Project 2: Data Representations and Clustering

February 12, 2023

0.1 Module Loading

[75]:
```
# modify this line to your local path
path = '/content/drive/MyDrive/Colab Notebooks/ECE219/Project2/'
```

[1]:
```
import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt
import matplotlib
%matplotlib inline
from IPython.core.pylabtools import figsize
import re
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
import string
from string import punctuation
from sklearn.decomposition import TruncatedSVD, NMF
from sklearn.utils.extmath import randomized_svd
from sklearn.metrics import auc, roc_curve, plot_roc_curve,␣
↪plot_confusion_matrix, accuracy_score, recall_score, precision_score,␣
↪f1_score
from sklearn import datasets, metrics, model_selection, svm
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.pipeline import Pipeline
from tempfile import mkdtemp
from joblib import Memory
from sklearn.multiclass import OneVsRestClassifier, OneVsOneClassifier
from sklearn.datasets import fetch_20newsgroups
from sklearn.cluster import KMeans,  AgglomerativeClustering
from sklearn.metrics.cluster import contingency_matrix, homogeneity_score,␣
↪completeness_score, v_measure_score, adjusted_rand_score,␣
↪adjusted_mutual_info_score
from sklearn.metrics import confusion_matrix
from scipy.optimize import linear_sum_assignment
```

```python
!pip install umap-learn
import umap
!pip install hdbscan
import hdbscan
import pickle
import bz2
from sklearn.manifold import TSNE
import seaborn as sns
from sklearn.model_selection import train_test_split
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting umap-learn
  Downloading umap-learn-0.5.3.tar.gz (88 kB)
                           88.2/88.2 KB
9.0 MB/s eta 0:00:00
  Preparing metadata (setup.py) … done
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.8/dist-
packages (from umap-learn) (1.21.6)
Requirement already satisfied: scikit-learn>=0.22 in
/usr/local/lib/python3.8/dist-packages (from umap-learn) (1.0.2)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.8/dist-
packages (from umap-learn) (1.7.3)
Requirement already satisfied: numba>=0.49 in /usr/local/lib/python3.8/dist-
packages (from umap-learn) (0.56.4)
Collecting pynndescent>=0.5
  Downloading pynndescent-0.5.8.tar.gz (1.1 MB)
                           1.1/1.1 MB
38.4 MB/s eta 0:00:00
  Preparing metadata (setup.py) … done
Requirement already satisfied: tqdm in /usr/local/lib/python3.8/dist-packages
(from umap-learn) (4.64.1)
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: setuptools in /usr/local/lib/python3.8/dist-
packages (from numba>=0.49->umap-learn) (57.4.0)
Requirement already satisfied: importlib-metadata in
/usr/local/lib/python3.8/dist-packages (from numba>=0.49->umap-learn) (6.0.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.8/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) (3.12.1)
Building wheels for collected packages: umap-learn, pynndescent
  Building wheel for umap-learn (setup.py) … done
  Created wheel for umap-learn: filename=umap_learn-0.5.3-py3-none-any.whl
size=82829
sha256=d21bdbbbb4d585960b24595bfbf1f1caa6c48d54cd75cdcc6db5e4877d525a38
  Stored in directory: /root/.cache/pip/wheels/a9/3a/67/06a8950e053725912e6a8c42
c4a3a241410f6487b8402542ea
  Building wheel for pynndescent (setup.py) … done
  Created wheel for pynndescent: filename=pynndescent-0.5.8-py3-none-any.whl
size=55513
sha256=7e05b58c086f474870a25f5511f3a656c1146c71fad07984fc699e23077109bf
  Stored in directory: /root/.cache/pip/wheels/1c/63/3a/29954bca1a27ba100ed8c279
73a78cb71b43dc67aed62e80c3
Successfully built umap-learn pynndescent
Installing collected packages: pynndescent, umap-learn
Successfully installed pynndescent-0.5.8 umap-learn-0.5.3
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting hdbscan
  Downloading hdbscan-0.8.29.tar.gz (5.2 MB)
                           5.2/5.2 MB
87.1 MB/s eta 0:00:00
  Installing build dependencies … done
  Getting requirements to build wheel … done
  Preparing metadata (pyproject.toml) … done
Requirement already satisfied: scikit-learn>=0.20 in
/usr/local/lib/python3.8/dist-packages (from hdbscan) (1.0.2)
Requirement already satisfied: joblib>=1.0 in /usr/local/lib/python3.8/dist-
packages (from hdbscan) (1.2.0)
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.8/dist-
packages (from hdbscan) (1.21.6)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.8/dist-
packages (from hdbscan) (1.7.3)
Requirement already satisfied: cython>=0.27 in /usr/local/lib/python3.8/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)
Building wheels for collected packages: hdbscan
  Building wheel for hdbscan (pyproject.toml) … done
  Created wheel for hdbscan: filename=hdbscan-0.8.29-cp38-cp38-linux_x86_64.whl
size=3773945
sha256=c6ba2a629c1df8a860bd5b00c970d289520fc1838accc963271ea407a4c79e4e
  Stored in directory: /root/.cache/pip/wheels/76/06/48/527e038689c581cc9e519c73
840efdc7473805149e55bd7ffd
Successfully built hdbscan
Installing collected packages: hdbscan
Successfully installed hdbscan-0.8.29
```

[2]:
```python
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/MyDrive/Colab Notebooks/ECE219/Project2')
from plotmat import plot_mat
```

```
Mounted at /content/drive
```

[3]:
```python
# helper code
!pip install torch
!pip install torchvision
import torch
import torch.nn as nn
from torchvision import transforms, datasets
from torch.utils.data import DataLoader, TensorDataset
from tqdm import tqdm
import requests
import os
import tarfile
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.base import TransformerMixin
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: torch in /usr/local/lib/python3.8/dist-packages
(1.13.1+cu116)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.8/dist-packages (from torch) (4.4.0)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: torchvision in /usr/local/lib/python3.8/dist-
packages (0.14.1+cu116)
Requirement already satisfied: requests in /usr/local/lib/python3.8/dist-
packages (from torchvision) (2.25.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packages
(from torchvision) (1.21.6)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.8/dist-packages (from torchvision) (4.4.0)
Requirement already satisfied: torch==1.13.1 in /usr/local/lib/python3.8/dist-
packages (from torchvision) (1.13.1+cu116)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/local/lib/python3.8/dist-packages (from torchvision) (7.1.2)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/usr/local/lib/python3.8/dist-packages (from requests->torchvision) (1.24.3)
Requirement already satisfied: chardet<5,>=3.0.2 in
/usr/local/lib/python3.8/dist-packages (from requests->torchvision) (4.0.0)
```

```
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dist-
packages (from requests->torchvision) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.8/dist-packages (from requests->torchvision) (2022.12.7)
```

0.2 Question 1

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

Ans:

the dimensions of the TF-IDF matrix is (7882, 23522).

```
[ ]: categories = ['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']
     dataset = fetch_20newsgroups(subset = 'all', categories = categories, shuffle =␣
      ↪True, random_state = 0,remove=('headers','footers'))

     vec = CountVectorizer(stop_words='english', min_df=3)
     tfidf = TfidfTransformer()

     X_vec = vec.fit_transform(dataset.data)
     X_tfidf = tfidf.fit_transform(X_vec)
     print("TFIDX dataset shape:", X_tfidf.shape)
```

```
TFIDX dataset shape: (7882, 23522)
```

0.3 Question 2

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

Ans:

The contingency table is

3232	671
54	3925

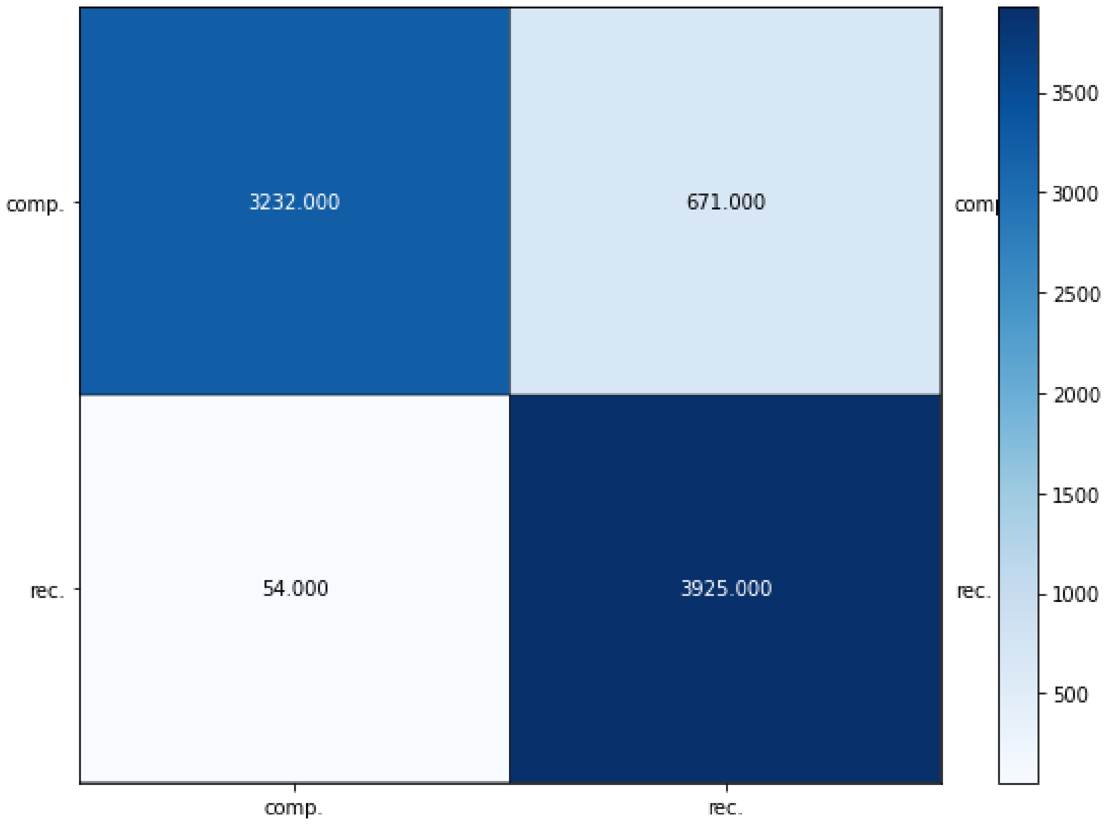
The contingency matrix doesn't have to be square-shaped but it's usually square-shaped. It has normally same rows and columns because the reference data and the map should have same dimensions (categories). However, it's not necessary, a category can exist in the reference data but doesn't exist in the map, and vice versa.

```
[ ]: # map_root = {"comp.graphics":0, "comp.os.ms-windows.misc":0, "comp.sys.ibm.pc.
      ↪hardware":0, "comp.sys.mac.hardware":0,
#              "rec.autos":1, "rec.motorcycles":1, "rec.sport.baseball":1, "rec.
↪sport.hockey":1}
# y_data = dataset.target.map(map_root)
y_data = []
for i in dataset.target:
  if i < 4:
    y_data.append(0)
  else:
    y_data.append(1)
```

[ ]:
```
km = KMeans(n_clusters=2, init='k-means++', max_iter=2000, n_init=50,␣
↪random_state=0)
km.fit(X_tfidf)
```

[ ]:
```
KMeans(max_iter=2000, n_clusters=2, n_init=50, random_state=0)
```

[ ]:
```
plot_mat(contingency_matrix(y_data, km.labels_), size=(8, 6),␣
↪xticklabels=['comp.','rec.'], yticklabels=['comp.','rec.'], pic_fname='Q2.
↪png')
```



0.4 Question 3

Report the 5 clustering measures explained in the introduction for Kmeans clustering.

Ans:

- Homogeneity: 0.5999
- Completeness: 0.6121
- V-measure: 0.6059
- Adjusted Rand-Index: 0.6659
- Adjusted Mutual Information Score: 0.6059

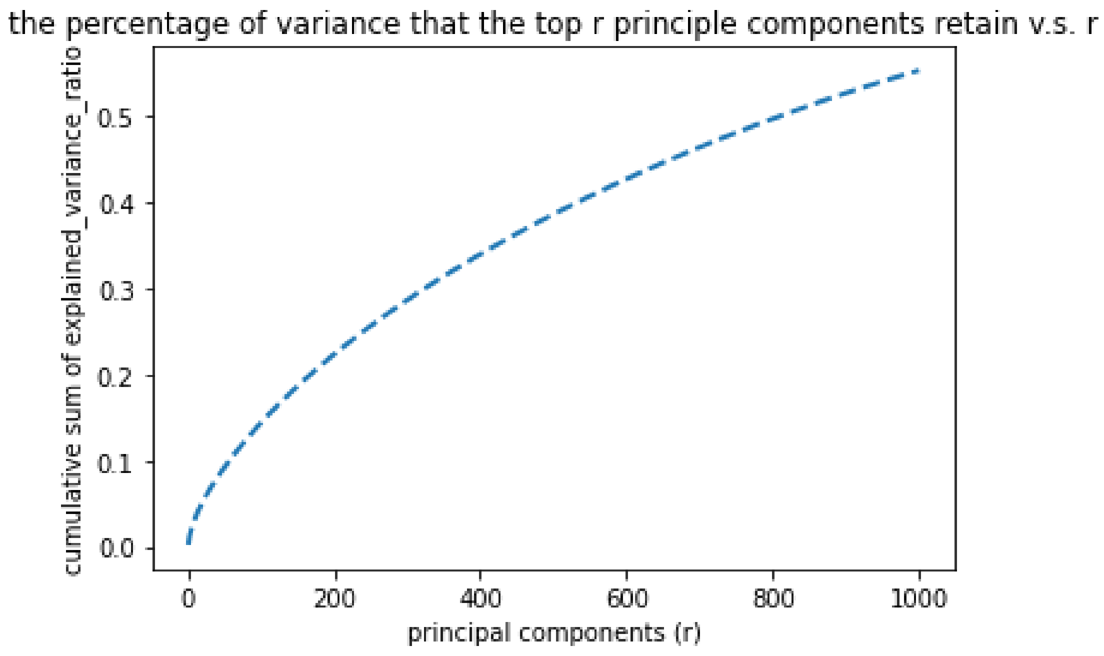
```
[ ]: def print_5_measure_scores(y_data, labels):
       print("Homogeneity: %0.4f" % homogeneity_score(y_data, labels))
       print("Completeness: %0.4f" % completeness_score(y_data, labels))
       print("V-measure: %0.4f" % v_measure_score(y_data, labels))
       print("Adjusted Rand-Index: %0.4f"% adjusted_rand_score(y_data, labels))
       print("Adjusted Mutual Information Score: %0.4f"%␣
      ↪adjusted_mutual_info_score(y_data, labels))
     print_5_measure_scores(y_data, km.labels_)
```

```
Homogeneity: 0.5999
Completeness: 0.6121
V-measure: 0.6059
Adjusted Rand-Index: 0.6659
Adjusted Mutual Information Score: 0.6059
```

0.5 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.

```
[ ]: svd = TruncatedSVD(n_components=1000, random_state=0)
     X_LSI = svd.fit_transform(X_tfidf)

     ratios = np.cumsum(svd.explained_variance_ratio_)
     fig = plt.figure()
     ax = fig.add_subplot(1, 1, 1)
     ax.plot(np.linspace(1, 1000, 1000), ratios, lw=2, linestyle='--')

     ax.set_ylabel('cumulative sum of explained_variance_ratio')
     ax.set_xlabel('principal components (r)')
     plt.title('the percentage of variance that the top r principle components␣
      ↪retain v.s. r')
     plt.show()
```



0.6 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.

```
[ ]: rs5 = [1, 10, 20, 50, 100, 300]
     # svd
     svd_homo = []
     svd_comp = []
     svd_vmes = []
     svd_ranI = []
     svd_muts = []
     # nmf
     nmf_homo = []
     nmf_comp = []
     nmf_vmes = []
     nmf_ranI = []
     nmf_muts = []
     kmm = KMeans(n_clusters=2, init='k-means++', max_iter=2000, n_init=50,␣
      ↪random_state=0)
     for i in range(len(rs5)):
       print("Try r =", rs5[i], "...")
       # SVD
       svd_tmp = TruncatedSVD(n_components=rs5[i], random_state=0)
       X_svd = svd_tmp.fit_transform(X_tfidf)
       km_svd = kmm.fit(X_svd)
```

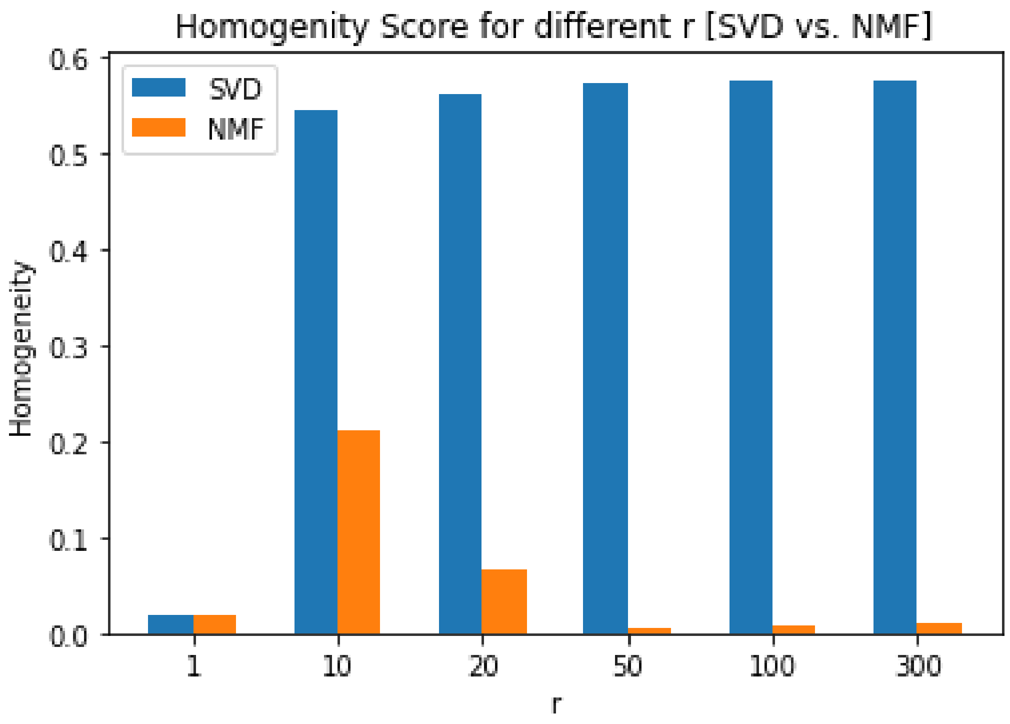
```
    svd_homo.append(homogeneity_score(y_data, km_svd.labels_))
    svd_comp.append(completeness_score(y_data, km_svd.labels_))
    svd_vmes.append(v_measure_score(y_data, km_svd.labels_))
    svd_ranI.append(adjusted_rand_score(y_data, km_svd.labels_))
    svd_muts.append(adjusted_mutual_info_score(y_data, km_svd.labels_))
    # NMF
    nmf_tmp = NMF(n_components=rs5[i], init='random', random_state=0)
    X_nmf = nmf_tmp.fit_transform(X_tfidf)
    km_nmf = kmm.fit(X_nmf)
    nmf_homo.append(homogeneity_score(y_data, km_nmf.labels_))
    nmf_comp.append(completeness_score(y_data, km_nmf.labels_))
    nmf_vmes.append(v_measure_score(y_data, km_nmf.labels_))
    nmf_ranI.append(adjusted_rand_score(y_data, km_nmf.labels_))
    nmf_muts.append(adjusted_mutual_info_score(y_data, km_nmf.labels_))
print('Done')
```

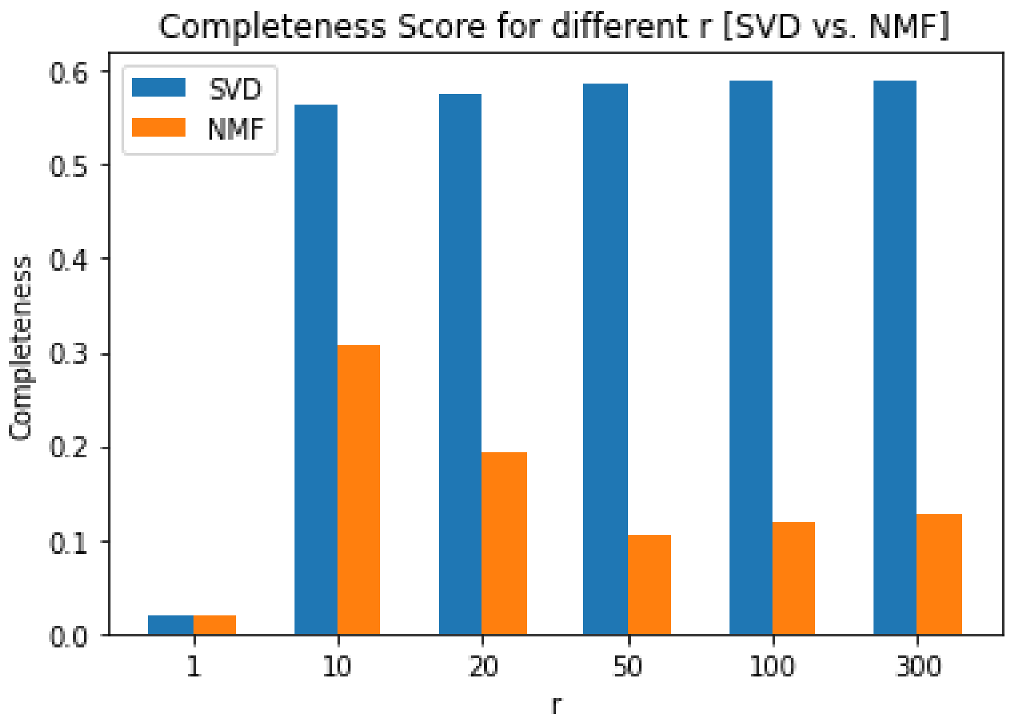
```
Try r = 1 …
Try r = 10 …
Try r = 20 …
Try r = 50 …
Try r = 100 …

/usr/local/lib/python3.8/dist-packages/sklearn/decomposition/_nmf.py:1637:
ConvergenceWarning: Maximum number of iterations 200 reached. Increase it to
improve convergence.
  warnings.warn(

Try r = 300 …
Done
```

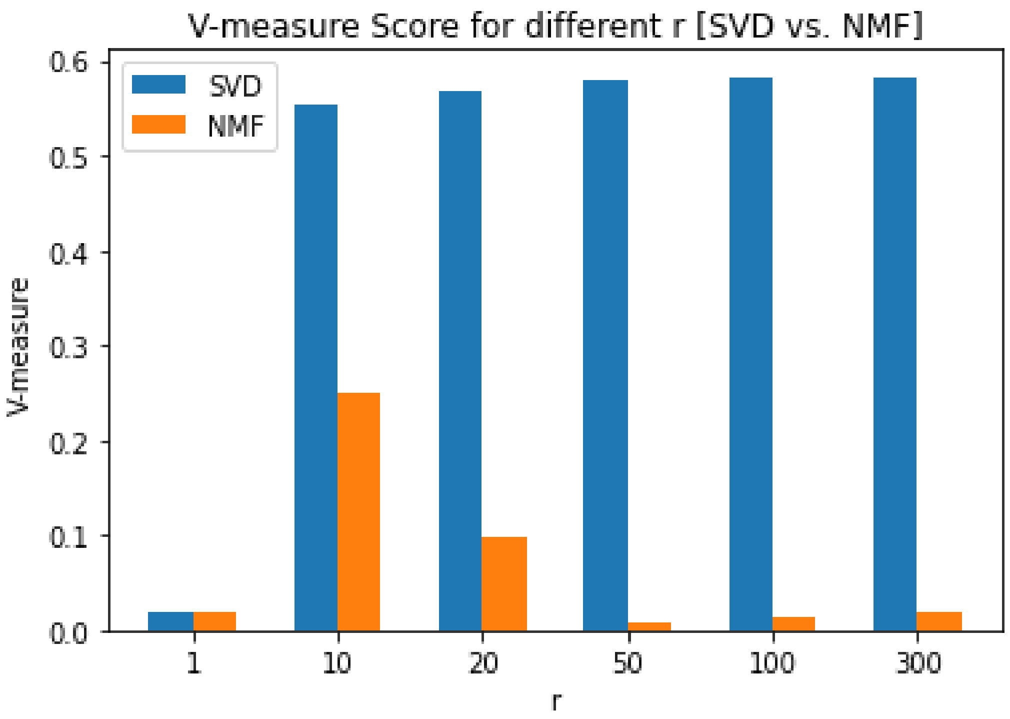
[ ]:
```
def print_bar_scores_result(width, rs, svd_score, nmf_score, score_name, title):
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    # ax.plot(rs, svd_homo, color='r', label="SVD")
    # ax.plot(rs, nmf_homo, color='b', label="NMF")
    ax.bar(np.arange(len(rs)) - width/2, svd_score, width, label="SVD")
    ax.bar(np.arange(len(rs)) + width/2, nmf_score, width, label="NMF")
    ax.set_xticks(np.arange(len(rs)))
    ax.set_xticklabels(rs)

    ax.legend()
    ax.set_ylabel(score_name)
    ax.set_xlabel('r')
    plt.title(title)
    plt.show()
```

