dimensionality reduction algorithm and its parameters). Does the performance of the model suffer with the reduced-dimension representations? Is it significant? Does the success in classification make sense in the context of the clustering results obtained for the same features in Question 24.

A: I test on the following three results, which is:

1.MLP on VGG without reduced-dimension, Accurracy: 90.46321525885558

2.MLP on VGG with UMAP (n_components = 50, metric = 'cosine') reduction, Accuracy : 84.87738419618529

3.MLP on VGG with Autoencoder(50) reduction, Accuracy : 87.60217983651226

Base on the results, we can observe that the performance of the model suffer with the reduced-dimension representations. However, it is not very significant with about at most 5% of accuracy. Also, since MLP learns compless features and predicts the classes based on learning, and clustering models predict classes based on distance, it is likely that MLP is better on complex data.

In [27]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(f_all,y_all,test_s
mlp = MLP(f_all.shape[1])
mlp.train(X_train,y_train)
mlp.eval(X_test,y_test)

score = mlp.eval(X_test,y_test)
print("MLP on VGG without reduced-dimension, Accurracy:", score)
#MLP on VGG with UMAP (n_components = 50, metric = 'cosine) reducti
umap = umap.UMAP(n_components=50, metric='cosine')
red_umap = umap.fit_transform(f_all)
X_train,X_test,y_train,y_test = train_test_split(red_umap,y_all,tes
mlp = MLP(red_umap.shape[1])
mlp.train(X_train,y_train)
mlp.eval(X_test,y_test)
score = mlp.eval(X_test,y_test)
print("MLP on VGG with UMAP (n_components = 50, metric = 'cosine) r
#MLP on VGG with Autoencoder(50)
red_auto = Autoencoder(50).fit_transform(f_all)
X_train,X_test,y_train,y_test = train_test_split(red_auto,y_all,tes
mlp = MLP(red_auto.shape[1])
mlp.train(X_train,y_train)
mlp.eval(X_test,y_test)
score = mlp.eval(X_test,y_test)
print("MLP on VGG with Autoencoder(50) reduction, Accuracy :", scor
```

```
100%|██████████| 100/100 [00:14<00:00,  6.79it/s]

MLP on VGG without reduced-dimension, Accurracy: 90.46321525885558

100%|██████████| 100/100 [00:09<00:00, 10.64it/s]

MLP on VGG with UMAP (n_components = 50, metric = 'cosine) reducti
on, Accuracy : 84.87738419618529

100%|██████████| 100/100 [00:24<00:00,  4.14it/s]
100%|██████████| 100/100 [00:06<00:00, 15.83it/s]

MLP on VGG with Autoencoder(50) reduction, Accuracy : 87.602179836
51226
```