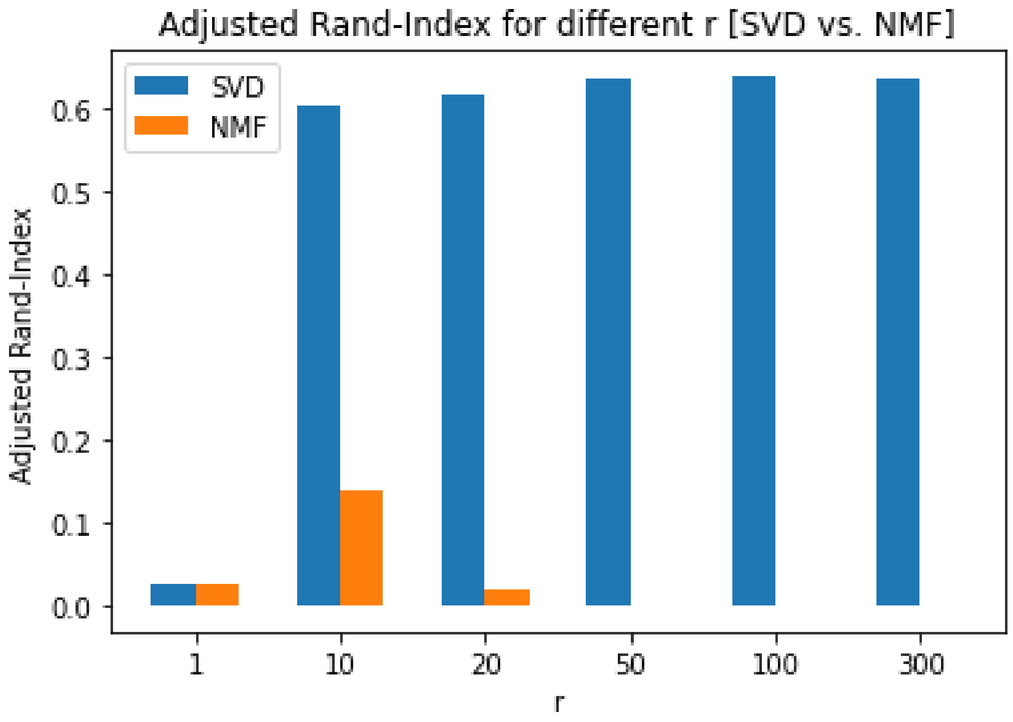
[ ]:
```
print_bar_scores_result(0.3, rs5, svd_homo, nmf_homo, 'Homogeneity',␣
↪'Homogenity Score for different r [SVD vs. NMF]')
```

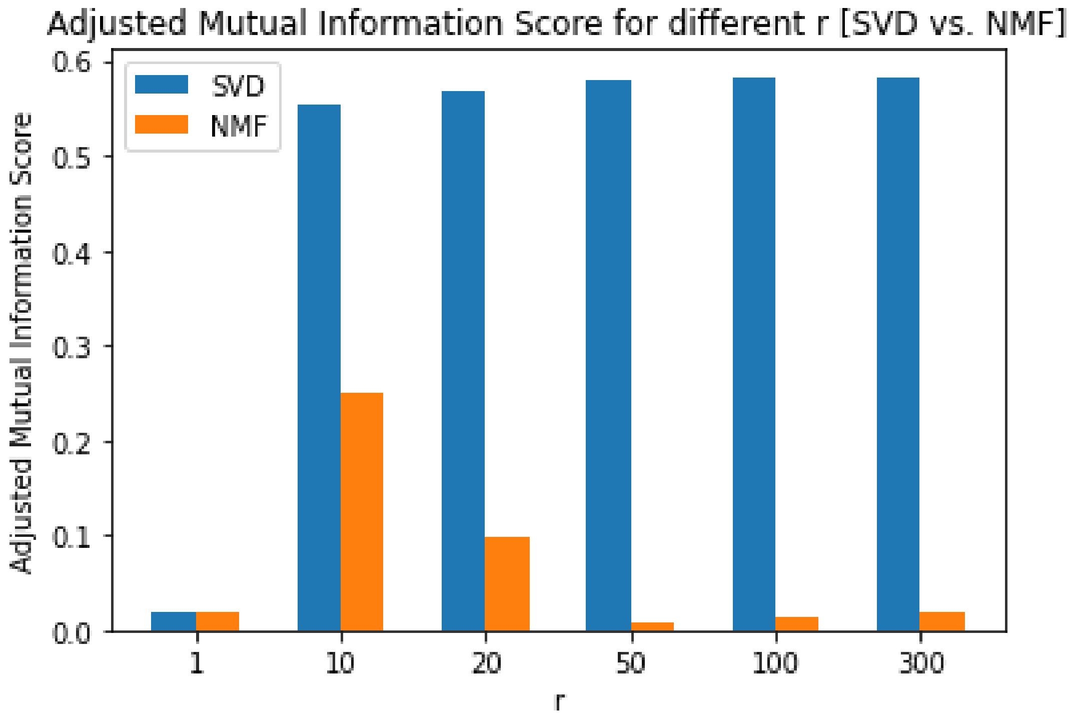
```
print_bar_scores_result(0.3, rs5, svd_comp, nmf_comp, 'Completeness',␣
↪'Completeness Score for different r [SVD vs. NMF]')
print_bar_scores_result(0.3, rs5, svd_vmes, nmf_vmes, 'V-measure', 'V-measure␣
↪Score for different r [SVD vs. NMF]')
print_bar_scores_result(0.3, rs5, svd_ranI, nmf_ranI, 'Adjusted Rand-Index',␣
↪'Adjusted Rand-Index for different r [SVD vs. NMF]')
print_bar_scores_result(0.3, rs5, svd_muts, nmf_muts, 'Adjusted Mutual␣
↪Information Score', 'Adjusted Mutual Information Score for different r [SVD␣
↪vs. NMF]')
```




Completeness Score for different r [SVD vs. NMF]
SVD
NMF
Completeness
0.0
0.1
0.2
0.3
0.4
0.5
0.6
1
10
20
50
100
300
r


V-measure Score for different r [SVD vs. NMF]
SVD
NMF
V-measure
0.0
0.1
0.2
0.3
0.4
0.5
0.6
1
10
20
50
100
300
r


Adjusted Rand-Index for different r [SVD vs. NMF]
Adjusted Rand-Index
r
SVD
NMF
0.0
0.1
0.2
0.3
0.4
0.5
0.6
1
10
20
50
100
300


Adjusted Mutual Information Score for different r [SVD vs. NMF]
Adjusted Mutual Information Score
r
SVD
NMF
0.0
0.1
0.2
0.3
0.4
0.5
0.6
1
10
20
50
100
300

Report a good choice of r for SVD and NMF respectively.

Ans:

From the graph above, we can find that a good choice of r for SVD could be $r = 50$, a good choice of r for NMF could be $r = 10$.

Because we can find that there are no significant increase for SVD after $r = 50$. And as r increases, we need to realize that the Euclidean distances in K-means we use will converge to a constant value, such that K-means doesn't perform well in high-dimensional prediction. Considering to this, choosing $r = 50$ is good enough. On the other hand, NMF performs obviously best when $r = 10$.

0.7 Question 6

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

Ans:

As r increases, the performance of the measures doesn't increase correspondingly. It's because that as r increases, it indicates that the target matrix behind it will become complicated and high-dimensional. In which restrained the performance of K-means as we mentioned before. The Euclidean distances in K-means will converge to a constant value between the sample points. The clustering method can't find centroids for different sample points so that its performance becomes poor.

On the other hand, we can find that SVD maintain the similar score performance even after r keeps increasing, however, NMF doesn't perform well after r is larger than 10. It might because of the fact that SVD is a much more deterministic method than NMF. As we discussed in Project 1, SVD is a more insightful method and is able to interpret high-dimensional data rather than NMF did. Since NMF only uses the positive entries in the reduced matrix and makes assumption about the missing values. SVD doesn't assume anything about the value. Therefore, it's less restricted and performs better in high-dimensional data.

0.8 Question 7

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

Ans:

No, even though we minus the outlier data ($r = 1$), the average performance of these measures is still worse than those computed in Question 3.

```
[ ]: print("Measure scores computed in Question 3:")
     print_5_measure_scores(y_data, km.labels_)

     print("\nAverage measure scores for SVD:")
     print("Homogeneity: %0.4f" % (sum(svd_homo) / len(svd_homo)))
     print("Completeness: %0.4f" % (sum(svd_comp) / len(svd_comp)))
     print("V-measure: %0.4f" % (sum(svd_vmes) / len(svd_vmes)))
     print("Adjusted Rand-Index: %0.4f" % (sum(svd_ranI) / len(svd_ranI)))
     print("Adjusted Mutual Information Score: %0.4f" % (sum(svd_muts) /␣
      ↪len(svd_muts)))
```

```
print("\nAverage measure scores for NMF:")
print("Homogeneity: %0.4f" % (sum(nmf_homo) / len(nmf_homo)))
print("Completeness: %0.4f" % (sum(nmf_comp) / len(nmf_comp)))
print("V-measure: %0.4f" % (sum(nmf_vmes) / len(nmf_vmes)))
print("Adjusted Rand-Index: %0.4f" % (sum(nmf_ranI) / len(nmf_ranI)))
print("Adjusted Mutual Information Score: %0.4f" % (sum(nmf_muts) /␣
 ↪len(nmf_muts)))

print("\nAverage measure scores for SVD (not including r=1):")
print("Homogeneity: %0.4f" % ( (sum(svd_homo) - svd_homo[0]) / (len(svd_homo) -␣
 ↪1)))
print("Completeness: %0.4f" % ( (sum(svd_comp) - svd_comp[0]) / (len(svd_comp)␣
 ↪- 1)))
print("V-measure: %0.4f" % ( (sum(svd_vmes) - svd_vmes[0]) / (len(svd_vmes) -␣
 ↪1)))
print("Adjusted Rand-Index: %0.4f" % ( (sum(svd_ranI) - svd_ranI[0]) /␣
 ↪(len(svd_ranI) - 1)))
print("Adjusted Mutual Information Score: %0.4f" % ( (sum(svd_muts) -␣
 ↪svd_ranI[0]) / (len(svd_muts) - 1)))
```

```
Measure scores computed in Question 3:
Homogeneity: 0.5999
Completeness: 0.6121
V-measure: 0.6059
Adjusted Rand-Index: 0.6659
Adjusted Mutual Information Score: 0.6059

Average measure scores for SVD:
Homogeneity: 0.4746
Completeness: 0.4875
V-measure: 0.4810
Adjusted Rand-Index: 0.5254
Adjusted Mutual Information Score: 0.4809

Average measure scores for NMF:
Homogeneity: 0.0528
Completeness: 0.1453
V-measure: 0.0680
Adjusted Rand-Index: 0.0307
Adjusted Mutual Information Score: 0.0679

Average measure scores for SVD (not including r=1):
Homogeneity: 0.5658
Completeness: 0.5811
V-measure: 0.5733
Adjusted Rand-Index: 0.6253
Adjusted Mutual Information Score: 0.5719
```

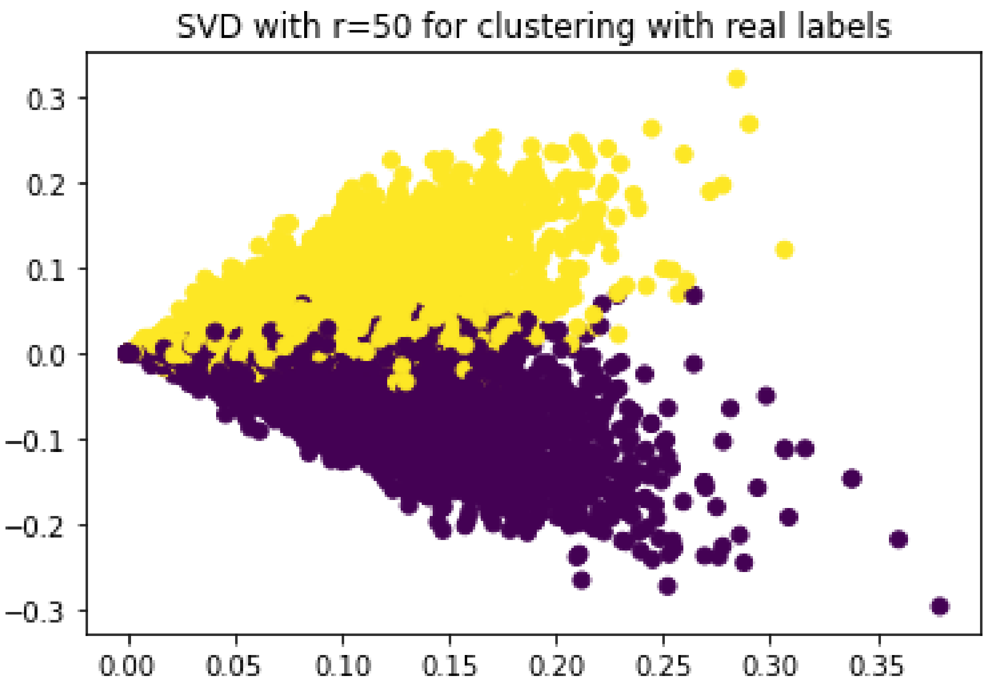
0.9 Question 8

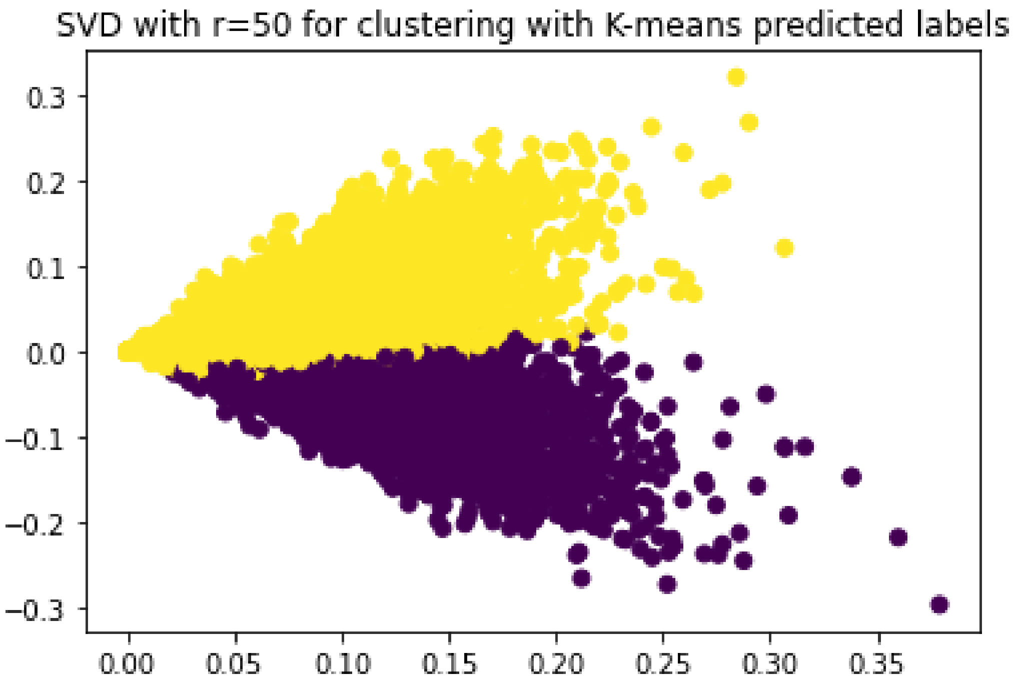
Visualize the clustering results for:

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

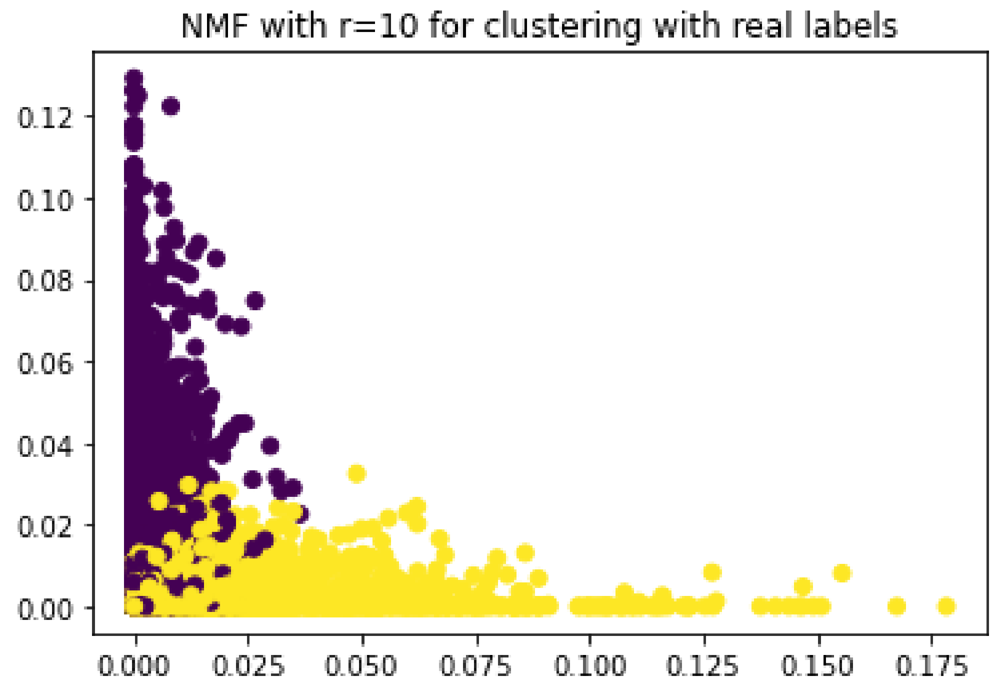
```
[ ]: # best parameter of SVD and NMF
best_r_SVD = 50
best_r_NMF = 10
kmm = KMeans(n_clusters=2, init='k-means++', max_iter=2000, n_init=50,␣
 ↪random_state=0)
# best SVD
best_svd = TruncatedSVD(n_components=best_r_SVD, random_state=0)
best_X_svd = best_svd.fit_transform(X_tfidf)
y_pred_best_svd = kmm.fit_predict(best_X_svd)
# best NMF
best_nmf = NMF(n_components=best_r_NMF, init='random', random_state=0)
best_X_nmf = best_nmf.fit_transform(X_tfidf)
y_pred_best_nmf = kmm.fit_predict(best_X_nmf)
```

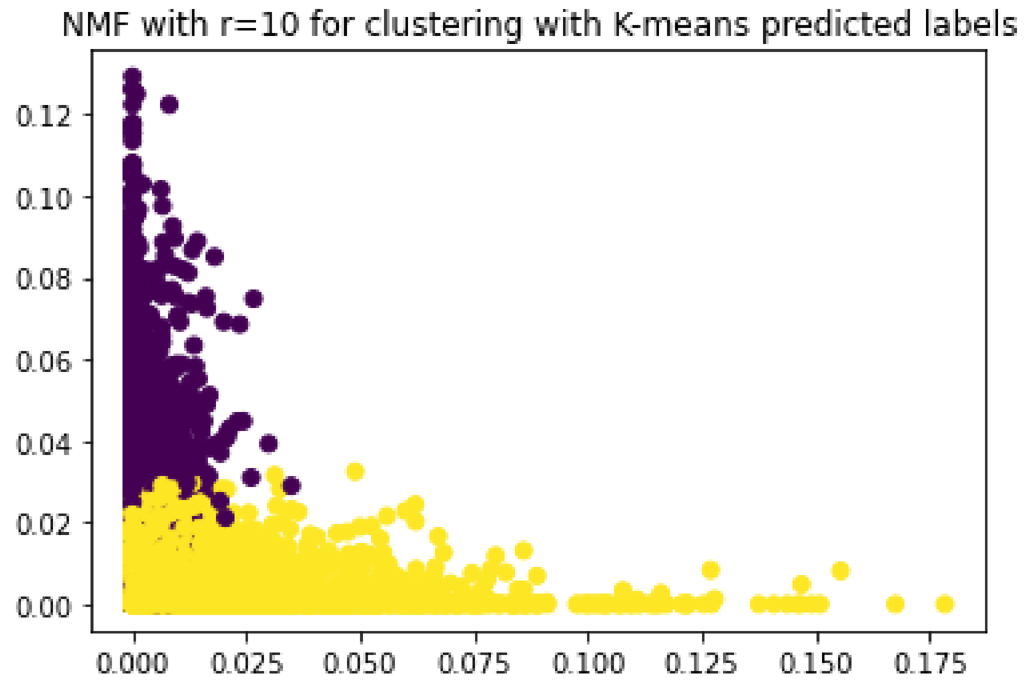
```
[ ]: plt.scatter(best_X_svd[:,0], best_X_svd[:,1], c=y_data)
plt.title("SVD with r=50 for clustering with real labels")
plt.show()
plt.scatter(best_X_svd[:,0], best_X_svd[:,1], c=y_pred_best_svd)
plt.title("SVD with r=50 for clustering with K-means predicted labels")
plt.show()
```





[ ]:
```
plt.scatter(best_X_nmf[:,0], best_X_nmf[:,1], c=y_data)
plt.title("NMF with r=10 for clustering with real labels")
plt.show()
plt.scatter(best_X_nmf[:,0], best_X_nmf[:,1], c=y_pred_best_nmf)
plt.title("NMF with r=10 for clustering with K-means predicted labels")
plt.show()
```





0.10 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?

Ans:

1. From the visualization, we can see that there are no spherical distributions on the plots for both SVD or NMF. there are many overlapping points among the two clusters which makes the boundary of two clusters hard to define.
2. The data points in the SVD are distributed more ideally than NMF did. Its shape also looks more like a sphere comparing to NMF one. On the other hand, we can analyze the performances from their homogeneity scores. SVD gets a higher score than NMF did but it's still not great enough.
3. In conclusion, the distribution of the data is not ideal for K-Means clustering. K-Means clustering assumes spherical shapes of clusters but we can not find a similar shape no matter in SVD or NMF here.

0.11 Question 10

Load documents with the same configuration as in Question 1, but for ALL 20 categories.

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.

```
[ ]: dataset_all = fetch_20newsgroups(subset = 'all', shuffle = True, random_state =␣
      ↪0,remove=('headers','footers'))

     vec = CountVectorizer(stop_words='english', min_df=3)
     tfidf = TfidfTransformer()

     X_a_vec = vec.fit_transform(dataset_all.data)
     X_a_tfidf = tfidf.fit_transform(X_a_vec)
     print("TFIDX dataset shape:", X_a_tfidf.shape)
```

```
TFIDX dataset shape: (18846, 45365)
```

```
[ ]: y_a_data = dataset_all.target

     kma = KMeans(n_clusters=20, init='k-means++', max_iter=2000, n_init=50,␣
      ↪random_state=0)
     # kma.fit(X_a_tfidf)
```

Construct the TF-IDF matrix, reduce its dimensionality using BOTH NMF and SVD (specify settings you choose and why).

Ans:

We choose $r = 20$ for SVD, $r = 10$ for NMF respectively. we will explain the reason in the following cells.

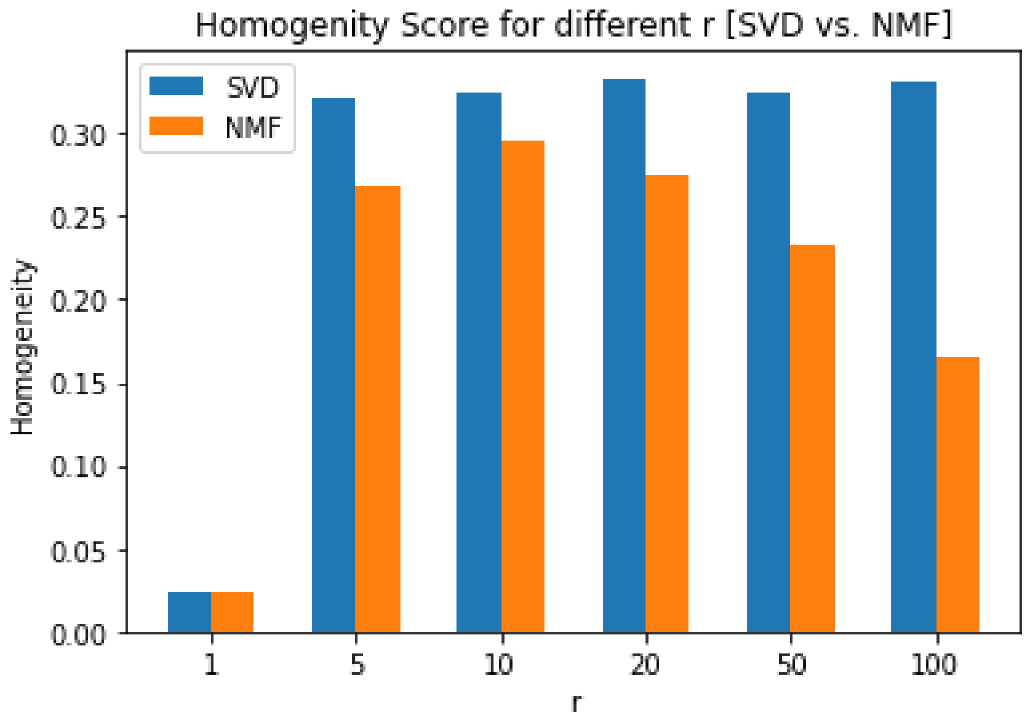
```
[ ]: rs10 = [1, 5, 10, 20, 50, 100]
     # svd
     a_svd_homo = []
     a_svd_comp = []
     a_svd_vmes = []
     a_svd_ranI = []
     a_svd_muts = []
     # nmf
     a_nmf_homo = []
     a_nmf_comp = []
     a_nmf_vmes = []
     a_nmf_ranI = []
     a_nmf_muts = []
     for i in range(len(rs10)):
       print("Try r =", rs10[i], "...")
       # SVD
       svd_tmp = TruncatedSVD(n_components=rs10[i], random_state=0)
       X_a_svd = svd_tmp.fit_transform(X_a_tfidf)
       kma_svd = kma.fit(X_a_svd)
       a_svd_homo.append(homogeneity_score(y_a_data, kma_svd.labels_))
       a_svd_comp.append(completeness_score(y_a_data, kma_svd.labels_))
       a_svd_vmes.append(v_measure_score(y_a_data, kma_svd.labels_))
       a_svd_ranI.append(adjusted_rand_score(y_a_data, kma_svd.labels_))
       a_svd_muts.append(adjusted_mutual_info_score(y_a_data, kma_svd.labels_))
       # NMF
       nmf_tmp = NMF(n_components=rs10[i], init='random', random_state=0)
       X_a_nmf = nmf_tmp.fit_transform(X_a_tfidf)
       kma_nmf = kma.fit(X_a_nmf)
       a_nmf_homo.append(homogeneity_score(y_a_data, kma_nmf.labels_))
       a_nmf_comp.append(completeness_score(y_a_data, kma_nmf.labels_))
       a_nmf_vmes.append(v_measure_score(y_a_data, kma_nmf.labels_))
       a_nmf_ranI.append(adjusted_rand_score(y_a_data, kma_nmf.labels_))
       a_nmf_muts.append(adjusted_mutual_info_score(y_a_data, kma_nmf.labels_))
     print('Done')
```

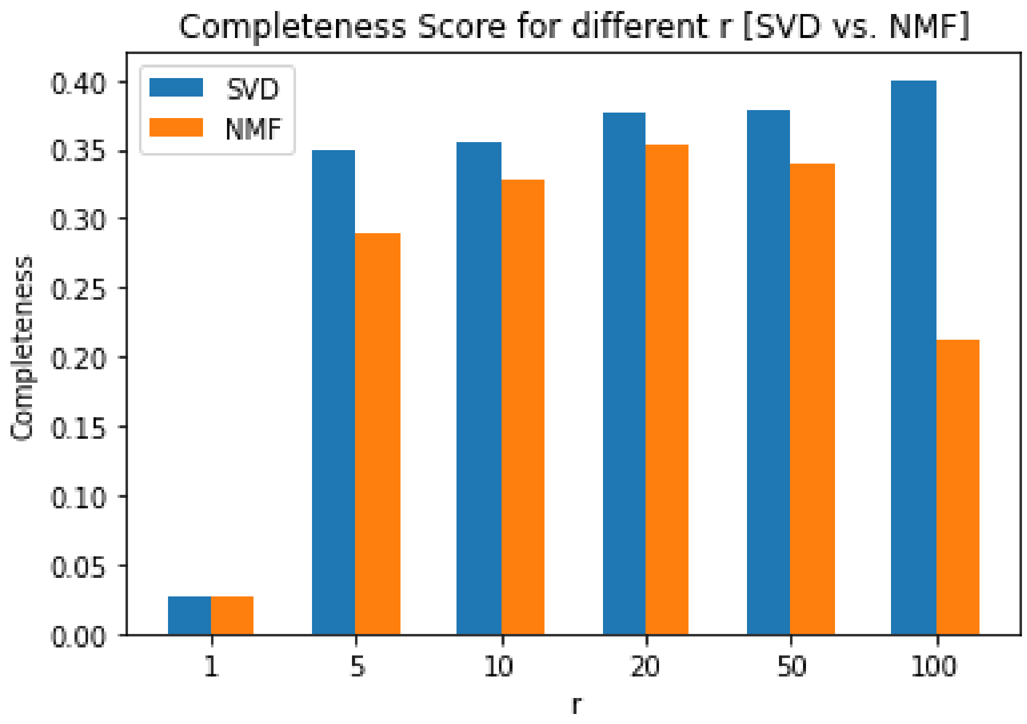
```
Try r = 1 …
Try r = 5 …
Try r = 10 …
```

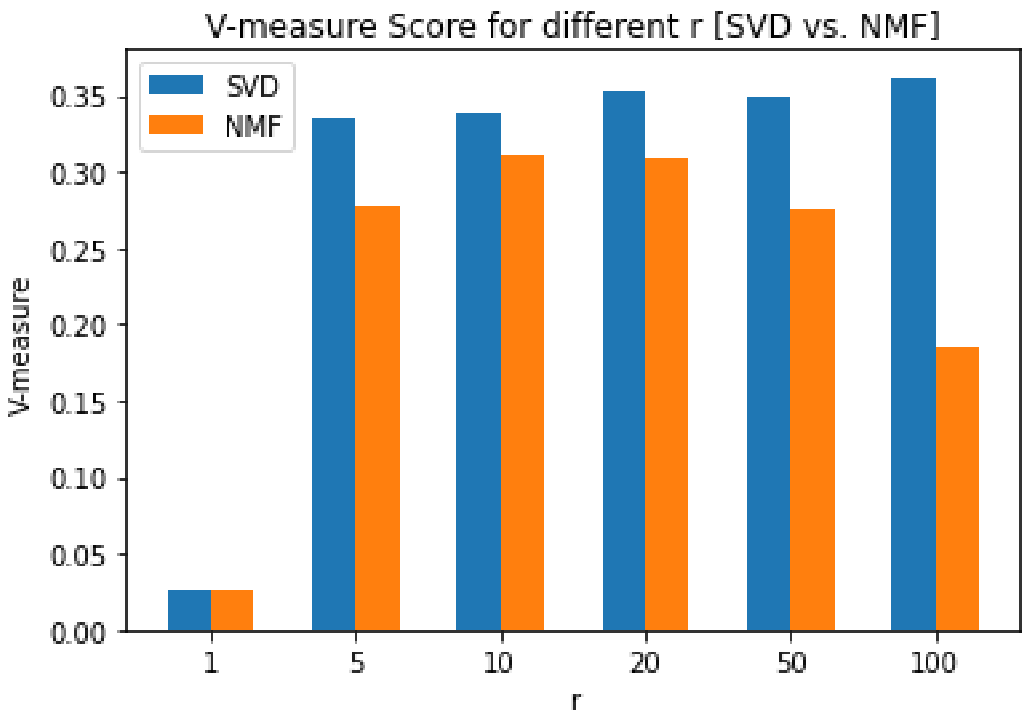
```
/usr/local/lib/python3.8/dist-packages/sklearn/decomposition/_nmf.py:1637:
ConvergenceWarning: Maximum number of iterations 200 reached. Increase it to
improve convergence.
  warnings.warn(
```

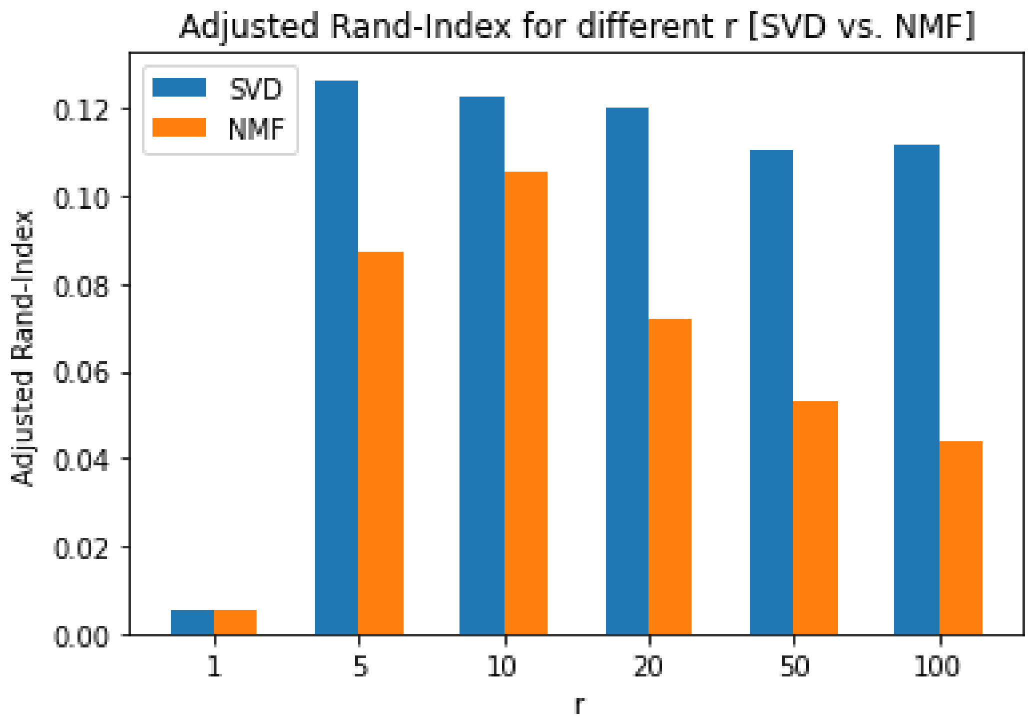
```
Try r = 20 …
Try r = 50 …
Try r = 100 …
Done
```

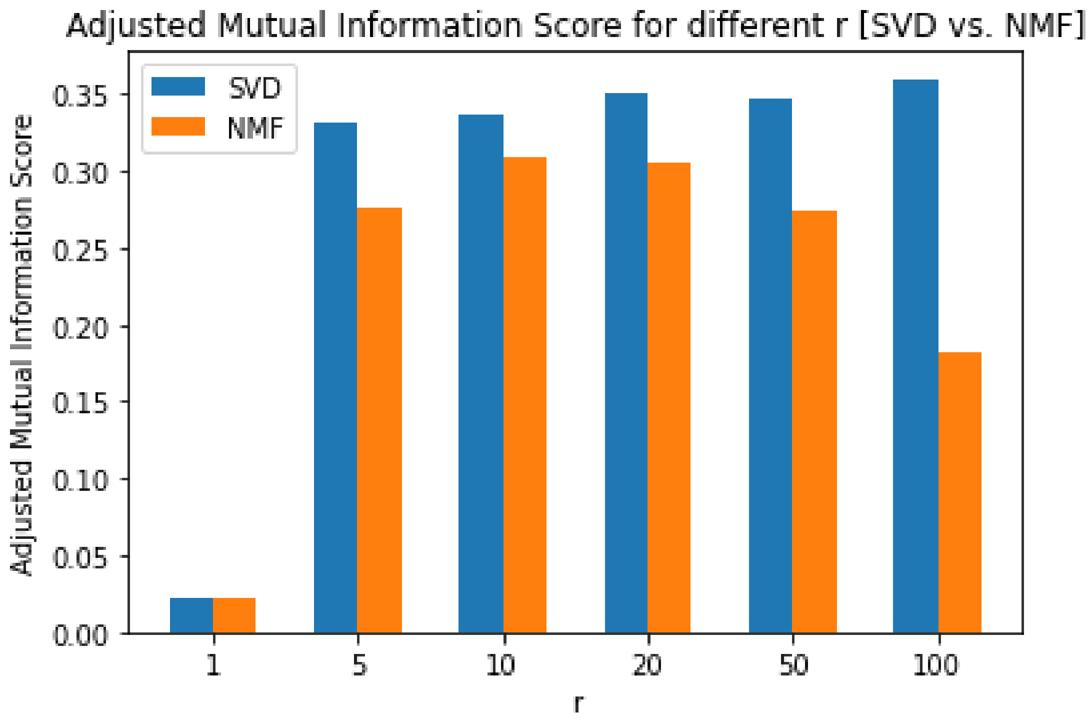
```
[ ]: print_bar_scores_result(0.3, rs10, a_svd_homo, a_nmf_homo, 'Homogeneity',␣
      ↪'Homogenity Score for different r [SVD vs. NMF]')
     print_bar_scores_result(0.3, rs10, a_svd_comp, a_nmf_comp, 'Completeness',␣
      ↪'Completeness Score for different r [SVD vs. NMF]')
     print_bar_scores_result(0.3, rs10, a_svd_vmes, a_nmf_vmes, 'V-measure',␣
      ↪'V-measure Score for different r [SVD vs. NMF]')
     print_bar_scores_result(0.3, rs10, a_svd_ranI, a_nmf_ranI, 'Adjusted␣
      ↪Rand-Index', 'Adjusted Rand-Index for different r [SVD vs. NMF]')
     print_bar_scores_result(0.3, rs10, a_svd_muts, a_nmf_muts, 'Adjusted Mutual␣
      ↪Information Score', 'Adjusted Mutual Information Score for different r [SVD␣
      ↪vs. NMF]')
```




Completeness Score for different r [SVD vs. NMF]
SVD
NMF
Completeness
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
1
5
10
20
50
100
r


V-measure Score for different r [SVD vs. NMF]
SVD
NMF
V-measure
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
1
5
10
20
50
100
r


Adjusted Rand-Index for different r [SVD vs. NMF]
SVD
NMF
Adjusted Rand-Index
0.12
0.10
0.08
0.06
0.04
0.02
0.00
1
5
10
20
50
100
r


Adjusted Mutual Information Score for different r [SVD vs. NMF]
SVD
NMF
Adjusted Mutual Information Score
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
1
5
10
20
50
100
r

```
[ ]: avg_svd_homo = sum(a_svd_homo) / len(a_svd_homo)
avg_svd_comp = sum(a_svd_comp) / len(a_svd_comp)
avg_svd_vmes = sum(a_svd_vmes) / len(a_svd_vmes)
avg_svd_ranI = sum(a_svd_ranI) / len(a_svd_ranI)
avg_svd_muts = sum(a_svd_muts) / len(a_svd_muts)

avg_nmf_homo = sum(a_nmf_homo) / len(a_nmf_homo)
avg_nmf_comp = sum(a_nmf_comp) / len(a_nmf_comp)
avg_nmf_vmes = sum(a_nmf_vmes) / len(a_nmf_vmes)
avg_nmf_ranI = sum(a_nmf_ranI) / len(a_nmf_ranI)
avg_nmf_muts = sum(a_nmf_muts) / len(a_nmf_muts)

avg_svd = []
avg_nmf = []
for i in range(len(rs10)):
  tmp_svd = 0
  tmp_svd += a_svd_homo[i] / avg_svd_homo
  tmp_svd += a_svd_comp[i] / avg_svd_comp
  tmp_svd += a_svd_vmes[i] / avg_svd_vmes
  tmp_svd += a_svd_ranI[i] / avg_svd_ranI
  tmp_svd += a_svd_muts[i] / avg_svd_muts
  avg_svd.append(tmp_svd / 5)
  tmp_nmf = 0
  tmp_nmf += a_nmf_homo[i] / avg_nmf_homo
  tmp_nmf += a_nmf_comp[i] / avg_nmf_comp
  tmp_nmf += a_nmf_vmes[i] / avg_nmf_vmes
  tmp_nmf += a_nmf_ranI[i] / avg_nmf_ranI
  tmp_nmf += a_nmf_muts[i] / avg_nmf_muts
  avg_nmf.append(tmp_nmf / 5)
for i in range(len(avg_svd)):
  print("Average normalized scores for SVD when r =", rs10[i], ":", avg_svd[i])
print("")
for i in range(len(avg_nmf)):
  print("Average normalized scores for NMF when r =", rs10[i], ":", avg_nmf[i])

sorted_avg_svd = sorted(avg_svd, reverse=True)
sorted_avg_nmf = sorted(avg_nmf, reverse=True)

top3_svd = [avg_svd.index(v) for v in sorted_avg_svd[:3]]
top3_nmf = [avg_nmf.index(v) for v in sorted_avg_nmf[:3]]

print("\nTop 3 best values of r for SVD:")
for i in top3_svd:
  print("r =", rs10[i], ":", avg_svd[i])

print("\nTop 3 best values of r for NMF:")
for i in top3_nmf:
```

```
print("r =", rs10[i], ":", avg_nmf[i])
```

```
Average normalized scores for SVD when r = 1 : 0.07766046505240246
Average normalized scores for SVD when r = 5 : 1.164613083355113
Average normalized scores for SVD when r = 10 : 1.169345407002839
Average normalized scores for SVD when r = 20 : 1.2027367831942999
Average normalized scores for SVD when r = 50 : 1.174476757975515
Average normalized scores for SVD when r = 100 : 1.2111675034198306

Average normalized scores for NMF when r = 1 : 0.10256664314454249
Average normalized scores for NMF when r = 5 : 1.2478007770753372
Average normalized scores for NMF when r = 10 : 1.4200459030547532
Average normalized scores for NMF when r = 20 : 1.3052363009210974
Average normalized scores for NMF when r = 50 : 1.1389088770343618
Average normalized scores for NMF when r = 100 : 0.7854414987699082

Top 3 best values of r for SVD:
r = 100 : 1.2111675034198306
r = 20 : 1.2027367831942999
r = 50 : 1.174476757975515

Top 3 best values of r for NMF:
r = 10 : 1.4200459030547532
r = 20 : 1.3052363009210974
r = 5 : 1.2478007770753372
```

We computed the 5 measure scores, normalized it to sum up and get average. As you can see, we obtained the top 3 best values of r for SVD and NMF respectively.

Based on above result, we choose $r = 20$ for SVD, $r = 10$ for NMF as setting.

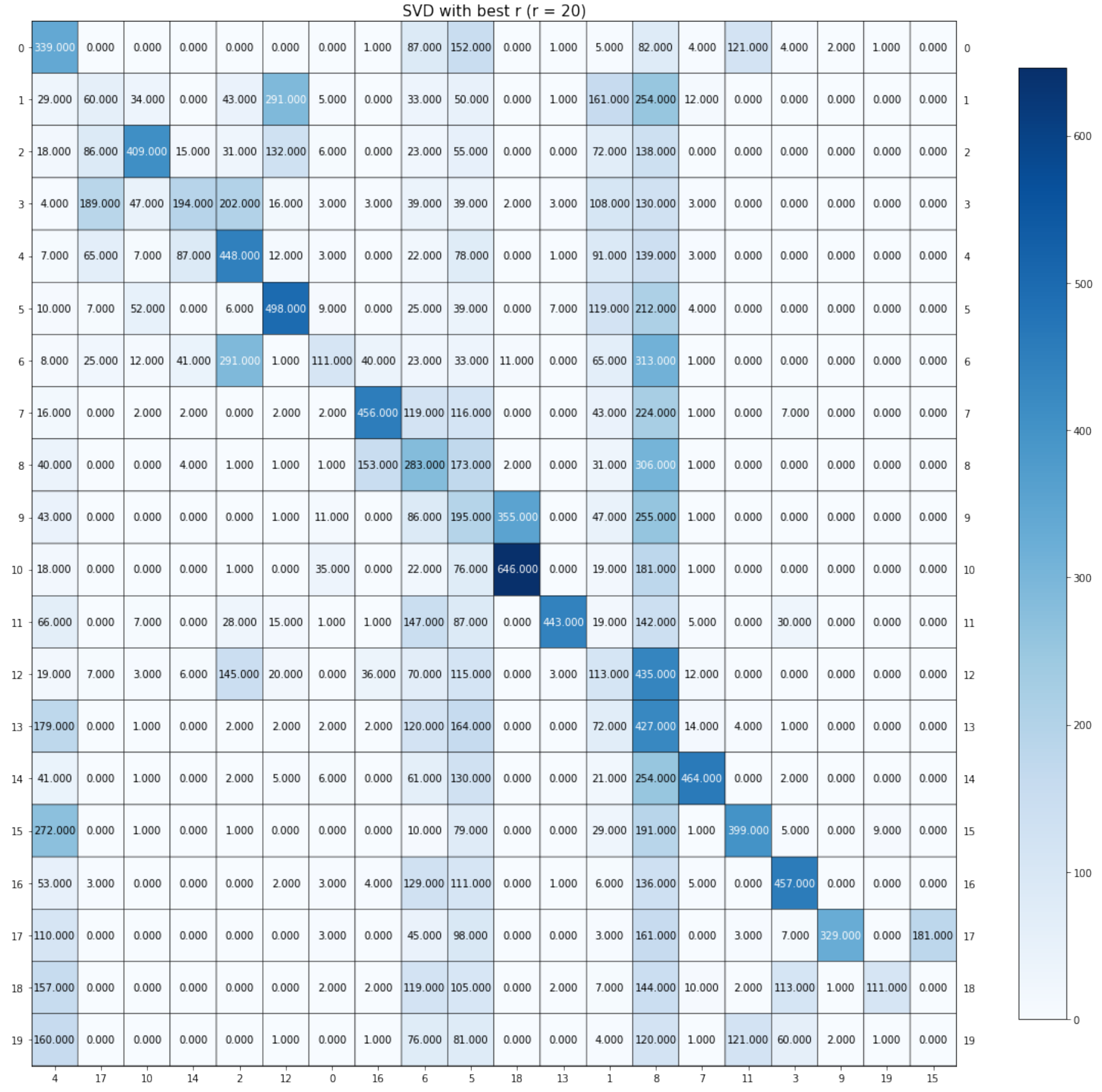
As r increases to 100 for SVD, there is no corresponding significant increasing in the normalized scores for it. Therefore, choosing $r = 20$ is good enough.

Choosing $r = 10$ for NMF is easy here, since it's lower than 20 but get a higher normalized average score.

Visualize the contingency matrix and report the five clustering metrics (DO BOTH NMF AND SVD).

```
[ ]: best_a_svd = 20
     # SVD
     best_svd = TruncatedSVD(n_components=best_a_svd, random_state=0)
     best_X_a_svd = best_svd.fit_transform(X_a_tfidf)
     best_kma_svd = kma.fit(best_X_a_svd)
     cm = confusion_matrix(dataset_all.target, kma.labels_)
     rows, cols = linear_sum_assignment(cm, maximize=True)
     plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
      ↪title='SVD with best r (r = 20)', size=(15,15))
```

SVD with best r (r = 20)

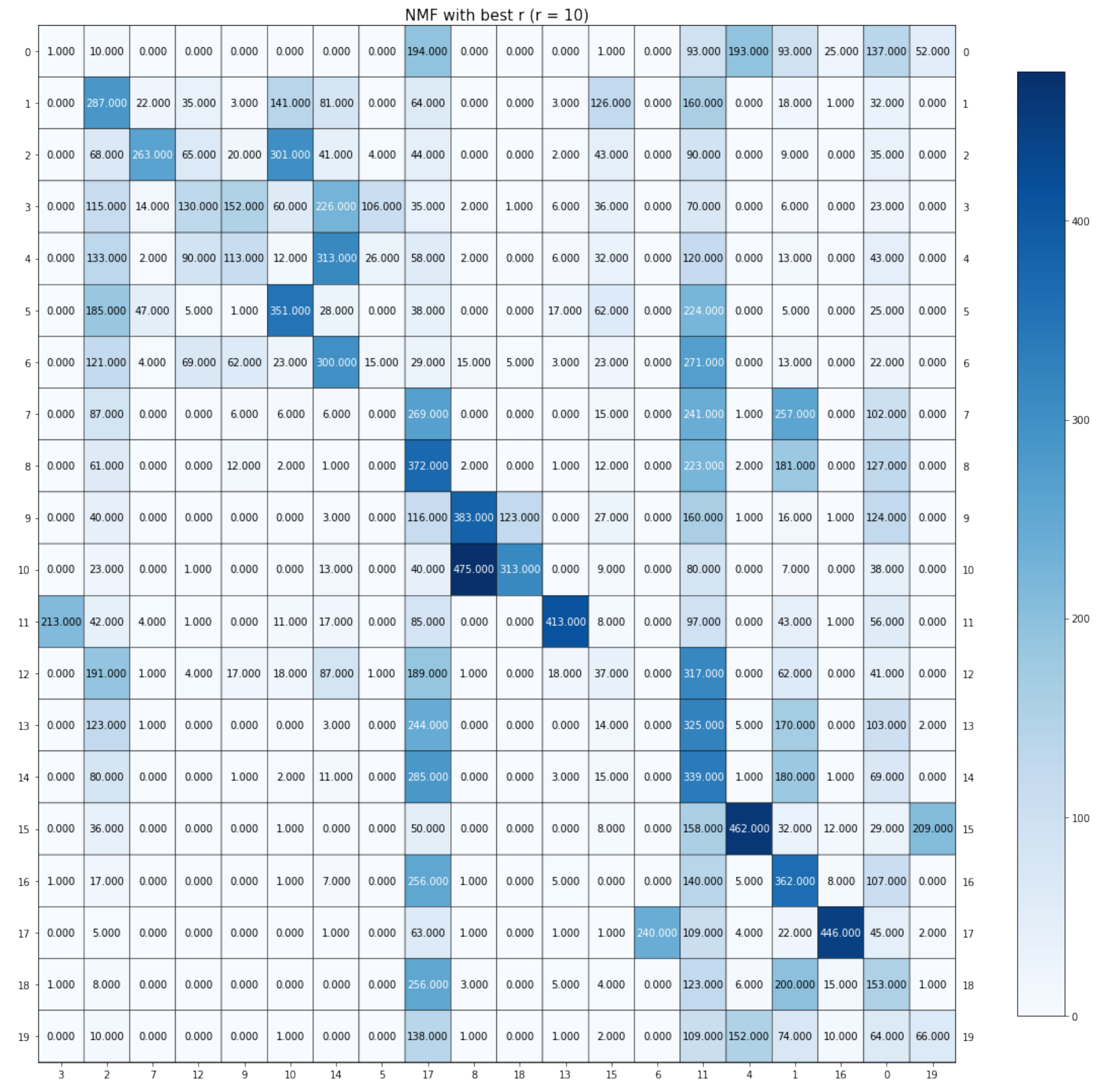


	4	17	10	14	2	12	0	16	6	5	18	13	1	8	7	11	3	9	19	15
0	339.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	87.000	152.000	0.000	1.000	5.000	82.000	4.000	121.000	4.000	2.000	1.000	0.000
1	29.000	60.000	34.000	0.000	43.000	291.000	5.000	0.000	33.000	50.000	0.000	1.000	161.000	254.000	12.000	0.000	0.000	0.000	0.000	0.000
2	18.000	86.000	409.000	15.000	31.000	132.000	6.000	0.000	23.000	55.000	0.000	0.000	72.000	138.000	0.000	0.000	0.000	0.000	0.000	0.000
3	4.000	189.000	47.000	194.000	202.000	16.000	3.000	3.000	39.000	39.000	2.000	3.000	108.000	130.000	3.000	0.000	0.000	0.000	0.000	0.000
4	7.000	65.000	7.000	87.000	448.000	12.000	3.000	0.000	22.000	78.000	0.000	1.000	91.000	139.000	3.000	0.000	0.000	0.000	0.000	0.000
5	10.000	7.000	52.000	0.000	6.000	498.000	9.000	0.000	25.000	39.000	0.000	7.000	119.000	212.000	4.000	0.000	0.000	0.000	0.000	0.000
6	8.000	25.000	12.000	41.000	291.000	1.000	111.000	40.000	23.000	33.000	11.000	0.000	65.000	313.000	1.000	0.000	0.000	0.000	0.000	0.000
7	16.000	0.000	2.000	2.000	0.000	2.000	2.000	456.000	119.000	116.000	0.000	0.000	43.000	224.000	1.000	0.000	7.000	0.000	0.000	0.000
8	40.000	0.000	0.000	4.000	1.000	1.000	1.000	153.000	283.000	173.000	2.000	0.000	31.000	306.000	1.000	0.000	0.000	0.000	0.000	0.000
9	43.000	0.000	0.000	0.000	0.000	1.000	11.000	0.000	86.000	195.000	355.000	0.000	47.000	255.000	1.000	0.000	0.000	0.000	0.000	0.000
10	18.000	0.000	0.000	0.000	1.000	0.000	35.000	0.000	22.000	76.000	646.000	0.000	19.000	181.000	1.000	0.000	0.000	0.000	0.000	0.000
11	66.000	0.000	7.000	0.000	28.000	15.000	1.000	1.000	147.000	87.000	0.000	443.000	19.000	142.000	5.000	0.000	30.000	0.000	0.000	0.000
12	19.000	7.000	3.000	6.000	145.000	20.000	0.000	36.000	70.000	115.000	0.000	3.000	113.000	435.000	12.000	0.000	0.000	0.000	0.000	0.000
13	179.000	0.000	1.000	0.000	2.000	2.000	2.000	2.000	120.000	164.000	0.000	0.000	72.000	427.000	14.000	4.000	1.000	0.000	0.000	0.000
14	41.000	0.000	1.000	0.000	2.000	5.000	6.000	0.000	61.000	130.000	0.000	0.000	21.000	254.000	464.000	0.000	2.000	0.000	0.000	0.000
15	272.000	0.000	1.000	0.000	1.000	0.000	0.000	0.000	10.000	79.000	0.000	0.000	29.000	191.000	1.000	399.000	5.000	0.000	9.000	0.000
16	53.000	3.000	0.000	0.000	0.000	2.000	3.000	4.000	129.000	111.000	0.000	1.000	6.000	136.000	5.000	0.000	457.000	0.000	0.000	0.000
17	110.000	0.000	0.000	0.000	0.000	0.000	3.000	0.000	45.000	98.000	0.000	0.000	3.000	161.000	0.000	3.000	7.000	329.000	0.000	181.000
18	157.000	0.000	0.000	0.000	0.000	0.000	2.000	2.000	119.000	105.000	0.000	2.000	7.000	144.000	10.000	2.000	113.000	1.000	111.000	0.000
19	160.000	0.000	0.000	0.000	0.000	1.000	0.000	1.000	76.000	81.000	0.000	0.000	4.000	120.000	1.000	121.000	60.000	2.000	1.000	0.000

```
[ ]: best_a_nmf = 10
     # NMF
     best_nmf = NMF(n_components=best_a_nmf, random_state=0)
     best_X_a_nmf = best_nmf.fit_transform(X_a_tfidf)
     best_kma_nmf = kma.fit(best_X_a_nmf)
     cm = confusion_matrix(dataset_all.target, kma.labels_)
     rows, cols = linear_sum_assignment(cm, maximize=True)
     plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
      ↪title='NMF with best r (r = 10)', size=(15,15))
```

```
/usr/local/lib/python3.8/dist-packages/sklearn/decomposition/_nmf.py:289:
FutureWarning: The 'init' value, when 'init=None' and n_components is less than
n_samples and n_features, will be changed from 'nndsvd' to 'nndsvda' in 1.1
```

```
(renaming of 0.26).
  warnings.warn(
```



	3	2	7	12	9	10	14	5	17	8	18	13	15	6	11	4	1	16	0	19
0	1.000	10.000	0.000	0.000	0.000	0.000	0.000	0.000	194.000	0.000	0.000	0.000	1.000	0.000	93.000	193.000	93.000	25.000	137.000	52.000
1	0.000	287.000	22.000	35.000	3.000	141.000	81.000	0.000	64.000	0.000	0.000	3.000	126.000	0.000	160.000	0.000	18.000	1.000	32.000	0.000
2	0.000	68.000	263.000	65.000	20.000	301.000	41.000	4.000	44.000	0.000	0.000	2.000	43.000	0.000	90.000	0.000	9.000	0.000	35.000	0.000
3	0.000	115.000	14.000	130.000	152.000	60.000	226.000	106.000	35.000	2.000	1.000	6.000	36.000	0.000	70.000	0.000	6.000	0.000	23.000	0.000
4	0.000	133.000	2.000	90.000	113.000	12.000	313.000	26.000	58.000	2.000	0.000	6.000	32.000	0.000	120.000	0.000	13.000	0.000	43.000	0.000
5	0.000	185.000	47.000	5.000	1.000	351.000	28.000	0.000	38.000	0.000	0.000	17.000	62.000	0.000	224.000	0.000	5.000	0.000	25.000	0.000
6	0.000	121.000	4.000	69.000	62.000	23.000	300.000	15.000	29.000	15.000	5.000	3.000	23.000	0.000	271.000	0.000	13.000	0.000	22.000	0.000
7	0.000	87.000	0.000	0.000	6.000	6.000	6.000	0.000	269.000	0.000	0.000	0.000	15.000	0.000	241.000	1.000	257.000	0.000	102.000	0.000
8	0.000	61.000	0.000	0.000	12.000	2.000	1.000	0.000	372.000	2.000	0.000	1.000	12.000	0.000	223.000	2.000	181.000	0.000	127.000	0.000
9	0.000	40.000	0.000	0.000	0.000	0.000	3.000	0.000	116.000	383.000	123.000	0.000	27.000	0.000	160.000	1.000	16.000	1.000	124.000	0.000
10	0.000	23.000	0.000	1.000	0.000	0.000	13.000	0.000	40.000	475.000	313.000	0.000	9.000	0.000	80.000	0.000	7.000	0.000	38.000	0.000
11	213.000	42.000	4.000	1.000	0.000	11.000	17.000	0.000	85.000	0.000	0.000	413.000	8.000	0.000	97.000	0.000	43.000	1.000	56.000	0.000
12	0.000	191.000	1.000	4.000	17.000	18.000	87.000	1.000	189.000	1.000	0.000	18.000	37.000	0.000	317.000	0.000	62.000	0.000	41.000	0.000
13	0.000	123.000	1.000	0.000	0.000	0.000	3.000	0.000	244.000	0.000	0.000	0.000	14.000	0.000	325.000	5.000	170.000	0.000	103.000	2.000
14	0.000	80.000	0.000	0.000	1.000	2.000	11.000	0.000	285.000	0.000	0.000	3.000	15.000	0.000	339.000	1.000	180.000	1.000	69.000	0.000
15	0.000	36.000	0.000	0.000	0.000	1.000	0.000	0.000	50.000	0.000	0.000	0.000	8.000	0.000	158.000	462.000	32.000	12.000	29.000	209.000
16	1.000	17.000	0.000	0.000	0.000	1.000	7.000	0.000	256.000	1.000	0.000	5.000	0.000	0.000	140.000	5.000	362.000	8.000	107.000	0.000
17	0.000	5.000	0.000	0.000	0.000	0.000	1.000	0.000	63.000	1.000	0.000	1.000	1.000	240.000	109.000	4.000	22.000	446.000	45.000	2.000
18	1.000	8.000	0.000	0.000	0.000	0.000	0.000	0.000	256.000	3.000	0.000	5.000	4.000	0.000	123.000	6.000	200.000	15.000	153.000	1.000
19	0.000	10.000	0.000	0.000	0.000	1.000	0.000	0.000	138.000	1.000	0.000	1.000	2.000	0.000	109.000	152.000	74.000	10.000	64.000	66.000

0.12 Question 11

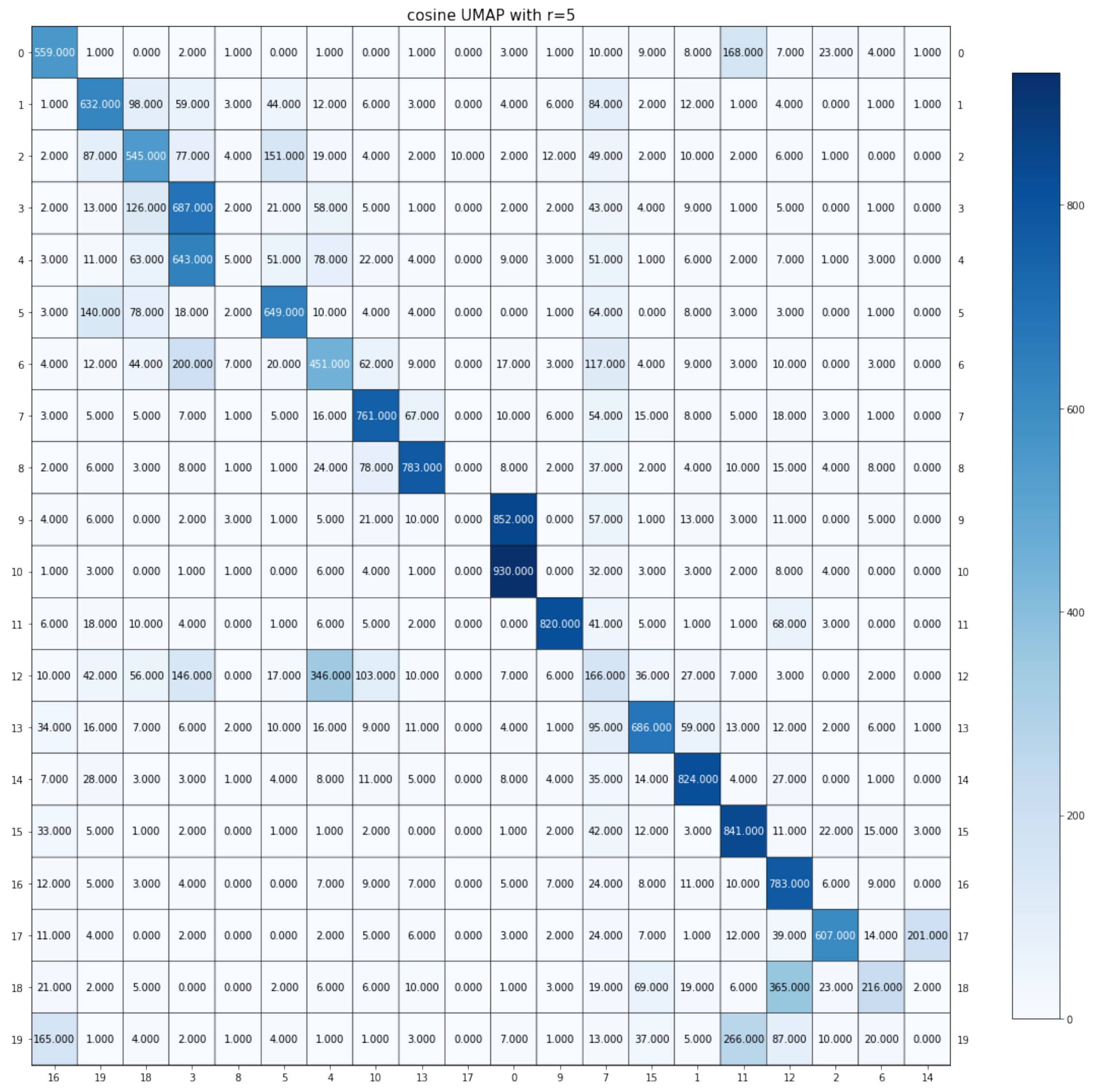
Reduce the dimension of your dataset with UMAP. Consider the following settings: n components = [5, 20, 200], metric = ”cosine” vs. ”euclidean”. If ”cosine” metric fails, please look at the FAQ at the end of this spec.

Report the permuted contingency matrix and the five clustering evaluation metrics for the different combinations (6 combinations).

```
[ ]: rs11 = [5, 20, 200]
     # metrics = ['cosine', 'euclidean']
     # UMAP cosine
     umap_cos_homo = []
     umap_cos_comp = []
     umap_cos_vmes = []
     umap_cos_ranI = []
     umap_cos_muts = []
     # UMAP euclidean
     umap_euc_homo = []
     umap_euc_comp = []
     umap_euc_vmes = []
     umap_euc_ranI = []
     umap_euc_muts = []

     kma = KMeans(n_clusters=20, init='k-means++', max_iter=2000, n_init=50,␣
      ↪random_state=0)

     for i in range(len(rs11)):
       print("\nTry r =", rs11[i], "...")
       # cosine
       print("Cosine UMAP when r =", rs11[i], ":")
       tmp_umap_cos = umap.UMAP(n_components=rs11[i], metric='cosine')
       X_umap_cos = tmp_umap_cos.fit_transform(X_a_tfidf)
       kma_cos = kma.fit(X_umap_cos)
       umap_cos_homo.append(homogeneity_score(y_a_data, kma_cos.labels_))
       umap_cos_comp.append(completeness_score(y_a_data, kma_cos.labels_))
       umap_cos_vmes.append(v_measure_score(y_a_data, kma_cos.labels_))
       umap_cos_ranI.append(adjusted_rand_score(y_a_data, kma_cos.labels_))
       umap_cos_muts.append(adjusted_mutual_info_score(y_a_data, kma_cos.labels_))
       # print permuted contingency matrix
       cm = confusion_matrix(y_a_data, kma.labels_)
       rows, cols = linear_sum_assignment(cm, maximize=True)
       plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
      ↪title='cosine UMAP with r=' + str(rs11[i]), size=(15,15))
       # print the five clustering evaluation metrics
       print_5_measure_scores(y_a_data, kma_cos.labels_)
       # euclidean
       print("\nEuclidean UMAP when r =", rs11[i], ":")
       tmp_umap_euc = umap.UMAP(n_components=rs11[i], metric='euclidean')
       X_umap_euc = tmp_umap_cos.fit_transform(X_a_tfidf)
       kma_euc = kma.fit(X_umap_euc)
       umap_euc_homo.append(homogeneity_score(y_a_data, kma_euc.labels_))
       umap_euc_comp.append(completeness_score(y_a_data, kma_euc.labels_))
       umap_euc_vmes.append(v_measure_score(y_a_data, kma_euc.labels_))
       umap_euc_ranI.append(adjusted_rand_score(y_a_data, kma_euc.labels_))
       umap_euc_muts.append(adjusted_mutual_info_score(y_a_data, kma_euc.labels_))
       # print permuted contingency matrix
       cm = confusion_matrix(y_a_data, kma.labels_)
```

```
    rows, cols = linear_sum_assignment(cm, maximize=True)
    plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
 ↪title='euclidean UMAP with r=' + str(rs11[i]), size=(15,15))
    # print the five clustering evaluation metrics
    print_5_measure_scores(y_a_data, kma_euc.labels_)
print('Done')
```

```
Try r = 5 …
Cosine UMAP when r = 5 :
```

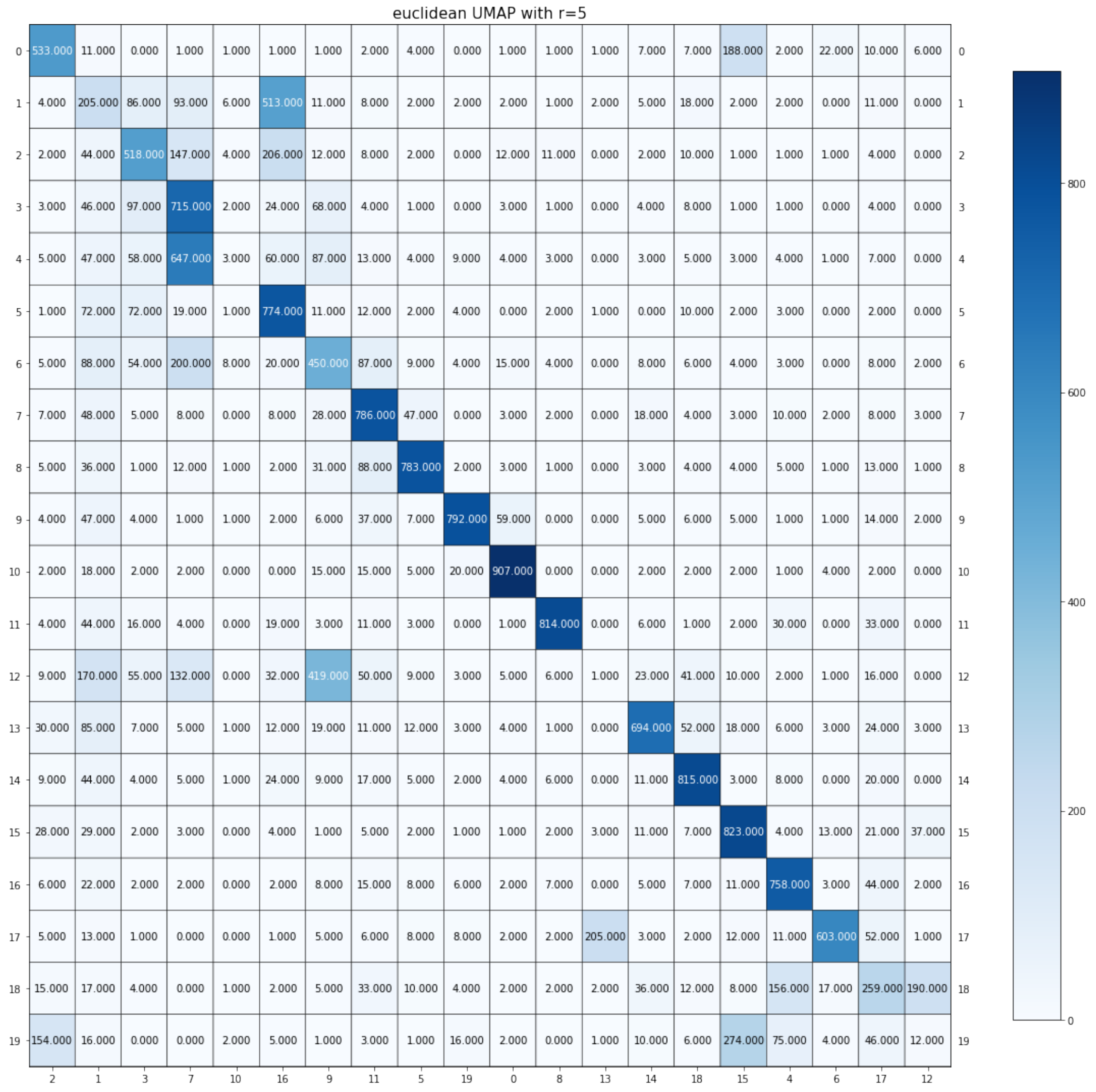


	16	19	18	3	8	5	4	10	13	17	0	9	7	15	1	11	12	2	6	14
0	559.000	1.000	0.000	2.000	1.000	0.000	1.000	0.000	1.000	0.000	3.000	1.000	10.000	9.000	8.000	168.000	7.000	23.000	4.000	1.000
1	1.000	632.000	98.000	59.000	3.000	44.000	12.000	6.000	3.000	0.000	4.000	6.000	84.000	2.000	12.000	1.000	4.000	0.000	1.000	1.000
2	2.000	87.000	545.000	77.000	4.000	151.000	19.000	4.000	2.000	10.000	2.000	12.000	49.000	2.000	10.000	2.000	6.000	1.000	0.000	0.000
3	2.000	13.000	126.000	687.000	2.000	21.000	58.000	5.000	1.000	0.000	2.000	2.000	43.000	4.000	9.000	1.000	5.000	0.000	1.000	0.000
4	3.000	11.000	63.000	643.000	5.000	51.000	78.000	22.000	4.000	0.000	9.000	3.000	51.000	1.000	6.000	2.000	7.000	1.000	3.000	0.000
5	3.000	140.000	78.000	18.000	2.000	649.000	10.000	4.000	4.000	0.000	0.000	1.000	64.000	0.000	8.000	3.000	3.000	0.000	1.000	0.000
6	4.000	12.000	44.000	200.000	7.000	20.000	451.000	62.000	9.000	0.000	17.000	3.000	117.000	4.000	9.000	3.000	10.000	0.000	3.000	0.000
7	3.000	5.000	5.000	7.000	1.000	5.000	16.000	761.000	67.000	0.000	10.000	6.000	54.000	15.000	8.000	5.000	18.000	3.000	1.000	0.000
8	2.000	6.000	3.000	8.000	1.000	1.000	24.000	78.000	783.000	0.000	8.000	2.000	37.000	2.000	4.000	10.000	15.000	4.000	8.000	0.000
9	4.000	6.000	0.000	2.000	3.000	1.000	5.000	21.000	10.000	0.000	852.000	0.000	57.000	1.000	13.000	3.000	11.000	0.000	5.000	0.000
10	1.000	3.000	0.000	1.000	1.000	0.000	6.000	4.000	1.000	0.000	930.000	0.000	32.000	3.000	3.000	2.000	8.000	4.000	0.000	0.000
11	6.000	18.000	10.000	4.000	0.000	1.000	6.000	5.000	2.000	0.000	0.000	820.000	41.000	5.000	1.000	1.000	68.000	3.000	0.000	0.000
12	10.000	42.000	56.000	146.000	0.000	17.000	346.000	103.000	10.000	0.000	7.000	6.000	166.000	36.000	27.000	7.000	3.000	0.000	2.000	0.000
13	34.000	16.000	7.000	6.000	2.000	10.000	16.000	9.000	11.000	0.000	4.000	1.000	95.000	686.000	59.000	13.000	12.000	2.000	6.000	1.000
14	7.000	28.000	3.000	3.000	1.000	4.000	8.000	11.000	5.000	0.000	8.000	4.000	35.000	14.000	824.000	4.000	27.000	0.000	1.000	0.000
15	33.000	5.000	1.000	2.000	0.000	1.000	1.000	2.000	0.000	0.000	1.000	2.000	42.000	12.000	3.000	841.000	11.000	22.000	15.000	3.000
16	12.000	5.000	3.000	4.000	0.000	0.000	7.000	9.000	7.000	0.000	5.000	7.000	24.000	8.000	11.000	10.000	783.000	6.000	9.000	0.000
17	11.000	4.000	0.000	2.000	0.000	0.000	2.000	5.000	6.000	0.000	3.000	2.000	24.000	7.000	1.000	12.000	39.000	607.000	14.000	201.000
18	21.000	2.000	5.000	0.000	0.000	2.000	6.000	6.000	10.000	0.000	1.000	3.000	19.000	69.000	19.000	6.000	365.000	23.000	216.000	2.000
19	165.000	1.000	4.000	2.000	1.000	4.000	1.000	1.000	3.000	0.000	7.000	1.000	13.000	37.000	5.000	266.000	87.000	10.000	20.000	0.000

```
Homogeneity: 0.5566
Completeness: 0.5911
```

```
V-measure: 0.5733
Adjusted Rand-Index: 0.4403
Adjusted Mutual Information Score: 0.5719
```

Euclidean UMAP when r = 5 :

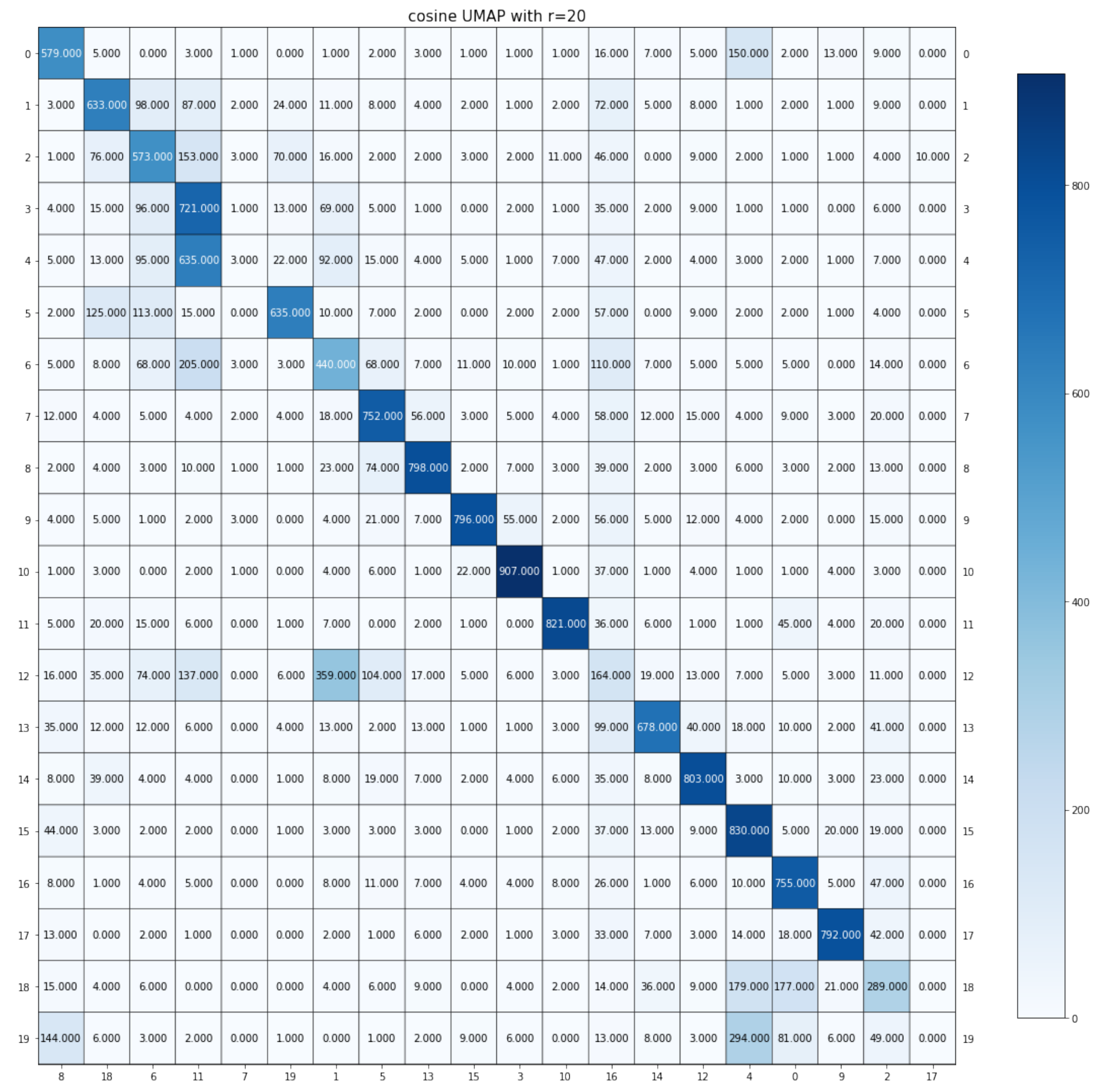


euclidean UMAP with r=5

	2	1	3	7	10	16	9	11	5	19	0	8	13	14	18	15	4	6	17	12
0	533.000	11.000	0.000	1.000	1.000	1.000	1.000	2.000	4.000	0.000	1.000	1.000	1.000	7.000	7.000	188.000	2.000	22.000	10.000	6.000
1	4.000	205.000	86.000	93.000	6.000	513.000	11.000	8.000	2.000	2.000	2.000	1.000	2.000	5.000	18.000	2.000	2.000	0.000	11.000	0.000
2	2.000	44.000	518.000	147.000	4.000	206.000	12.000	8.000	2.000	0.000	12.000	11.000	0.000	2.000	10.000	1.000	1.000	1.000	4.000	0.000
3	3.000	46.000	97.000	715.000	2.000	24.000	68.000	4.000	1.000	0.000	3.000	1.000	0.000	4.000	8.000	1.000	1.000	0.000	4.000	0.000
4	5.000	47.000	58.000	647.000	3.000	60.000	87.000	13.000	4.000	9.000	4.000	3.000	0.000	3.000	5.000	3.000	4.000	1.000	7.000	0.000
5	1.000	72.000	72.000	19.000	1.000	774.000	11.000	12.000	2.000	4.000	0.000	2.000	1.000	0.000	10.000	2.000	3.000	0.000	2.000	0.000
6	5.000	88.000	54.000	200.000	8.000	20.000	450.000	87.000	9.000	4.000	15.000	4.000	0.000	8.000	6.000	4.000	3.000	0.000	8.000	2.000
7	7.000	48.000	5.000	8.000	0.000	8.000	28.000	786.000	47.000	0.000	3.000	2.000	0.000	18.000	4.000	3.000	10.000	2.000	8.000	3.000
8	5.000	36.000	1.000	12.000	1.000	2.000	31.000	88.000	783.000	2.000	3.000	1.000	0.000	3.000	4.000	4.000	5.000	1.000	13.000	1.000
9	4.000	47.000	4.000	1.000	1.000	2.000	6.000	37.000	7.000	792.000	59.000	0.000	0.000	5.000	6.000	5.000	1.000	1.000	14.000	2.000
10	2.000	18.000	2.000	2.000	0.000	0.000	15.000	15.000	5.000	20.000	907.000	0.000	0.000	2.000	2.000	2.000	1.000	4.000	2.000	0.000
11	4.000	44.000	16.000	4.000	0.000	19.000	3.000	11.000	3.000	0.000	1.000	814.000	0.000	6.000	1.000	2.000	30.000	0.000	33.000	0.000
12	9.000	170.000	55.000	132.000	0.000	32.000	419.000	50.000	9.000	3.000	5.000	6.000	1.000	23.000	41.000	10.000	2.000	1.000	16.000	0.000
13	30.000	85.000	7.000	5.000	1.000	12.000	19.000	11.000	12.000	3.000	4.000	1.000	0.000	694.000	52.000	18.000	6.000	3.000	24.000	3.000
14	9.000	44.000	4.000	5.000	1.000	24.000	9.000	17.000	5.000	2.000	4.000	6.000	0.000	11.000	815.000	3.000	8.000	0.000	20.000	0.000
15	28.000	29.000	2.000	3.000	0.000	4.000	1.000	5.000	2.000	1.000	1.000	2.000	3.000	11.000	7.000	823.000	4.000	13.000	21.000	37.000
16	6.000	22.000	2.000	2.000	0.000	2.000	8.000	15.000	8.000	6.000	2.000	7.000	0.000	5.000	7.000	11.000	758.000	3.000	44.000	2.000
17	5.000	13.000	1.000	0.000	0.000	1.000	5.000	6.000	8.000	8.000	2.000	2.000	205.000	3.000	2.000	12.000	11.000	603.000	52.000	1.000
18	15.000	17.000	4.000	0.000	1.000	2.000	5.000	33.000	10.000	4.000	2.000	2.000	2.000	36.000	12.000	8.000	156.000	17.000	259.000	190.000
19	154.000	16.000	0.000	0.000	2.000	5.000	1.000	3.000	1.000	16.000	2.000	0.000	1.000	10.000	6.000	274.000	75.000	4.000	46.000	12.000

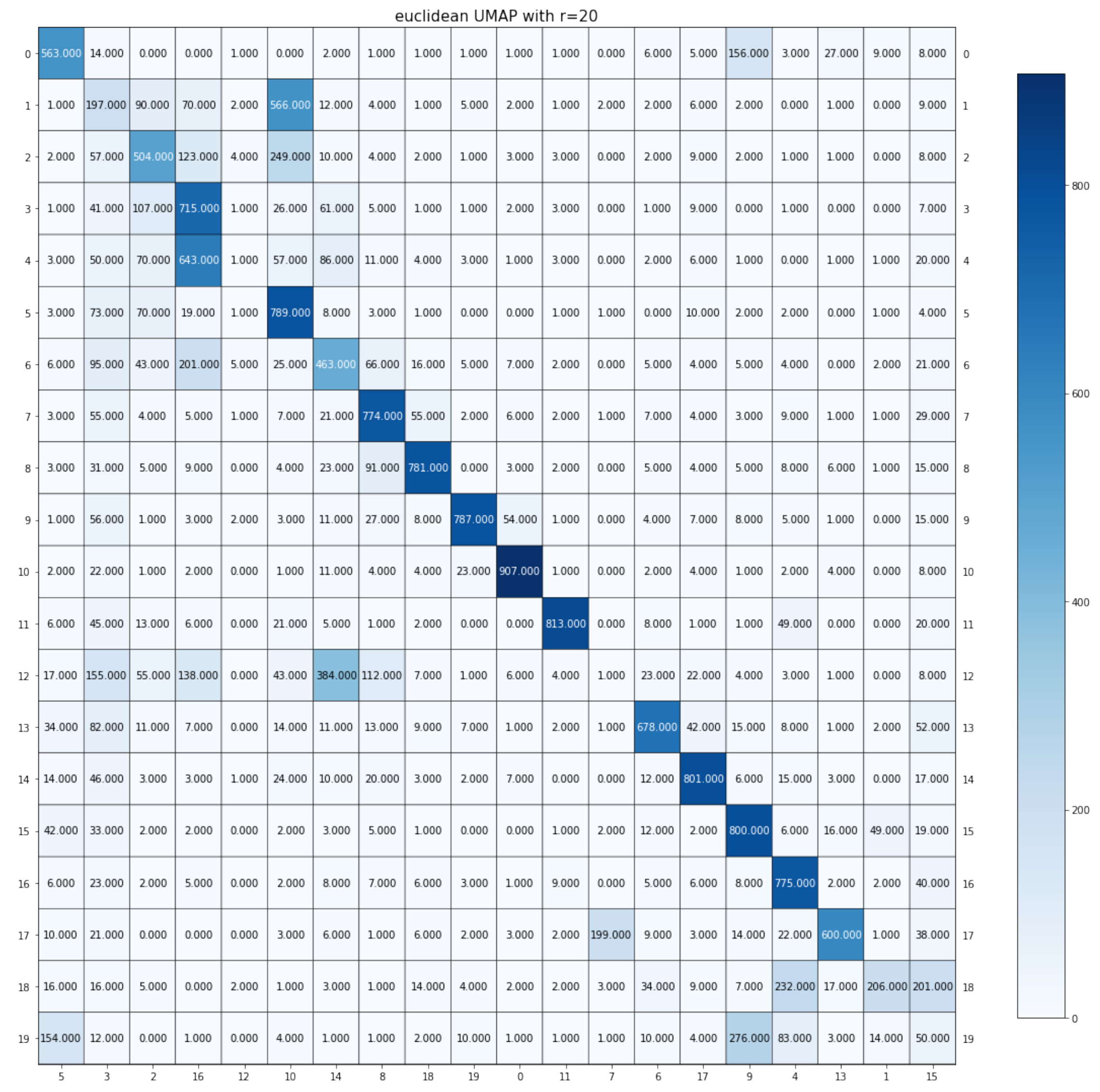
```
Homogeneity: 0.5651
Completeness: 0.5910
V-measure: 0.5778
Adjusted Rand-Index: 0.4451
Adjusted Mutual Information Score: 0.5764
```

Try r = 20 …
Cosine UMAP when r = 20 :



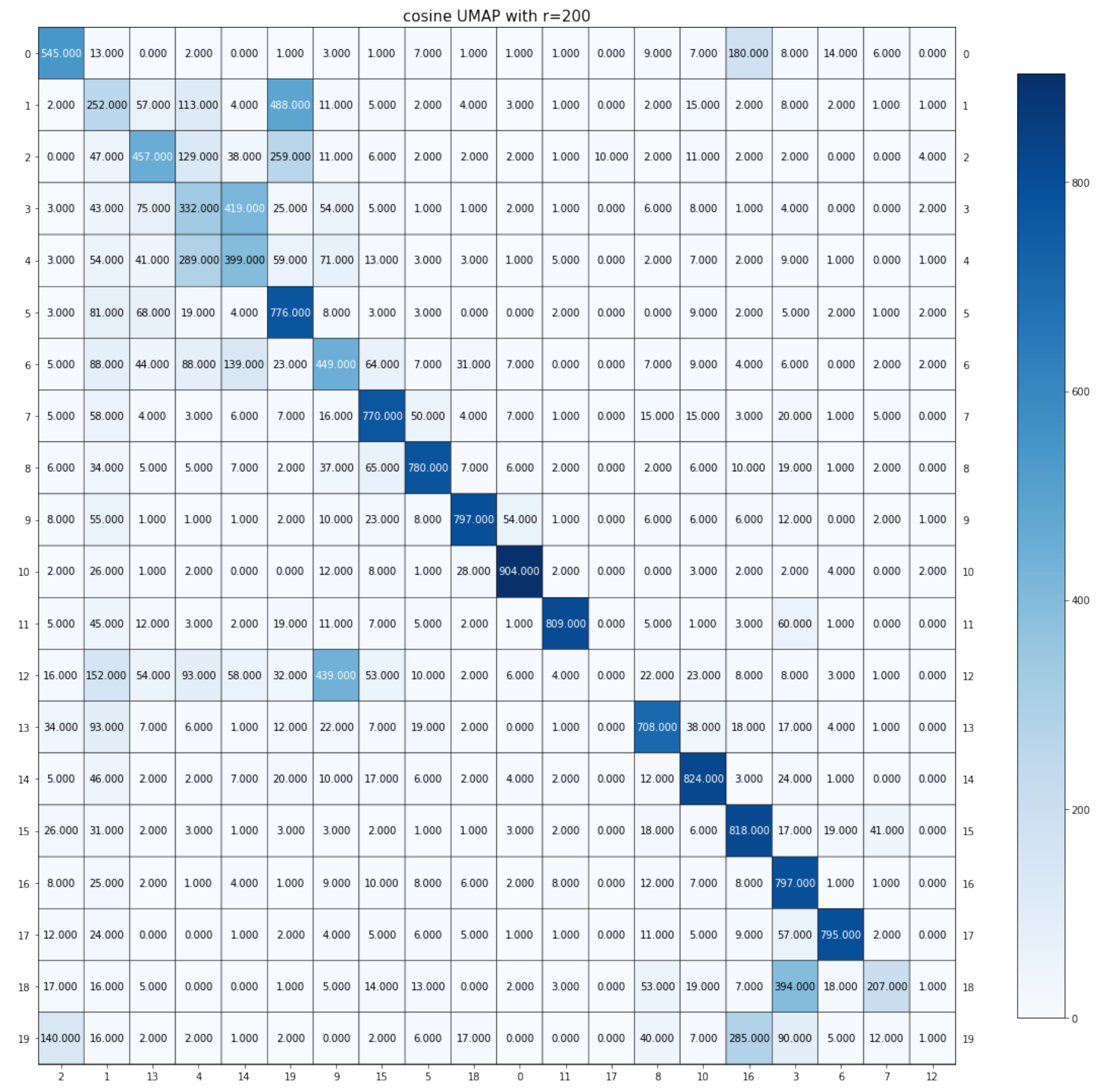
	8	18	6	11	7	19	1	5	13	15	3	10	16	14	12	4	0	9	2	17	
0	579.000	5.000	0.000	3.000	1.000	0.000	1.000	2.000	3.000	1.000	1.000	1.000	16.000	7.000	5.000	150.000	2.000	13.000	9.000	0.000	0
1	3.000	633.000	98.000	87.000	2.000	24.000	11.000	8.000	4.000	2.000	1.000	2.000	72.000	5.000	8.000	1.000	2.000	1.000	9.000	0.000	1
2	1.000	76.000	573.000	153.000	3.000	70.000	16.000	2.000	2.000	3.000	2.000	11.000	46.000	0.000	9.000	2.000	1.000	1.000	4.000	10.000	2
3	4.000	15.000	96.000	721.000	1.000	13.000	69.000	5.000	1.000	0.000	2.000	1.000	35.000	2.000	9.000	1.000	1.000	0.000	6.000	0.000	3
4	5.000	13.000	95.000	635.000	3.000	22.000	92.000	15.000	4.000	5.000	1.000	7.000	47.000	2.000	4.000	3.000	2.000	1.000	7.000	0.000	4
5	2.000	125.000	113.000	15.000	0.000	635.000	10.000	7.000	2.000	0.000	2.000	2.000	57.000	0.000	9.000	2.000	2.000	1.000	4.000	0.000	5
6	5.000	8.000	68.000	205.000	3.000	3.000	440.000	68.000	7.000	11.000	10.000	1.000	110.000	7.000	5.000	5.000	5.000	0.000	14.000	0.000	6
7	12.000	4.000	5.000	4.000	2.000	4.000	18.000	752.000	56.000	3.000	5.000	4.000	58.000	12.000	15.000	4.000	9.000	3.000	20.000	0.000	7
8	2.000	4.000	3.000	10.000	1.000	1.000	23.000	74.000	798.000	2.000	7.000	3.000	39.000	2.000	3.000	6.000	3.000	2.000	13.000	0.000	8
9	4.000	5.000	1.000	2.000	3.000	0.000	4.000	21.000	7.000	796.000	55.000	2.000	56.000	5.000	12.000	4.000	2.000	0.000	15.000	0.000	9
10	1.000	3.000	0.000	2.000	1.000	0.000	4.000	6.000	1.000	22.000	907.000	1.000	37.000	1.000	4.000	1.000	1.000	4.000	3.000	0.000	10
11	5.000	20.000	15.000	6.000	0.000	1.000	7.000	0.000	2.000	1.000	0.000	821.000	36.000	6.000	1.000	1.000	45.000	4.000	20.000	0.000	11
12	16.000	35.000	74.000	137.000	0.000	6.000	359.000	104.000	17.000	5.000	6.000	3.000	164.000	19.000	13.000	7.000	5.000	3.000	11.000	0.000	12
13	35.000	12.000	12.000	6.000	0.000	4.000	13.000	2.000	13.000	1.000	1.000	3.000	99.000	678.000	40.000	18.000	10.000	2.000	41.000	0.000	13
14	8.000	39.000	4.000	4.000	0.000	1.000	8.000	19.000	7.000	2.000	4.000	6.000	35.000	8.000	803.000	3.000	10.000	3.000	23.000	0.000	14
15	44.000	3.000	2.000	2.000	0.000	1.000	3.000	3.000	3.000	0.000	1.000	2.000	37.000	13.000	9.000	830.000	5.000	20.000	19.000	0.000	15
16	8.000	1.000	4.000	5.000	0.000	0.000	8.000	11.000	7.000	4.000	4.000	8.000	26.000	1.000	6.000	10.000	755.000	5.000	47.000	0.000	16
17	13.000	0.000	2.000	1.000	0.000	0.000	2.000	1.000	6.000	2.000	1.000	3.000	33.000	7.000	3.000	14.000	18.000	792.000	42.000	0.000	17
18	15.000	4.000	6.000	0.000	0.000	0.000	4.000	6.000	9.000	0.000	4.000	2.000	14.000	36.000	9.000	179.000	177.000	21.000	289.000	0.000	18
19	144.000	6.000	3.000	2.000	0.000	1.000	0.000	1.000	2.000	9.000	6.000	0.000	13.000	8.000	3.000	294.000	81.000	6.000	49.000	0.000	19

```
Homogeneity: 0.5709
Completeness: 0.5962
V-measure: 0.5833
Adjusted Rand-Index: 0.4623
Adjusted Mutual Information Score: 0.5819

Euclidean UMAP when r = 20 :
```



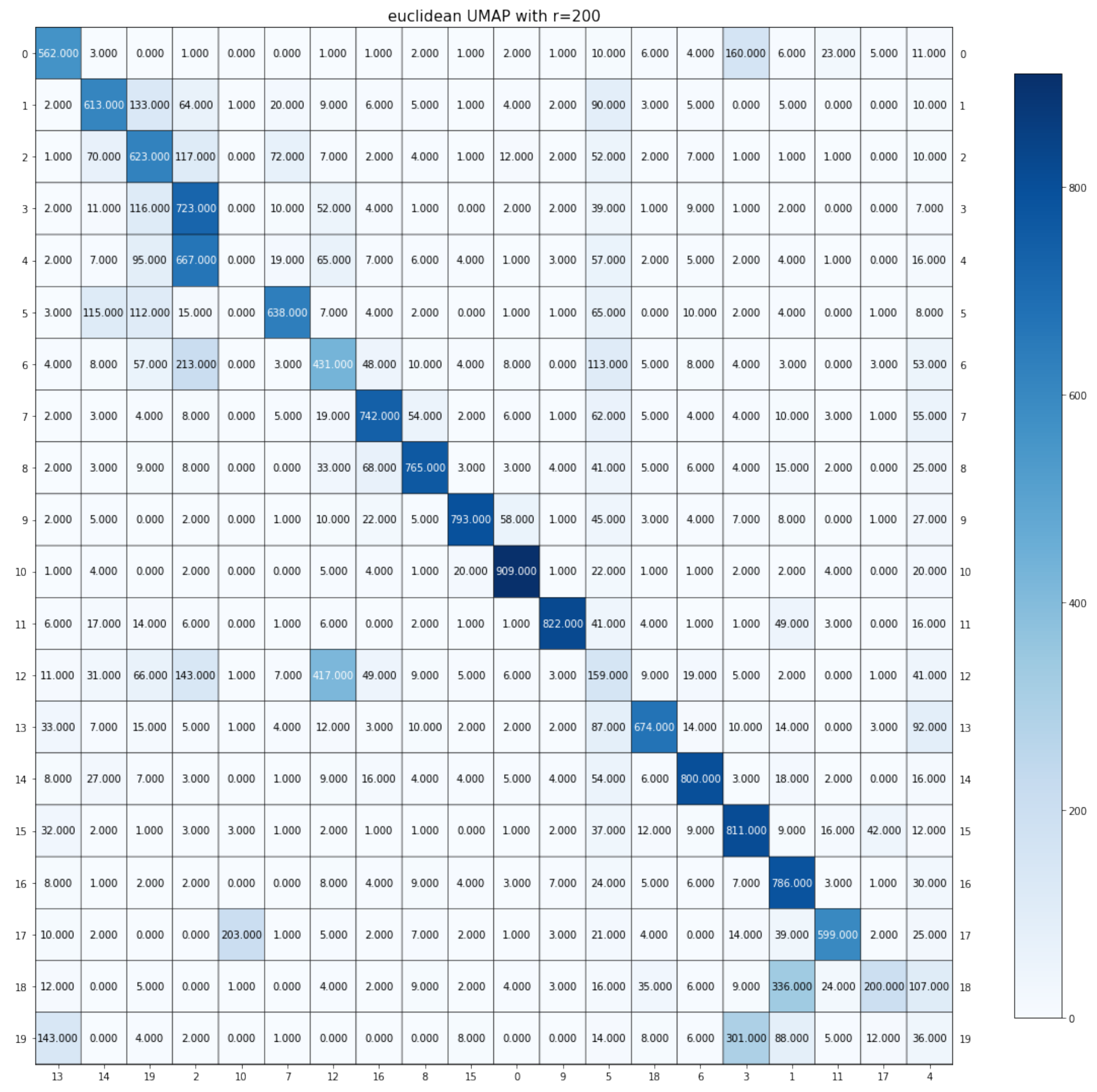
	5	3	2	16	12	10	14	8	18	19	0	11	7	6	17	9	4	13	1	15	
0	563.000	14.000	0.000	0.000	1.000	0.000	2.000	1.000	1.000	1.000	1.000	1.000	0.000	6.000	5.000	156.000	3.000	27.000	9.000	8.000	0
1	1.000	197.000	90.000	70.000	2.000	566.000	12.000	4.000	1.000	5.000	2.000	1.000	2.000	2.000	6.000	2.000	0.000	1.000	0.000	9.000	1
2	2.000	57.000	504.000	123.000	4.000	249.000	10.000	4.000	2.000	1.000	3.000	3.000	0.000	2.000	9.000	2.000	1.000	1.000	0.000	8.000	2
3	1.000	41.000	107.000	715.000	1.000	26.000	61.000	5.000	1.000	1.000	2.000	3.000	0.000	1.000	9.000	0.000	1.000	0.000	0.000	7.000	3
4	3.000	50.000	70.000	643.000	1.000	57.000	86.000	11.000	4.000	3.000	1.000	3.000	0.000	2.000	6.000	1.000	0.000	1.000	1.000	20.000	4
5	3.000	73.000	70.000	19.000	1.000	789.000	8.000	3.000	1.000	0.000	0.000	1.000	1.000	0.000	10.000	2.000	2.000	0.000	1.000	4.000	5
6	6.000	95.000	43.000	201.000	5.000	25.000	463.000	66.000	16.000	5.000	7.000	2.000	0.000	5.000	4.000	5.000	4.000	0.000	2.000	21.000	6
7	3.000	55.000	4.000	5.000	1.000	7.000	21.000	774.000	55.000	2.000	6.000	2.000	1.000	7.000	4.000	3.000	9.000	1.000	1.000	29.000	7
8	3.000	31.000	5.000	9.000	0.000	4.000	23.000	91.000	781.000	0.000	3.000	2.000	0.000	5.000	4.000	5.000	8.000	6.000	1.000	15.000	8
9	1.000	56.000	1.000	3.000	2.000	3.000	11.000	27.000	8.000	787.000	54.000	1.000	0.000	4.000	7.000	8.000	5.000	1.000	0.000	15.000	9
10	2.000	22.000	1.000	2.000	0.000	1.000	11.000	4.000	4.000	23.000	907.000	1.000	0.000	2.000	4.000	1.000	2.000	4.000	0.000	8.000	10
11	6.000	45.000	13.000	6.000	0.000	21.000	5.000	1.000	2.000	0.000	0.000	813.000	0.000	8.000	1.000	1.000	49.000	0.000	0.000	20.000	11
12	17.000	155.000	55.000	138.000	0.000	43.000	384.000	112.000	7.000	1.000	6.000	4.000	1.000	23.000	22.000	4.000	3.000	1.000	0.000	8.000	12
13	34.000	82.000	11.000	7.000	0.000	14.000	11.000	13.000	9.000	7.000	1.000	2.000	1.000	678.000	42.000	15.000	8.000	1.000	2.000	52.000	13
14	14.000	46.000	3.000	3.000	1.000	24.000	10.000	20.000	3.000	2.000	7.000	0.000	0.000	12.000	801.000	6.000	15.000	3.000	0.000	17.000	14
15	42.000	33.000	2.000	2.000	0.000	2.000	3.000	5.000	1.000	0.000	0.000	1.000	2.000	12.000	2.000	800.000	6.000	16.000	49.000	19.000	15
16	6.000	23.000	2.000	5.000	0.000	2.000	8.000	7.000	6.000	3.000	1.000	9.000	0.000	5.000	6.000	8.000	775.000	2.000	2.000	40.000	16
17	10.000	21.000	0.000	0.000	0.000	3.000	6.000	1.000	6.000	2.000	3.000	2.000	199.000	9.000	3.000	14.000	22.000	600.000	1.000	38.000	17
18	16.000	16.000	5.000	0.000	2.000	1.000	3.000	1.000	14.000	4.000	2.000	2.000	3.000	34.000	9.000	7.000	232.000	17.000	206.000	201.000	18
19	154.000	12.000	0.000	1.000	0.000	4.000	1.000	1.000	2.000	10.000	1.000	1.000	1.000	10.000	4.000	276.000	83.000	3.000	14.000	50.000	19

```
Homogeneity: 0.5691
Completeness: 0.5957
V-measure: 0.5821
Adjusted Rand-Index: 0.4443
Adjusted Mutual Information Score: 0.5807

Try r = 200 …
Cosine UMAP when r = 200 :
```



	2	1	13	4	14	19	9	15	5	18	0	11	17	8	10	16	3	6	7	12
0	545.000	13.000	0.000	2.000	0.000	1.000	3.000	1.000	7.000	1.000	1.000	1.000	0.000	9.000	7.000	180.000	8.000	14.000	6.000	0.000
1	2.000	252.000	57.000	113.000	4.000	488.000	11.000	5.000	2.000	4.000	3.000	1.000	0.000	2.000	15.000	2.000	8.000	2.000	1.000	1.000
2	0.000	47.000	457.000	129.000	38.000	259.000	11.000	6.000	2.000	2.000	2.000	1.000	10.000	2.000	11.000	2.000	2.000	0.000	0.000	4.000
3	3.000	43.000	75.000	332.000	419.000	25.000	54.000	5.000	1.000	1.000	2.000	1.000	0.000	6.000	8.000	1.000	4.000	0.000	0.000	2.000
4	3.000	54.000	41.000	289.000	399.000	59.000	71.000	13.000	3.000	3.000	1.000	5.000	0.000	2.000	7.000	2.000	9.000	1.000	0.000	1.000
5	3.000	81.000	68.000	19.000	4.000	776.000	8.000	3.000	3.000	0.000	0.000	2.000	0.000	0.000	9.000	2.000	5.000	2.000	1.000	2.000
6	5.000	88.000	44.000	88.000	139.000	23.000	449.000	64.000	7.000	31.000	7.000	0.000	0.000	7.000	9.000	4.000	6.000	0.000	2.000	2.000
7	5.000	58.000	4.000	3.000	6.000	7.000	16.000	770.000	50.000	4.000	7.000	1.000	0.000	15.000	15.000	3.000	20.000	1.000	5.000	0.000
8	6.000	34.000	5.000	5.000	7.000	2.000	37.000	65.000	780.000	7.000	6.000	2.000	0.000	2.000	6.000	10.000	19.000	1.000	2.000	0.000
9	8.000	55.000	1.000	1.000	1.000	2.000	10.000	23.000	8.000	797.000	54.000	1.000	0.000	6.000	6.000	6.000	12.000	0.000	2.000	1.000
10	2.000	26.000	1.000	2.000	0.000	0.000	12.000	8.000	1.000	28.000	904.000	2.000	0.000	0.000	3.000	2.000	2.000	4.000	0.000	2.000
11	5.000	45.000	12.000	3.000	2.000	19.000	11.000	7.000	5.000	2.000	1.000	809.000	0.000	5.000	1.000	3.000	60.000	1.000	0.000	0.000
12	16.000	152.000	54.000	93.000	58.000	32.000	439.000	53.000	10.000	2.000	6.000	4.000	0.000	22.000	23.000	8.000	8.000	3.000	1.000	0.000
13	34.000	93.000	7.000	6.000	1.000	12.000	22.000	7.000	19.000	2.000	0.000	1.000	0.000	708.000	38.000	18.000	17.000	4.000	1.000	0.000
14	5.000	46.000	2.000	2.000	7.000	20.000	10.000	17.000	6.000	2.000	4.000	2.000	0.000	12.000	824.000	3.000	24.000	1.000	0.000	0.000
15	26.000	31.000	2.000	3.000	1.000	3.000	3.000	2.000	1.000	1.000	3.000	2.000	0.000	18.000	6.000	818.000	17.000	19.000	41.000	0.000
16	8.000	25.000	2.000	1.000	4.000	1.000	9.000	10.000	8.000	6.000	2.000	8.000	0.000	12.000	7.000	8.000	797.000	1.000	1.000	0.000
17	12.000	24.000	0.000	0.000	1.000	2.000	4.000	5.000	6.000	5.000	1.000	1.000	0.000	11.000	5.000	9.000	57.000	795.000	2.000	0.000
18	17.000	16.000	5.000	0.000	0.000	1.000	5.000	14.000	13.000	0.000	2.000	3.000	0.000	53.000	19.000	7.000	394.000	18.000	207.000	1.000
19	140.000	16.000	2.000	2.000	1.000	2.000	0.000	2.000	6.000	17.000	0.000	0.000	0.000	40.000	7.000	285.000	90.000	5.000	12.000	1.000

```
Homogeneity: 0.5623
Completeness: 0.5895
V-measure: 0.5756
Adjusted Rand-Index: 0.4471
Adjusted Mutual Information Score: 0.5742

Euclidean UMAP when r = 200 :
```



	13	14	19	2	10	7	12	16	8	15	0	9	5	18	6	3	1	11	17	4	
0	562.000	3.000	0.000	1.000	0.000	0.000	1.000	1.000	2.000	1.000	2.000	1.000	10.000	6.000	4.000	160.000	6.000	23.000	5.000	11.000	0
1	2.000	613.000	133.000	64.000	1.000	20.000	9.000	6.000	5.000	1.000	4.000	2.000	90.000	3.000	5.000	0.000	5.000	0.000	0.000	10.000	1
2	1.000	70.000	623.000	117.000	0.000	72.000	7.000	2.000	4.000	1.000	12.000	2.000	52.000	2.000	7.000	1.000	1.000	1.000	0.000	10.000	2
3	2.000	11.000	116.000	723.000	0.000	10.000	52.000	4.000	1.000	0.000	2.000	2.000	39.000	1.000	9.000	1.000	2.000	0.000	0.000	7.000	3
4	2.000	7.000	95.000	667.000	0.000	19.000	65.000	7.000	6.000	4.000	1.000	3.000	57.000	2.000	5.000	2.000	4.000	1.000	0.000	16.000	4
5	3.000	115.000	112.000	15.000	0.000	638.000	7.000	4.000	2.000	0.000	1.000	1.000	65.000	0.000	10.000	2.000	4.000	0.000	1.000	8.000	5
6	4.000	8.000	57.000	213.000	0.000	3.000	431.000	48.000	10.000	4.000	8.000	0.000	113.000	5.000	8.000	4.000	3.000	0.000	3.000	53.000	6
7	2.000	3.000	4.000	8.000	0.000	5.000	19.000	742.000	54.000	2.000	6.000	1.000	62.000	5.000	4.000	4.000	10.000	3.000	1.000	55.000	7
8	2.000	3.000	9.000	8.000	0.000	0.000	33.000	68.000	765.000	3.000	3.000	4.000	41.000	5.000	6.000	4.000	15.000	2.000	0.000	25.000	8
9	2.000	5.000	0.000	2.000	0.000	1.000	10.000	22.000	5.000	793.000	58.000	1.000	45.000	3.000	4.000	7.000	8.000	0.000	1.000	27.000	9
10	1.000	4.000	0.000	2.000	0.000	0.000	5.000	4.000	1.000	20.000	909.000	1.000	22.000	1.000	1.000	2.000	2.000	4.000	0.000	20.000	10
11	6.000	17.000	14.000	6.000	0.000	1.000	6.000	0.000	2.000	1.000	1.000	822.000	41.000	4.000	1.000	1.000	49.000	3.000	0.000	16.000	11
12	11.000	31.000	66.000	143.000	1.000	7.000	417.000	49.000	9.000	5.000	6.000	3.000	159.000	9.000	19.000	5.000	2.000	0.000	1.000	41.000	12
13	33.000	7.000	15.000	5.000	1.000	4.000	12.000	3.000	10.000	2.000	2.000	2.000	87.000	674.000	14.000	10.000	14.000	0.000	3.000	92.000	13
14	8.000	27.000	7.000	3.000	0.000	1.000	9.000	16.000	4.000	4.000	5.000	4.000	54.000	6.000	800.000	3.000	18.000	2.000	0.000	16.000	14
15	32.000	2.000	1.000	3.000	3.000	1.000	2.000	1.000	1.000	0.000	1.000	2.000	37.000	12.000	9.000	811.000	9.000	16.000	42.000	12.000	15
16	8.000	1.000	2.000	2.000	0.000	0.000	8.000	4.000	9.000	4.000	3.000	7.000	24.000	5.000	6.000	7.000	786.000	3.000	1.000	30.000	16
17	10.000	2.000	0.000	0.000	203.000	1.000	5.000	2.000	7.000	2.000	1.000	3.000	21.000	4.000	0.000	14.000	39.000	599.000	2.000	25.000	17
18	12.000	0.000	5.000	0.000	1.000	0.000	4.000	2.000	9.000	2.000	4.000	3.000	16.000	35.000	6.000	9.000	336.000	24.000	200.000	107.000	18
19	143.000	0.000	4.000	2.000	0.000	1.000	0.000	0.000	0.000	8.000	0.000	0.000	14.000	8.000	6.000	301.000	88.000	5.000	12.000	36.000	19

```
Homogeneity: 0.5852
Completeness: 0.6009
V-measure: 0.5930
Adjusted Rand-Index: 0.4623
Adjusted Mutual Information Score: 0.5916
Done
```

0.13 Question 12

Analyze the contingency matrices. Which setting works best and why?

Ans:

From the contingency metrices we found above, we can see that there are no many differences between the euclidean UMAP with different r. Therefore, setting $r = 20$ is good enough for euclidean UMAP.

As of cosine UMAP, setting $r = 20$ gives us a better result. Setting r equals to other numbers all gives more outliers or categories being distributed to other categories.

```
[ ]: avg_cos = []
     avg_euc = []
     for i in range(len(rs11)):
       tmp_cos = 0
       tmp_cos += umap_cos_homo[i] / (sum(umap_cos_homo) / len(umap_cos_homo))
       tmp_cos += umap_cos_comp[i] / (sum(umap_cos_comp) / len(umap_cos_comp))
       tmp_cos += umap_cos_vmes[i] / (sum(umap_cos_vmes) / len(umap_cos_vmes))
       tmp_cos += umap_cos_ranI[i] / (sum(umap_cos_ranI) / len(umap_cos_ranI))
       tmp_cos += umap_cos_muts[i] / (sum(umap_cos_muts) / len(umap_cos_muts))
       avg_cos.append(tmp_cos / 5)
       tmp_euc = 0
       tmp_euc += umap_euc_homo[i] / (sum(umap_euc_homo) / len(umap_euc_homo))
       tmp_euc += umap_euc_comp[i] / (sum(umap_euc_comp) / len(umap_euc_comp))
       tmp_euc += umap_euc_vmes[i] / (sum(umap_euc_vmes) / len(umap_euc_vmes))
       tmp_euc += umap_euc_ranI[i] / (sum(umap_euc_ranI) / len(umap_euc_ranI))
       tmp_euc += umap_euc_muts[i] / (sum(umap_euc_muts) / len(umap_euc_muts))
       avg_euc.append(tmp_euc / 5)
     for i in range(len(avg_cos)):
       print("Average normalized scores for cosine UMAP when r =", rs11[i], ":",␣
      ↪avg_cos[i])
     print("")
     for i in range(len(avg_euc)):
       print("Average normalized scores for euclidean UMAP when r =", rs11[i], ":",␣
      ↪avg_euc[i])

     best_r_cos = rs11[avg_cos.index(max(avg_cos))]
     # best_r_cos = 5
     best_r_euc = rs11[avg_euc.index(max(avg_euc))]
     print("\nBest value of r for cosine UMAP:", best_r_cos, ", its average␣
      ↪normalized scores:", avg_cos[rs11.index(best_r_cos)])
     print("Best value of r for euclidean UMAP:", best_r_euc, ", its average␣
      ↪normalized scores:", avg_euc[rs11.index(best_r_euc)])
```

```
Average normalized scores for cosine UMAP when r = 5 : 0.9901320195164086
Average normalized scores for cosine UMAP when r = 20 : 1.0136027226843964
Average normalized scores for cosine UMAP when r = 200 : 0.9962652577991953

Average normalized scores for euclidean UMAP when r = 5 : 0.9886982164174374
Average normalized scores for euclidean UMAP when r = 20 : 0.9942207323171074
Average normalized scores for euclidean UMAP when r = 200 : 1.0170810512654551

Best value of r for cosine UMAP: 20 , its average normalized scores:
1.0136027226843964
```

```
Best value of r for euclidean UMAP: 200 , its average normalized scores:
1.0170810512654551
```

What about for each metric choice?

Ans:

From the result above, we calculated the average normalized scores for each combination.

It also indicates that there are no many differences between the euclidean UMAP with different r. Although the scores when $r = 200$ is a little bit better, choosing $r = 20$ is good enough considering to the calculation time.

On the other hand, a good choice of r for cosine UMAP is 20.

Comparing the scores of these two UMAP, cosine UMAP with $r = 20$ works best and we choose it for the following comparison.

0.14 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.

Ans:

Comparin the clustering results across the 4 choices, we can see that the performance of UMAP-reduced is better than other 3 results. Therefore, we suggest **UMAP-reduced (cosine UMAP with $r = 20$)** is best for the K-Means clustering task on the 20-class text data.

[ ]:
```
def print_scores(title_name, score_homo, score_comp, score_vmes, score_ranI,␣
↪score_muts):
  print(title_name)
  print("Homogeneity: %0.4f" % score_homo)
  print("Completeness: %0.4f" % score_comp)
  print("V-measure: %0.4f" % score_vmes)
  print("Adjusted Rand-Index: %0.4f" % score_ranI)
  print("Adjusted Mutual Information Score: %0.4f \n" % score_muts)
```

[ ]:
```
kmm_o = KMeans(n_clusters=20, init='k-means++', max_iter=2000, n_init=50,␣
↪random_state=0)
kmm_o.fit(X_a_tfidf)
```

[ ]:
```
print("scores for sparse TF-IDF representation:")
print_5_measure_scores(y_a_data, kmm_o.labels_)
print("")
br_svd_i = rs10.index(best_a_svd)
br_nmf_i = rs10.index(best_a_nmf)
br_cos_i = avg_cos.index(max(avg_cos))
```

```
print_scores("scores for SVD when r = " + str(best_a_svd) + ":",␣
 ↪a_svd_homo[br_svd_i], a_svd_comp[br_svd_i], a_svd_vmes[br_svd_i],␣
 ↪a_svd_ranI[br_svd_i], a_svd_muts[br_svd_i])
print_scores("scores for NMF when r = " + str(best_a_nmf) + ":",␣
 ↪a_nmf_homo[br_nmf_i], a_nmf_comp[br_nmf_i], a_nmf_vmes[br_nmf_i],␣
 ↪a_nmf_ranI[br_nmf_i], a_nmf_muts[br_nmf_i])
print_scores("scores for cosine UMAP when r = " + str(rs11[br_cos_i]) + ":",␣
 ↪umap_cos_homo[br_cos_i], umap_cos_comp[br_cos_i], umap_cos_vmes[br_cos_i],␣
 ↪umap_cos_ranI[br_cos_i], umap_euc_muts[br_cos_i])
```

```
scores for sparse TF-IDF representation:
Homogeneity: 0.3470
Completeness: 0.3928
V-measure: 0.3684
Adjusted Rand-Index: 0.1265
Adjusted Mutual Information Score: 0.3663

scores for SVD when r = 20:
Homogeneity: 0.3330
Completeness: 0.3756
V-measure: 0.3530
Adjusted Rand-Index: 0.1202
Adjusted Mutual Information Score: 0.3508

scores for NMF when r = 10:
Homogeneity: 0.2953
Completeness: 0.3275
V-measure: 0.3106
Adjusted Rand-Index: 0.1056
Adjusted Mutual Information Score: 0.3082

scores for cosine UMAP when r = 20:
Homogeneity: 0.5709
Completeness: 0.5962
V-measure: 0.5833
Adjusted Rand-Index: 0.4623
Adjusted Mutual Information Score: 0.5807
```

0.15 Question 14

Use UMAP to reduce the dimensionality properly, and perform Agglomerative clustering with n_clusters=20 . Compare the performance of "ward" and "single" linkage criteria.

Report the five clustering evaluation metrics for each case.

Ans:

From the five clustering evaluation metrics indicate below, we can find that the performance of

Agglomerative Clustering with single linkage criterion is much worse than ward linkage criterion one did.

It might because of the using method behind these two criterions. The ward linkage criterion minimizes the variance of the clusters being merged. The single linkage criterion minimizes the distance between all observations of the two sets. Therefore, single linkage criterion is not robust enough to handle this high dimensional problem. It fails to deal with the noise or outlier data.

[ ]:
```
best_umap = umap.UMAP(n_components=rs11[br_cos_i], metric='cosine')
X_umap = best_umap.fit_transform(X_a_tfidf)
agg_w = AgglomerativeClustering(n_clusters=20, linkage='ward').fit(X_umap)
agg_s = AgglomerativeClustering(n_clusters=20, linkage='single').fit(X_umap)

print("Agglomerative Clustering with ward linkage criterion:")
print_5_measure_scores(y_a_data, agg_w.labels_)
print("\nAgglomerative Clustering with single linkage criterion:")
print_5_measure_scores(y_a_data, agg_s.labels_)
```

```
Agglomerative Clustering with ward linkage criterion:
Homogeneity: 0.5594
Completeness: 0.5925
V-measure: 0.5755
Adjusted Rand-Index: 0.4295
Adjusted Mutual Information Score: 0.5741

Agglomerative Clustering with single linkage criterion:
Homogeneity: 0.0192
Completeness: 0.3767
V-measure: 0.0365
Adjusted Rand-Index: 0.0006
Adjusted Mutual Information Score: 0.0316
```

0.16 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.

Ans:

From the five clustering evaluation metrics we found, the best min_cluster_size to use here is 100.

[ ]:
```
cluster_sizes = [20, 100, 200]
hdbs_homo = []
hdbs_comp = []
hdbs_vmes = []
hdbs_ranI = []
```

```
hdbs_muts = []

for min_size in cluster_sizes:
  print("\nmin_cluster_size =", min_size, ":")
  y_pred_hdbs = hdbscan.HDBSCAN(min_cluster_size=min_size).fit_predict(X_umap)
  hdbs_homo.append(homogeneity_score(y_a_data, y_pred_hdbs))
  hdbs_comp.append(completeness_score(y_a_data, y_pred_hdbs))
  hdbs_vmes.append(v_measure_score(y_a_data, y_pred_hdbs))
  hdbs_ranI.append(adjusted_rand_score(y_a_data, y_pred_hdbs))
  hdbs_muts.append(adjusted_mutual_info_score(y_a_data, y_pred_hdbs))
  print_5_measure_scores(y_a_data, y_pred_hdbs)
```

```
min_cluster_size = 20 :
Homogeneity: 0.4253
Completeness: 0.4479
V-measure: 0.4363
Adjusted Rand-Index: 0.0777
Adjusted Mutual Information Score: 0.4236

min_cluster_size = 100 :
Homogeneity: 0.4157
Completeness: 0.6266
V-measure: 0.4998
Adjusted Rand-Index: 0.2263
Adjusted Mutual Information Score: 0.4989

min_cluster_size = 200 :
Homogeneity: 0.4200
Completeness: 0.6166
V-measure: 0.4997
Adjusted Rand-Index: 0.2191
Adjusted Mutual Information Score: 0.4987
```

```
[ ]: avg_hdbs = []
for i in range(len(cluster_sizes)):
  tmp_hdbs = 0
  tmp_hdbs += hdbs_homo[i] / (sum(hdbs_homo) / len(hdbs_homo))
  tmp_hdbs += hdbs_comp[i] / (sum(hdbs_comp) / len(hdbs_comp))
  tmp_hdbs += hdbs_vmes[i] / (sum(hdbs_vmes) / len(hdbs_vmes))
  tmp_hdbs += hdbs_ranI[i] / (sum(hdbs_ranI) / len(hdbs_ranI))
  tmp_hdbs += hdbs_muts[i] / (sum(hdbs_muts) / len(hdbs_muts))
  avg_hdbs.append(tmp_hdbs / 5)
for i in range(len(avg_hdbs)):
  print("Average normalized scores for HDBSCAN when min_cluster_size =",␣
 ↪cluster_sizes[i], ":", avg_hdbs[i])
```

```
best_mcs = cluster_sizes[avg_hdbs.index(max(avg_hdbs))]
print("\nBest value of min_cluster_size for HDBSCAN:", best_mcs, ", its average␣
↪normalized scores:", max(avg_hdbs))
```

```
Average normalized scores for HDBSCAN when min_cluster_size = 20 :
0.8115684759351203
Average normalized scores for HDBSCAN when min_cluster_size = 100 :
1.0991266633883054
Average normalized scores for HDBSCAN when min_cluster_size = 200 :
1.0893048606765743

Best value of min_cluster_size for HDBSCAN: 100 , its average normalized scores:
1.0991266633883054
```

0.17 Question 16

Contingency matrix

Plot the contingency matrix for the best clustering model from Question 15. How many clusters are given by the model?

Interpret the contingency matrix considering the answer to these questions.

Ans:

According to our finding, there are total 10 major clusters given by the model. Furthermore, you can find there are many categories being distributed to other categories as a cluster.

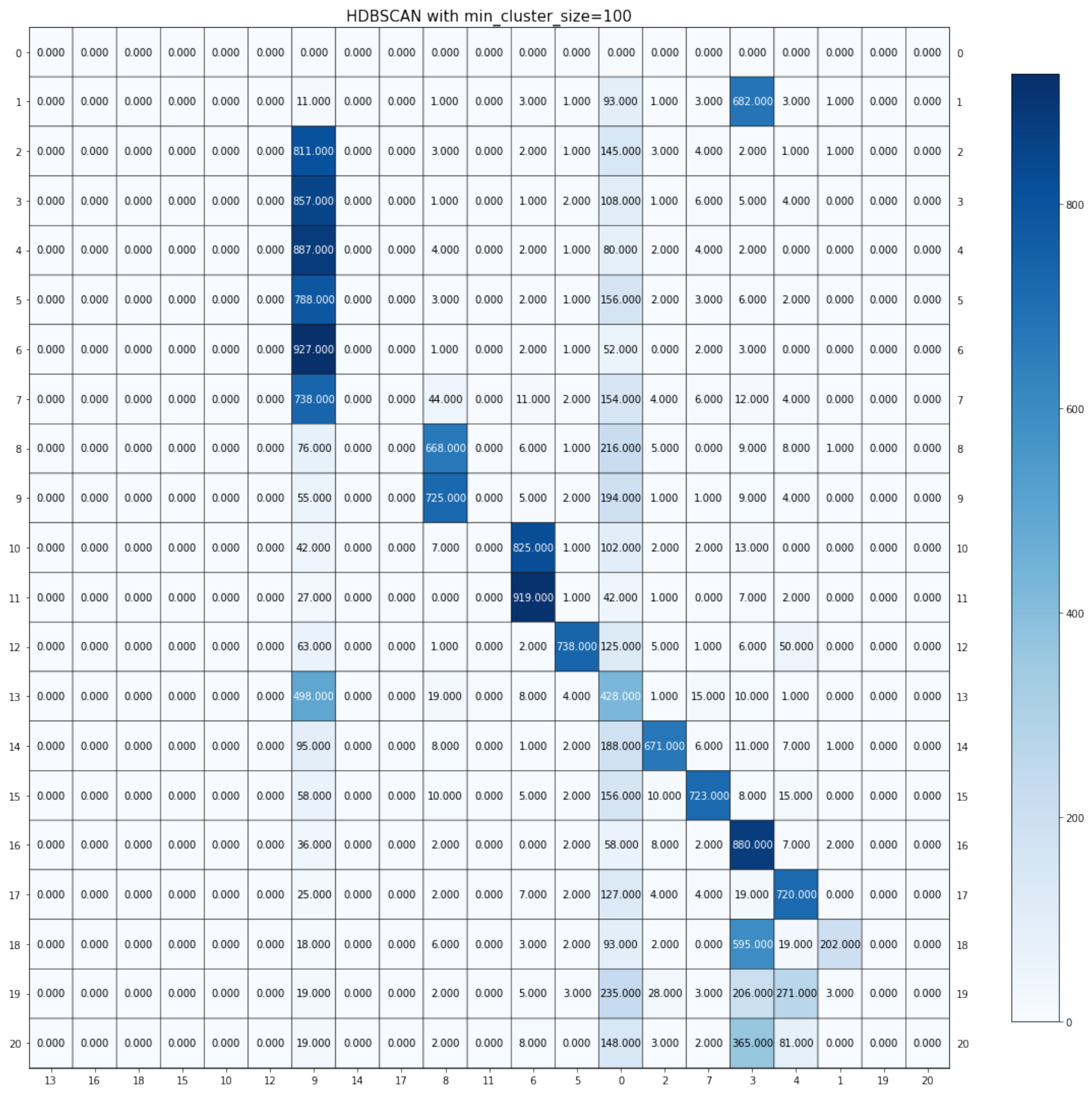
It might because of the reason that density based clustering relies on having enough data to separate dense areas. In higher dimensional spaces this becomes more difficult, and hence requires more data. Which makes HDBSCAN hard to perform well in this high dimensional data.

What does "-1" mean for the clustering labels?

Ans:

The "-1" in the clustering means noise or outlier sammples that can not been classfied into a cluster by the algorithms.

```
[ ]: # print contingency matrix
by_pred_hdbs = hdbscan.HDBSCAN(min_cluster_size=best_mcs).fit_predict(X_umap)
cm16 = confusion_matrix(y_a_data, by_pred_hdbs)
rows, cols = linear_sum_assignment(cm16, maximize=True)
plot_mat(cm16[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
↪title='HDBSCAN with min_cluster_size=' + str(best_mcs), size=(15,15))
```



0.18 Question 17

Based on your experiments, which dimensionality reduction technique and clustering 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 in a pickle file.

Module	Alternatives	Hyperparameters
Dimensionality Reduction	None	N/A
Dimensionality Reduction	SVD	r = [5, 20, 200]
Dimensionality Reduction	NMF	r = [5, 20, 200]
Dimensionality Reduction	UMAP	n_components = [5, 20, 200]
Clustering	K-Means	k = [10, 20, 50]
Clustering	Agglomerative Clustering	n_clusters = [20]

Module	Alternatives	Hyperparameters
Clustering	HDBSCAN	min_cluster_size = [100, 200]

Ans:

The combination of **UMAP using 'cosine' matrix and n_components setting to 5 plus K-Means using k = 20** works best together for 20-class text data. we will analysis the reason of it in the following cells.

[68]:
```
def pkl_save(path, data, filename):
  with open(path + filename, 'wb') as f:
    # compressed_file = bz2.BZ2File(f, 'w')
    pickle.dump(data, f)
```

[ ]:
```
dataset_all = fetch_20newsgroups(subset = 'all', shuffle = True, random_state =␣
↪0, remove=('headers','footers'))
vec = CountVectorizer(stop_words='english', min_df=3)
tfidf = TfidfTransformer()
X_vec = vec.fit_transform(dataset_all.data)
X_tfidf = tfidf.fit_transform(X_vec)
y_data = dataset_all.target
```

[76]:
```
# modify this line to your local path (you don't need this cell if you already␣
↪specify your path in the first cell)
# path = '/content/drive/MyDrive/Colab Notebooks/ECE219/Project2/'
# path = '/Users/behind/Desktop/UCLA/Winter 2023/EC ENGR 219/Project2/pickle␣
↪file/'
```

[ ]:
```
# None
pkl_save(path, X_tfidf, 'none.pkl')

rs17 = [5, 20, 200]
for i in range(len(rs17)):
  print("\ntry r =", rs17[i], "...")
  # SVD
  print("using SVD ...")
  svd_tmp = TruncatedSVD(n_components=rs17[i], random_state=0)
  X_svd = svd_tmp.fit_transform(X_tfidf)
  pkl_save(path, X_svd, 'svd_' + str(rs17[i]) + '.pkl')

  # NMF
  print("using NMF ...")
  nmf_tmp = NMF(n_components=rs17[i], init='random', random_state=0)
  X_nmf = nmf_tmp.fit_transform(X_tfidf)
  pkl_save(path, X_nmf, 'nmf_' + str(rs17[i]) + '.pkl')

  # UMAP
  print("using UMAP ...")
  umap_tmp = umap.UMAP(n_components=rs17[i], metric='cosine')
  X_umap = umap_tmp.fit_transform(X_tfidf)
  pkl_save(path, X_umap, 'umap_' + str(rs17[i]) + '.pkl')
```

```
try r = 5 …
using SVD …
using NMF …
using UMAP …

try r = 20 …
using SVD …
using NMF …
using UMAP …

try r = 200 …
using SVD …
using NMF …
```

```
/usr/local/lib/python3.8/dist-packages/sklearn/decomposition/_nmf.py:1637:
ConvergenceWarning: Maximum number of iterations 200 reached. Increase it to
improve convergence.
  warnings.warn(
```

```
using UMAP …
```

[69]:
```
def pkl_read(path, filename):
  with open(path + filename, 'rb') as f:
    # compressed_file = bz2.BZ2File(f, 'r')
    X_dr = pickle.load(f)
  return X_dr
DRs = ['none.pkl',
       'svd_5.pkl', 'svd_20.pkl', 'svd_200.pkl',
       'nmf_5.pkl', 'nmf_20.pkl', 'nmf_200.pkl',
       'umap_5.pkl', 'umap_20.pkl', 'umap_200.pkl']
```

[ ]:
```
ks = [10, 20, 50]
km_scores = [ [0] * len(DRs) for i in range(3)]
for i in range(len(ks)):
  print("\nk =", ks[i], "...")
  kmeans_tmp = KMeans(n_clusters=ks[i], init='k-means++', max_iter=2000,␣
↪n_init=50, random_state=0)
  for j in range(len(DRs)):
    print("Current dr:", DRs[j])
    # reading Dimensionality Reduction data
    X_dr = pkl_read(path, DRs[j])
    kmeans_fit = kmeans_tmp.fit(X_dr)
    # To increase efficiency, we calculate the average score of each␣
↪combination instead.
    tmp_s = 0
    tmp_s += homogeneity_score(y_data, kmeans_fit.labels_)
    tmp_s += completeness_score(y_data, kmeans_fit.labels_)
    tmp_s += v_measure_score(y_data, kmeans_fit.labels_)
    tmp_s += adjusted_rand_score(y_data, kmeans_fit.labels_)
    tmp_s += adjusted_mutual_info_score(y_data, kmeans_fit.labels_)
    tmp_s /= 5
    km_scores[i][j] = tmp_s
    print("Kmeans when k =", ks[i], ", average score:", tmp_s)
```

```
k = 10 …
Current dr: none.pkl
Kmeans when k = 10 , average score: 0.3128862783009779
Current dr: svd_5.pkl
Kmeans when k = 10 , average score: 0.2882612154581411
Current dr: svd_20.pkl
Kmeans when k = 10 , average score: 0.28103632815154544
Current dr: svd_200.pkl
Kmeans when k = 10 , average score: 0.2739717062086616
Current dr: nmf_5.pkl
Kmeans when k = 10 , average score: 0.24139614802401646
Current dr: nmf_20.pkl
Kmeans when k = 10 , average score: 0.21331247059166852
Current dr: nmf_200.pkl
Kmeans when k = 10 , average score: 0.032385718417162965
Current dr: umap_5.pkl
Kmeans when k = 10 , average score: 0.497940596596295
Current dr: umap_20.pkl
Kmeans when k = 10 , average score: 0.5023552400963418
Current dr: umap_200.pkl
Kmeans when k = 10 , average score: 0.5034860305394875

k = 20 …
Current dr: none.pkl
Kmeans when k = 20 , average score: 0.319977761965902
Current dr: svd_5.pkl
Kmeans when k = 20 , average score: 0.292763515525936
Current dr: svd_20.pkl
Kmeans when k = 20 , average score: 0.30654759487815475
Current dr: svd_200.pkl
Kmeans when k = 20 , average score: 0.31032718027283984
Current dr: nmf_5.pkl
Kmeans when k = 20 , average score: 0.2396382853881612
Current dr: nmf_20.pkl
```

```
Kmeans when k = 20 , average score: 0.2627086703407877
Current dr: nmf_200.pkl
Kmeans when k = 20 , average score: 0.11487118434220356
Current dr: umap_5.pkl
Kmeans when k = 20 , average score: 0.5641612641269881
Current dr: umap_20.pkl
Kmeans when k = 20 , average score: 0.5488304758703203
Current dr: umap_200.pkl
Kmeans when k = 20 , average score: 0.5568396812973211

k = 50 …
Current dr: none.pkl
Kmeans when k = 50 , average score: 0.346192788364497
Current dr: svd_5.pkl
Kmeans when k = 50 , average score: 0.28413792519012615
Current dr: svd_20.pkl
Kmeans when k = 50 , average score: 0.32867074172086935
Current dr: svd_200.pkl
Kmeans when k = 50 , average score: 0.3389048121584812
Current dr: nmf_5.pkl
Kmeans when k = 50 , average score: 0.23368389238229748
Current dr: nmf_20.pkl
Kmeans when k = 50 , average score: 0.3047935570521014
Current dr: nmf_200.pkl
Kmeans when k = 50 , average score: 0.13782058586457885
Current dr: umap_5.pkl
Kmeans when k = 50 , average score: 0.5148044954293607
Current dr: umap_20.pkl
Kmeans when k = 50 , average score: 0.5303504605460957
Current dr: umap_200.pkl
Kmeans when k = 50 , average score: 0.5273297101254659
```

[ ]:
```
agg_n = 20
agg_scores = [ [0] * len(DRs)]
for j in range(len(DRs)):
  print("Current dr:", DRs[j])
  # it takes over 1hr to finish the none.pkl, we save the result that we have␣
 ↪found before in advance here
  if (DRs[j] == 'none.pkl'):
    print("Agglomerative Clustering when n_clusters =", agg_n, ", average score:
 ↪", 0.3437828455142916)
    agg_scores[0][j] = 0.3437828455142916
    continue
  # reading Dimensionality Reduction data
  X_dr = pkl_read(path, DRs[j])
  # if (DRs[j] == 'none.pkl'):
  #   X_dr = X_dr.toarray()
  agg_tmp = AgglomerativeClustering(n_clusters=agg_n, linkage='ward').fit(X_dr)
  # To increase efficiency, we calculate the average score of each combination␣
↪instead.
  tmp_s = 0
  tmp_s += homogeneity_score(y_data, agg_tmp.labels_)
  tmp_s += completeness_score(y_data, agg_tmp.labels_)
  tmp_s += v_measure_score(y_data, agg_tmp.labels_)
  tmp_s += adjusted_rand_score(y_data, agg_tmp.labels_)
  tmp_s += adjusted_mutual_info_score(y_data, agg_tmp.labels_)
  tmp_s /= 5
  agg_scores[0][j] = tmp_s
  print("Agglomerative Clustering when n_clusters =", agg_n, ", average score:
↪", tmp_s)
```

```
Current dr: none.pkl
Agglomerative Clustering when n_clusters = 20 , average score:
0.3437828455142916
Current dr: svd_5.pkl
Agglomerative Clustering when n_clusters = 20 , average score: 0.281758252396262
Current dr: svd_20.pkl
Agglomerative Clustering when n_clusters = 20 , average score:
0.35480890471057575
Current dr: svd_200.pkl
Agglomerative Clustering when n_clusters = 20 , average score:
0.32757379932917774
Current dr: nmf_5.pkl
Agglomerative Clustering when n_clusters = 20 , average score:
0.23868230288974474
Current dr: nmf_20.pkl
Agglomerative Clustering when n_clusters = 20 , average score: 0.331828295098176
Current dr: nmf_200.pkl
Agglomerative Clustering when n_clusters = 20 , average score:
0.10838508498003048
Current dr: umap_5.pkl
Agglomerative Clustering when n_clusters = 20 , average score:
0.5437828303701593
Current dr: umap_20.pkl
Agglomerative Clustering when n_clusters = 20 , average score:
0.5311649868566655
Current dr: umap_200.pkl
Agglomerative Clustering when n_clusters = 20 , average score:
0.5290589073081226
```

```
[ ]: mcss = [100, 200]
     hdbs_scores = [ [0] * len(DRs) for i in range(2)]
     for i in range(len(mcss)):
       print("\nmin_cluster_size =", mcss[i], "...")
```

```
  for j in range(len(DRs)):
    print("Current dr:", DRs[j])
    # reading Dimensionality Reduction data
    X_dr = pkl_read(path, DRs[j])
    y_pred_hdbs = hdbscan.HDBSCAN(min_cluster_size=mcss[i],␣
↪allow_single_cluster=True).fit_predict(X_dr)
    # To increase efficiency, we calculate the average score of each␣
↪combination instead.
    tmp_s = 0
    tmp_s += homogeneity_score(y_data, y_pred_hdbs)
    tmp_s += completeness_score(y_data, y_pred_hdbs)
    tmp_s += v_measure_score(y_data, y_pred_hdbs)
    tmp_s += adjusted_rand_score(y_data, y_pred_hdbs)
    tmp_s += adjusted_mutual_info_score(y_data, y_pred_hdbs)
    tmp_s /= 5
    hdbs_scores[i][j] = tmp_s
    print("HDBSCAN when min_cluster_size =", mcss[i], ", average score:", tmp_s)
```

```
min_cluster_size = 100 …
Current dr: none.pkl
HDBSCAN when min_cluster_size = 100 , average score: 0.016708235408510047
Current dr: svd_5.pkl
HDBSCAN when min_cluster_size = 100 , average score: 0.010478759361255884
Current dr: svd_20.pkl
HDBSCAN when min_cluster_size = 100 , average score: 0.008450507111460306
Current dr: svd_200.pkl
HDBSCAN when min_cluster_size = 100 , average score: 0.006087403063703824
Current dr: nmf_5.pkl
HDBSCAN when min_cluster_size = 100 , average score: 0.09425820851516531
Current dr: nmf_20.pkl
HDBSCAN when min_cluster_size = 100 , average score: 0.010023000451534418
Current dr: nmf_200.pkl
HDBSCAN when min_cluster_size = 100 , average score: 0.006022948696612804
Current dr: umap_5.pkl
HDBSCAN when min_cluster_size = 100 , average score: 0.019719014605143435
Current dr: umap_20.pkl
HDBSCAN when min_cluster_size = 100 , average score: 0.024785356321964856
Current dr: umap_200.pkl
HDBSCAN when min_cluster_size = 100 , average score: 0.018821756889884196

min_cluster_size = 200 …
Current dr: none.pkl
HDBSCAN when min_cluster_size = 200 , average score: 0.023505928045753802
Current dr: svd_5.pkl
HDBSCAN when min_cluster_size = 200 , average score: 0.022856027344854608
Current dr: svd_20.pkl
```

```
HDBSCAN when min_cluster_size = 200 , average score: 0.006763272708372309
Current dr: svd_200.pkl
HDBSCAN when min_cluster_size = 200 , average score: 0.007532258670429807
Current dr: nmf_5.pkl
HDBSCAN when min_cluster_size = 200 , average score: 0.04213099629994795
Current dr: nmf_20.pkl
HDBSCAN when min_cluster_size = 200 , average score: 0.007863958796589979
Current dr: nmf_200.pkl
HDBSCAN when min_cluster_size = 200 , average score: 0.005149949351729706
Current dr: umap_5.pkl
HDBSCAN when min_cluster_size = 200 , average score: 0.01843515443795917
Current dr: umap_20.pkl
HDBSCAN when min_cluster_size = 200 , average score: 0.024252152756405333
Current dr: umap_200.pkl
HDBSCAN when min_cluster_size = 200 , average score: 0.017886948206185305
```

Ans:

From the result we obtained, the combination of **UMAP using 'cosine' matrix and n_components setting to 5 plus K-Means using k = 20** works best together for 20-class text data.

It might because of that the categories it need to predict is also 20. On the other hand, UMAP captures the structure quickly and efficiently comparing to other methods. Combining these two, they performs better than other combinations.

0.19 Question 18

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

Ans:

From the result we obtained, we can see that any clustering method using UMAP as the Dimensionality Reduction method performs much better than other methods.

From this perspective, we try to adjust the parameters of UMAP to see if we can obtain a better result.

[ ]:
```python
rs18 = [30, 50, 100]
for i in range(len(rs18)):
  print("\ntry r =", rs18[i], "...")

  # UMAP
  print("using UMAP ...")
  umap_tmp = umap.UMAP(n_components=rs18[i], metric='cosine')
  X_umap = umap_tmp.fit_transform(X_tfidf)
  pkl_save(path, X_umap, 'umap_' + str(rs18[i]) + '.pkl')
```

```
try r = 30 …
```

```
using UMAP …

try r = 50 …
using UMAP …

try r = 100 …
using UMAP …
```

```
[ ]: umaps = ['umap_5.pkl', 'umap_20.pkl', 'umap_30.pkl', 'umap_50.pkl', 'umap_100.
      ↪pkl', 'umap_200.pkl']
```

```
[ ]: ks = [10, 20, 30, 50, 100]
     km_scores = [ [0] * len(umaps) for i in range(len(ks))]
     for i in range(len(ks)):
       print("\nk =", ks[i], "...")
       kmeans_tmp = KMeans(n_clusters=ks[i], init='k-means++', max_iter=2000,␣
      ↪n_init=50, random_state=0)
       for j in range(len(umaps)):
         print("Current UMAP:", umaps[j])
         # reading Dimensionality Reduction data
         X_dr = pkl_read(path, umaps[j])
         kmeans_fit = kmeans_tmp.fit(X_dr)
         # To increase efficiency, we calculate the average score of each␣
      ↪combination instead.
         tmp_s = 0
         tmp_s += homogeneity_score(y_data, kmeans_fit.labels_)
         tmp_s += completeness_score(y_data, kmeans_fit.labels_)
         tmp_s += v_measure_score(y_data, kmeans_fit.labels_)
         tmp_s += adjusted_rand_score(y_data, kmeans_fit.labels_)
         tmp_s += adjusted_mutual_info_score(y_data, kmeans_fit.labels_)
         tmp_s /= 5
         km_scores[i][j] = tmp_s
         print("Kmeans when k =", ks[i], ", average score:", tmp_s)
```

```
k = 10 …
Current UMAP: umap_5.pkl
Kmeans when k = 10 , average score: 0.4983759015992576
Current UMAP: umap_20.pkl
Kmeans when k = 10 , average score: 0.5023552400963419
Current UMAP: umap_30.pkl
Kmeans when k = 10 , average score: 0.5025394557537396
Current UMAP: umap_50.pkl
Kmeans when k = 10 , average score: 0.5028938404781135
Current UMAP: umap_100.pkl
Kmeans when k = 10 , average score: 0.5012430382777919
Current UMAP: umap_200.pkl
Kmeans when k = 10 , average score: 0.5034860305394877
```

```
k = 20 …
Current UMAP: umap_5.pkl
Kmeans when k = 20 , average score: 0.5527447708674883
Current UMAP: umap_20.pkl
Kmeans when k = 20 , average score: 0.5485179865082734
Current UMAP: umap_30.pkl
Kmeans when k = 20 , average score: 0.5594055845292851
Current UMAP: umap_50.pkl
Kmeans when k = 20 , average score: 0.5648750854798654
Current UMAP: umap_100.pkl
Kmeans when k = 20 , average score: 0.5660340490887646
Current UMAP: umap_200.pkl
Kmeans when k = 20 , average score: 0.556148359746299

k = 30 …
Current UMAP: umap_5.pkl
Kmeans when k = 30 , average score: 0.5570951177263229
Current UMAP: umap_20.pkl
Kmeans when k = 30 , average score: 0.5551454175119898
Current UMAP: umap_30.pkl
Kmeans when k = 30 , average score: 0.5536573120116477
Current UMAP: umap_50.pkl
Kmeans when k = 30 , average score: 0.5495626680424508
Current UMAP: umap_100.pkl
Kmeans when k = 30 , average score: 0.5503191097974127
Current UMAP: umap_200.pkl
Kmeans when k = 30 , average score: 0.559685131958972

k = 50 …
Current UMAP: umap_5.pkl
Kmeans when k = 50 , average score: 0.5225005310003482
Current UMAP: umap_20.pkl
Kmeans when k = 50 , average score: 0.5311132275419802
Current UMAP: umap_30.pkl
Kmeans when k = 50 , average score: 0.5165038124127054
Current UMAP: umap_50.pkl
Kmeans when k = 50 , average score: 0.520842165533297
Current UMAP: umap_100.pkl
Kmeans when k = 50 , average score: 0.51853794581093
Current UMAP: umap_200.pkl
Kmeans when k = 50 , average score: 0.5255663945896251

k = 100 …
Current UMAP: umap_5.pkl
Kmeans when k = 100 , average score: 0.4731704103593664
Current UMAP: umap_20.pkl
Kmeans when k = 100 , average score: 0.480518314730074
```

```
Current UMAP: umap_30.pkl
Kmeans when k = 100 , average score: 0.4752450376025603
Current UMAP: umap_50.pkl
Kmeans when k = 100 , average score: 0.48481560206411894
Current UMAP: umap_100.pkl
Kmeans when k = 100 , average score: 0.47507179317347054
Current UMAP: umap_200.pkl
Kmeans when k = 100 , average score: 0.478029997188872
```

From above result, we find that the performace of UMAP_100 is a little bit better than UMAP_50. And $k = 20$ is the best setting for K-Means. We will try to adjust some parameters for these two settings to get a better result.

```
[ ]: n_neighs = [20, 30, 50, 100]
     min_dists = [0.2, 0.3, 0.5, 0.87]
     kmeans_tmp = KMeans(n_clusters=20, init='k-means++', max_iter=2000, n_init=50,␣
      ↪random_state=0)
     for i in range(len(n_neighs)):
         print("\nn_neighbors =", n_neighs[i], "...")
         for j in range(len(min_dists)):
             print("Current min_dist:", min_dists[j])
             umap_tmp = umap.UMAP(n_components=100, n_neighbors=n_neighs[i],␣
      ↪min_dist=min_dists[j], metric='cosine')
             X_umap = umap_tmp.fit_transform(X_tfidf)
             kmeans_fit = kmeans_tmp.fit(X_umap)
             tmp_s = 0
             tmp_s += homogeneity_score(y_data, kmeans_fit.labels_)
             tmp_s += completeness_score(y_data, kmeans_fit.labels_)
             tmp_s += v_measure_score(y_data, kmeans_fit.labels_)
             tmp_s += adjusted_rand_score(y_data, kmeans_fit.labels_)
             tmp_s += adjusted_mutual_info_score(y_data, kmeans_fit.labels_)
             tmp_s /= 5
             km_scores[i][j] = tmp_s
             print("Kmeans when k = 20", ", UMAP when n_neighbors =", n_neighs[i],␣
      ↪", min_dist =", min_dists[j], ", average score:", tmp_s)
```

```
n_neighbors = 20 …
Current min_dist: 0.2
Kmeans when k = 20 , UMAP when n_neighbors = 20 , min_dist = 0.2 , average
score: 0.580787761743627
Current min_dist: 0.3
Kmeans when k = 20 , UMAP when n_neighbors = 20 , min_dist = 0.3 , average
score: 0.5724508070219805
Current min_dist: 0.5
Kmeans when k = 20 , UMAP when n_neighbors = 20 , min_dist = 0.5 , average
score: 0.5744771322175886
Current min_dist: 0.87
```

```
Kmeans when k = 20 , UMAP when n_neighbors = 20 , min_dist = 0.87 , average
score: 0.5945454230665579

n_neighbors = 30 …
Current min_dist: 0.2
Kmeans when k = 20 , UMAP when n_neighbors = 30 , min_dist = 0.2 , average
score: 0.5737626120600483
Current min_dist: 0.3
Kmeans when k = 20 , UMAP when n_neighbors = 30 , min_dist = 0.3 , average
score: 0.5736243636591776
Current min_dist: 0.5
Kmeans when k = 20 , UMAP when n_neighbors = 30 , min_dist = 0.5 , average
score: 0.5778039370853855
Current min_dist: 0.87
Kmeans when k = 20 , UMAP when n_neighbors = 30 , min_dist = 0.87 , average
score: 0.595183451482399

n_neighbors = 50 …
Current min_dist: 0.2
Kmeans when k = 20 , UMAP when n_neighbors = 50 , min_dist = 0.2 , average
score: 0.576943355602227
Current min_dist: 0.3
Kmeans when k = 20 , UMAP when n_neighbors = 50 , min_dist = 0.3 , average
score: 0.5755555116333844
Current min_dist: 0.5
Kmeans when k = 20 , UMAP when n_neighbors = 50 , min_dist = 0.5 , average
score: 0.5911223311681624
Current min_dist: 0.87
Kmeans when k = 20 , UMAP when n_neighbors = 50 , min_dist = 0.87 , average
score: 0.5970547503057227

n_neighbors = 100 …
Current min_dist: 0.2
Kmeans when k = 20 , UMAP when n_neighbors = 100 , min_dist = 0.2 , average
score: 0.5825166086844924
Current min_dist: 0.3
Kmeans when k = 20 , UMAP when n_neighbors = 100 , min_dist = 0.3 , average
score: 0.5785838926897937
Current min_dist: 0.5
Kmeans when k = 20 , UMAP when n_neighbors = 100 , min_dist = 0.5 , average
score: 0.5798158369565941
Current min_dist: 0.87
Kmeans when k = 20 , UMAP when n_neighbors = 100 , min_dist = 0.87 , average
score: 0.5967891400756058
```

After the experiment, we found that the performance usually gets better when we increase the **min_dist** parameter. Which makes sense because as the minimum distance between each data point increase, we should separate the cluster more easily and get a higher Homogeneity and

Completeness score.

In conclusion, we found a better clustering performance by using **UMAP with $ n__components = 100, n__neighbors = 50$, and** $min\_dist = 0.87$**. Combining it with K-means with k = 20**, we get a average score which is approximately 0.597.

0.20 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?

Ans:

We expect that the model is trained on a large and general enough dataset. Even though it's a dataset with perhaps totally different classes as targets, we can still get advantage from its learned feature maps to repurpose it.

To be specifically, there are always multiple layers inside our network model. We can imagine that the final layer of our network is the one that learn to identify classes specific to our project that need training. The initial layers or the middle layers are used to detect slant lines no matter what you want to classify. Therefore, we can always reuse the neural network instead of retraining them every time we create a new one.

Finally, after getting the pretrained model, we can use fine-tuning to make the model more relevant to the custom dataset we want to use. By doing this, we can expect that the features derived from previous network will have discriminative power for a custom dataset.

0.21 Question 20

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

Ans:

It first extracts feature layers and pooling layer from the VGG-16. After that, it constructs a flatten layer using torch, and extracts the first part of fully-connected layer from VGG-16.

Similar to the concept we mentioned in Q19, it extracts the basic layers from the pretrained model VGG-16. By doing this, it extracts the basic discriminative pwoer we have in the pretrained model. We can use it in further application such as fine-tuning on the custom dataset we want to test.

0.22 Question 21

[4]:
```
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_photos.tgz'
        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',␣
↪pretrained=True)

            # 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 VGG16
            self.fc = vgg.classifier[0]

        def forward(self, x):
            # It will take the input 'x' until it returns the feature vector␣
↪called 'out'
            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([transforms.
↪Resize(224),
                                                                 transforms.
↪CenterCrop(224),
                                                                 transforms.
↪ToTensor(),
                                                                 transforms.
↪Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])]))
```

```
    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()).cpu()])
            y_all = np.concatenate([y_all, y])
    np.savez(filename, f_all=f_all, y_all=y_all)
```

```
Downloading: "https://github.com/pytorch/vision/zipball/v0.10.0" to
/root/.cache/torch/hub/v0.10.0.zip
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior is
equivalent to passing `weights=VGG16_Weights.IMAGENET1K_V1`. You can also use
`weights=VGG16_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to
/root/.cache/torch/hub/checkpoints/vgg16-397923af.pth

  0%|          | 0.00/528M [00:00<?, ?B/s]


100%|     | 58/58 [00:44<00:00,  1.32it/s]
```

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?

Ans:

There are 224 x 224 pixels in the original images.

There are totally 3670 features being extracted by the VGG network per image.

```
[ ]: for x, y in tqdm(dataloader):
       print("\n")
       print(x.size())
       print(y.size())
       break

     print("\nfeature vector:", f_all.shape, y_all.shape)
     num_features = f_all.shape[1]
```

```
  0%|          | 0/58 [00:00<?, ?it/s]
```

```
torch.Size([64, 3, 224, 224])
torch.Size([64])

feature vector: (3670, 4096) (3670,)
```

0.23 Question 22

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

Ans:

From Q10, we found the shape of TFIDX dataset: (18846, 45365)

The extracted feature vectors we found: (3670, 4096)

Furthermore, we can see that the numbers inside TF-IDF features are much closer to zero comparing to the extracted features. Therefore, the extracted features are more dense.

```
[ ]: print(f_all.shape)
     print(f_all)
     print(X_tfidf.shape)
     print(X_tfidf)
```

```
(3670, 4096)
[[-0.45783606  2.75369859  0.84992045 … -2.20686865 -0.35974157
   1.0510205 ]
 [-0.01870249 -2.30375719 -0.63398772 … -4.68432951  1.58356237
   2.4888947 ]
 [-2.29198527 -1.39408898  0.04004309 … -1.57638204 -0.39932269
  -0.77769351]
 …
 [-1.71726406 -3.87869978 -4.15782833 … -0.03698874  0.24791169
   7.03038645]
 [-1.67345393 -0.80352944 -2.1430831  …  2.40008855 -0.81528664
   1.85219288]
 [ 1.27578044 -0.57890528 -0.29694718 … -3.28786254 -1.68270898
   0.94054085]]
(18846, 45365)
  (0, 45082)    0.10834863946498498
  (0, 44381)    0.1584590937040802
  (0, 42541)    0.058066446887881366
  (0, 41490)    0.08051514880421468
  (0, 41316)    0.13329293384982419
  (0, 38204)    0.16925886266575615
  (0, 37492)    0.09850240968265686
  (0, 36007)    0.09826054436567758
  (0, 33328)    0.47942224155915814
  (0, 33262)    0.18420734272692216
```

```
(0, 31695)    0.3963165278889442
(0, 31683)    0.19522385821708108
(0, 29617)    0.11875524902913023
(0, 29022)    0.0622296687474105
(0, 27706)    0.11540121691934306
(0, 25516)    0.08157968927565735
(0, 24419)    0.11312900170774419
(0, 24371)    0.1181419537275807
(0, 22243)    0.2851039251453595
(0, 18123)    0.16532689809474904
(0, 17976)    0.1311636085943422
(0, 17787)    0.1240128023295206
(0, 15208)    0.13079807275779112
(0, 14150)    0.20396815849564107
(0, 11697)    0.09878805754148298
:      :
(18845, 18514)        0.05860474143465704
(18845, 17963)        0.05712310612173224
(18845, 17506)        0.0722870974794078
(18845, 17232)        0.05272356612438965
(18845, 16972)        0.06351235739684634
(18845, 16255)        0.09477122354699491
(18845, 14717)        0.1514142823050571
(18845, 14608)        0.16023615155667814
(18845, 13592)        0.04186755908990516
(18845, 13576)        0.08863063242402147
(18845, 13168)        0.048522897301106066
(18845, 12635)        0.060843936981812494
(18845, 12514)        0.13454853488766122
(18845, 11547)        0.20054431862465735
(18845, 9782) 0.08533659547734827
(18845, 9006) 0.15577532500269423
(18845, 8379) 0.0701110751954832
(18845, 7355) 0.08376382234918138
(18845, 7162) 0.07120001286865578
(18845, 6709) 0.06481462637628999
(18845, 5388) 0.06852361213473072
(18845, 5385) 0.050512997363727795
(18845, 5154) 0.09822577469614639
(18845, 2468) 0.0792566940653086
(18845, 826)  0.04868629510260459
```

0.24 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.

Ans:

From our result, we can see that there are 5 categories (clusterings) and they have many overlapping spots when we project it onto 2 dimensions.

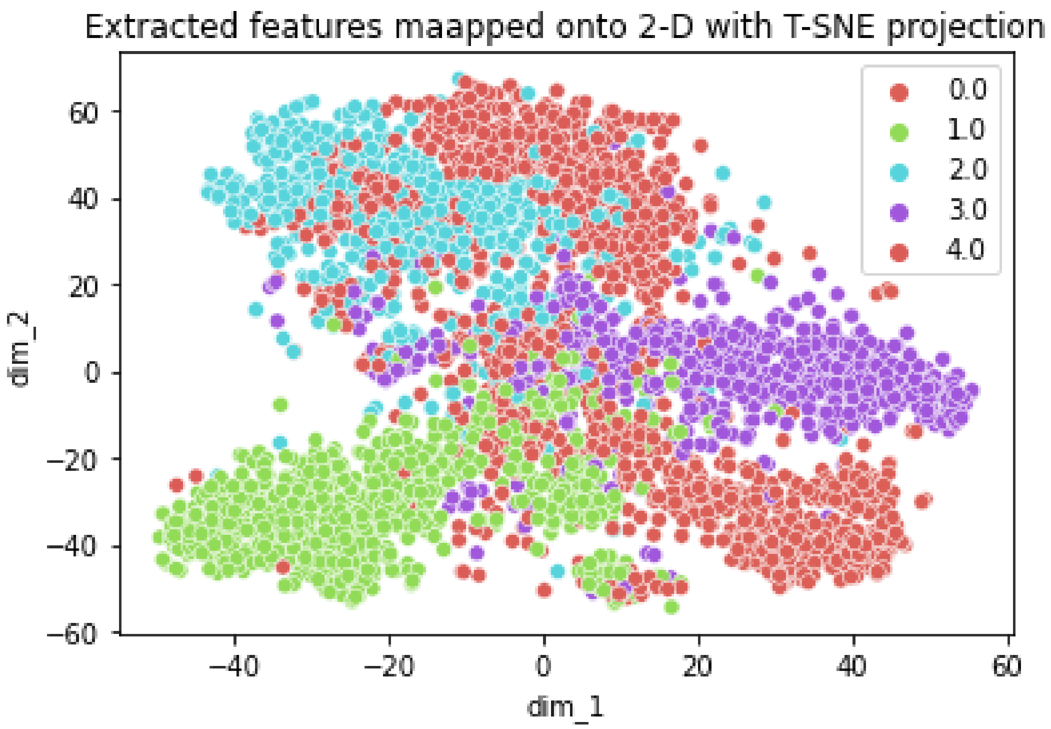
It indicates that we might face some problems when we try to use high dimensional clustering method on this dataset. The distance between each data point is really close according to our projection result.

```
[ ]: tsne = TSNE(n_components=2, verbose=1, random_state=0)
     z = tsne.fit_transform(f_all)
     df = pd.DataFrame()
     df["y"] = y_all
     df["dim_1"] = z[:,0]
     df["dim_2"] = z[:,1]
     sns.scatterplot(x="dim_1", y="dim_2", hue=df.y.tolist(),
                     palette=sns.color_palette("hls", as_cmap = True),
                     data=df).set(title="Extracted features maapped onto 2-D with␣
      ↪T-SNE projection")
```

```
/usr/local/lib/python3.8/dist-packages/sklearn/manifold/_t_sne.py:780:
FutureWarning: The default initialization in TSNE will change from 'random' to
'pca' in 1.2.
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/manifold/_t_sne.py:790:
FutureWarning: The default learning rate in TSNE will change from 200.0 to
'auto' in 1.2.
  warnings.warn(
```

```
[t-SNE] Computing 91 nearest neighbors…
[t-SNE] Indexed 3670 samples in 0.018s…
[t-SNE] Computed neighbors for 3670 samples in 3.112s…
[t-SNE] Computed conditional probabilities for sample 1000 / 3670
[t-SNE] Computed conditional probabilities for sample 2000 / 3670
[t-SNE] Computed conditional probabilities for sample 3000 / 3670
[t-SNE] Computed conditional probabilities for sample 3670 / 3670
[t-SNE] Mean sigma: 36.158544
[t-SNE] KL divergence after 250 iterations with early exaggeration: 78.663742
[t-SNE] KL divergence after 1000 iterations: 1.762485
```

```
[ ]: [Text(0.5, 1.0, 'Extracted features maapped onto 2-D with T-SNE projection')]
```



0.25 Question 24

0.25.1 Autoencoder

[5]:
```
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):
        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.Linear(120,␣
 ↪self.n_components))

    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,␣
↪weight_decay=1e-5)

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

        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()
```

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.

Module	Alternatives	Hyperparameters
Dimensionality Reduction	None	N/A
Dimensionality Reduction	SVD	r = 50
Dimensionality Reduction	UMAP	n_components = 50
Dimensionality Reduction	Autoencoder	num_features = 50
Clustering	K-Means	k = 5
Clustering	Agglomerative Clustering	n_clusters = 5
Clustering	HDBSCAN	min_cluster_size & min_samples

Ans:

The best rand score we get is approximately 0.467. Which is derived from the combination of UMAP [n_components = 50] + K-Means [k = 5].

For HDBSCAN, we set min_cluster_size = [50, 100, 200], min_samples = [20, 50, 100, 200] to proceed a grid search. However, the best rand score we can get is approximately only 0.094 when combining with UMAP [n_components = 50]. Which is still much worse than other combination.

[ ]:
```python
# None
pkl_save(path, f_all, 'f_none.pkl')

r24 = 50
print("\ntry r =", r24, "...")
# SVD
print("using SVD ...")
svd_tmp = TruncatedSVD(n_components=r24, random_state=0)
X_svd = svd_tmp.fit_transform(f_all)
pkl_save(path, X_svd, 'f_svd_' + str(r24) + '.pkl')

# UMAP
print("using UMAP ...")
umap_tmp = umap.UMAP(n_components=r24, metric='cosine')
X_umap = umap_tmp.fit_transform(f_all)
pkl_save(path, X_umap, 'f_umap_' + str(r24) + '.pkl')

# Autoencoder
print("using Autoencoder ...")
X_aec = Autoencoder(r24).fit_transform(f_all)
pkl_save(path, X_aec, 'f_aec_' + str(r24) + '.pkl')
```

```
try r = 50 …
using SVD …
using UMAP …
using Autoencoder …
```

```
100%|      | 100/100 [00:24<00:00,  4.08it/s]
```

```
[ ]: DRs = ['f_none.pkl', 'f_svd_50.pkl', 'f_umap_50.pkl', 'f_aec_50.pkl']
```

```
[ ]: ks = [5]
     km_scores24 = [ [0] * len(DRs) for i in range(len(ks))]
     for i in range(len(ks)):
       print("\nk =", ks[i], "...")
       kmeans_tmp = KMeans(n_clusters=ks[i], init='k-means++', max_iter=2000,␣
     ↪n_init=50, random_state=0)
       for j in range(len(DRs)):
         print("Current dr:", DRs[j])
         # reading Dimensionality Reduction data
         X_dr = pkl_read(path, DRs[j])
         kmeans_fit = kmeans_tmp.fit(X_dr)
         # As the question mentioned, we use rand score to determine which␣
     ↪combination performs best.
         tmp_s = adjusted_rand_score(y_all, kmeans_fit.labels_)
         km_scores24[i][j] = tmp_s
         print("Kmeans when k =", ks[i], ", rand score:", tmp_s)
```

```
k = 5 …
Current dr: f_none.pkl
Kmeans when k = 5 , rand score: 0.19216442375229253
Current dr: f_svd_50.pkl
Kmeans when k = 5 , rand score: 0.18805072438502163
Current dr: f_umap_50.pkl
Kmeans when k = 5 , rand score: 0.4665092104522564
Current dr: f_aec_50.pkl
Kmeans when k = 5 , rand score: 0.20346920248488434
```

```
[ ]: agg_n = 5
     agg_scores24 = [ [0] * len(DRs)]
     for j in range(len(DRs)):
       print("Current dr:", DRs[j])
       # reading Dimensionality Reduction data
       X_dr = pkl_read(path, DRs[j])
       agg_tmp = AgglomerativeClustering(n_clusters=agg_n, linkage='ward').fit(X_dr)
       # As the question mentioned, we use rand score to determine which combination␣
     ↪performs best.
       tmp_s = adjusted_rand_score(y_all, agg_tmp.labels_)
       agg_scores24[0][j] = tmp_s
       print("Agglomerative Clustering when n_clusters =", agg_n, ", rand score:",␣
     ↪tmp_s)
```

```
Current dr: f_none.pkl
Agglomerative Clustering when n_clusters = 5 , rand score: 0.18855278251971858
Current dr: f_svd_50.pkl
```

```
Agglomerative Clustering when n_clusters = 5 , rand score: 0.19998277233193223
Current dr: f_umap_50.pkl
Agglomerative Clustering when n_clusters = 5 , rand score: 0.45265411323193355
Current dr: f_aec_50.pkl
Agglomerative Clustering when n_clusters = 5 , rand score: 0.28906349861270764
```

```
[ ]: mc_ss = [50, 100, 200]
     min_ss = [20, 50, 100, 200]
     # hdbs_scores24 = [ [0] * len(DRs) for i in range(2)]
     for i in range(len(mc_ss)):
       print("\nmin_cluster_size =", mc_ss[i], "...")
       for k in range(len(min_ss)):
         print("\nmin_samples =", min_ss[k], "...")
         for j in range(len(DRs)):
           print("Current dr:", DRs[j])
           # reading Dimensionality Reduction data
           X_dr = pkl_read(path, DRs[j])
           y_pred_hdbs = hdbscan.HDBSCAN(min_cluster_size=mc_ss[i],␣
      ↪min_samples=min_ss[k], allow_single_cluster=True).fit_predict(X_dr)
           # As the question mentioned, we use rand score to determine which␣
      ↪combination performs best.
           tmp_s = adjusted_rand_score(y_all, y_pred_hdbs)
           # hdbs_scores24[i][j] = tmp_s
           print("HDBSCAN when min_cluster_size =", mc_ss[i], ", min_samples =",␣
      ↪min_ss[k], ", rand score:", tmp_s)
```

```
min_cluster_size = 50 …

min_samples = 20 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 20 , rand score:
-0.0010448965393208944
Current dr: f_svd_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 20 , rand score:
-0.000779608175723694
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 20 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 20 , rand score:
-0.0004412418123981401

min_samples = 50 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 50 , rand score:
-0.0007876010819439827
Current dr: f_svd_50.pkl
```

```
HDBSCAN when min_cluster_size = 50 , min_samples = 50 , rand score:
-0.0006014805513858321
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 50 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 50 , rand score:
-0.0004439061144715697

min_samples = 100 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 100 , rand score:
-0.0009604001021349855
Current dr: f_svd_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 100 , rand score:
-0.0006916862073005186
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 100 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 100 , rand score:
-0.0005748375306515365

min_samples = 200 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 200 , rand score:
-0.0010403291643378723
Current dr: f_svd_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 200 , rand score:
-0.0006349746345946608
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 200 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 50 , min_samples = 200 , rand score:
-0.0007579131445543391

min_cluster_size = 100 …

min_samples = 20 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 20 , rand score:
-0.0014971865521274726
Current dr: f_svd_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 20 , rand score:
-0.0011620618099798916
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 20 , rand score:
```

```
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 20 , rand score:
-0.0011309957852153418

min_samples = 50 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 50 , rand score:
-0.0012971990177056832
Current dr: f_svd_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 50 , rand score:
-0.0010401276627790335
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 50 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 50 , rand score:
-0.0010012951318233462

min_samples = 100 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 100 , rand score:
-0.0011500237253836286
Current dr: f_svd_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 100 , rand score:
-0.0010273129275636568
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 100 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 100 , rand score:
-0.0007411171744202416

min_samples = 200 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 200 , rand score:
-0.0011550719544078678
Current dr: f_svd_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 200 , rand score:
-0.0011663333883850172
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 200 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 100 , min_samples = 200 , rand score:
-0.0007966476936868744

min_cluster_size = 200 …
```

```
min_samples = 20 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 20 , rand score:
-0.0017188731500706628
Current dr: f_svd_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 20 , rand score:
-0.0019216794456046568
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 20 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 20 , rand score:
-0.001526570766257553

min_samples = 50 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 50 , rand score:
-0.001625953931001303
Current dr: f_svd_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 50 , rand score:
-0.001632013880071044
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 50 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 50 , rand score:
-0.0015944421958386506

min_samples = 100 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 100 , rand score:
-0.0016926133707684525
Current dr: f_svd_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 100 , rand score:
-0.0015988861584897939
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 100 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 100 , rand score:
-0.001404159795048787 7

min_samples = 200 …
Current dr: f_none.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 200 , rand score:
-0.0015665664301178425
Current dr: f_svd_50.pkl
```

```
HDBSCAN when min_cluster_size = 200 , min_samples = 200 , rand score:
-0.0017047332689079343
Current dr: f_umap_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 200 , rand score:
0.09411014415380671
Current dr: f_aec_50.pkl
HDBSCAN when min_cluster_size = 200 , min_samples = 200 , rand score:
-0.0014094117509092297
```

0.26 Question 25

0.26.1 MLP classifier

[61]:
```
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,␣
↪weight_decay=1e-5)

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

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

    def eval(self, X_test, y_test):
        X = torch.tensor(X_test, dtype=torch.float32, device='cuda')
        y = torch.tensor(y_test, dtype=torch.int64, device='cuda')
        dataset = TensorDataset(X, y)
        dataloader = DataLoader(dataset, batch_size=128, shuffle = False)
        correct_pred, num_examples = 0, 0
        for (X_, y_) in dataloader:
            X_ = X_.cuda()
            y_ = y_.cuda()
            logits = self(X_)
            predicted_labels = torch.argmax(logits, 1)
            num_examples += y_.size(0)
            correct_pred += (predicted_labels == y_).sum()
            # print("predicted_labels", predicted_labels)
            # print("y", y_)
        acc = correct_pred.float() / num_examples * 100
        return acc
```

Report the test accuracy of the MLP classifier on the original VGG features.

[62]:
```
X_train, X_test, y_train, y_test = train_test_split(f_all, y_all, test_size=0.
 ↪2, random_state=0)
print(X_train.shape)
```

```
(2936, 4096)
```

[65]:
```
model = MLP(X_train.shape[1])
model.train(X_train, y_train)
acc = model.eval(X_test, y_test)
print("\nMLP classifier on the original VGG features")
print("Test accuracy: %.2f" % acc, "%")
```

```
100%|      | 100/100 [00:08<00:00, 11.75it/s]


MLP classifier on the original VGG features
Test accuracy: 90.46 %
```

Report the same when using the reduced-dimension features (you have freedom in choosing the dimensionality reduction algorithm and its parameters).

[73]:
```
umap_50 = umap.UMAP(n_components=50, metric='cosine')
X_umap = umap_50.fit_transform(f_all)
# X_umap = pkl_read(path, 'f_umap_50.pkl')
X_train, X_test, y_train, y_test = train_test_split(X_umap, y_all, test_size=0.
 ↪2, random_state=0)
print(X_train.shape)
```

```
(2936, 50)
```

[74]:
```
model = MLP(X_train.shape[1])
model.train(X_train, y_train)
acc = model.eval(X_test, y_test)
print("\nMLP classifier on the reduced-dimension features")
print("Test accuracy: %.2f" % acc, "%")
```

```
100%|      | 100/100 [00:12<00:00,  8.21it/s]


MLP classifier on the reduced-dimension features
Test accuracy: 86.38 %
```

Does the performance of the model suffer with the reduced-dimension representations? Is it significant?

Ans:

Yes, the performance of the model gets worse when we use the reduced-dimension representation. It is not significant, it only decreases the test accuracy by 4% approximately.

Does the success in classification make sense in the context of the clustering results obtained for the same features in Question 24.

Ans:

Yes, it makes sense. The performance of the model should not be affected a lot if it's already a well-clustering dataset. Even though we reduce the dimensions of the features, the model can still get the structure of the data and performs well eventually.