Part 2: Limitation(s) of sklearn’s non-negative matrix factorization library

Copying the code from the previous assignment, we setup our data here. We then create a rating matrix from the test data as we are trying to input the missing ratings from there.

For reference on creating predictions from matrix factorization, this page was viewed

- https://www.kaggle.com/code/eminecerit/movie-recommender-matrix-factorization-based

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from itertools import permutations

from sklearn.cluster import KMeans
from sklearn.decomposition import TruncatedSVD
from sklearn.decomposition import NMF
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
```

In [2]:

```
from scipy.sparse import coo_matrix
from collections import namedtuple

MV_users = pd.read_csv('data/users.csv')
MV_movies = pd.read_csv('data/movies.csv')
train = pd.read_csv('data/train.csv')
test = pd.read_csv('data/test.csv')

Data = namedtuple('Data', ['users','movies','train','test'])
data = Data(MV_users, MV_movies, train, test)

allusers = list(data.users['uID'])
allmovies = list(data.movies['mID'])
mid2idx = dict(zip(data.movies.mID,list(range(len(data.movies)))))
uid2idx = dict(zip(data.users.uID,list(range(len(data.users)))))

ind_movie = [mid2idx[x] for x in data.test.mID]
ind_user = [uid2idx[x] for x in data.test.uID]
rating_test = list(data.test.rating)

rating_matrix = np.array(coo_matrix((rating_test, (ind_user, ind_movie)), shape=(len(allu
sers), len(allmovies))).toarray())
```

First we setup our non-negative matrix factorization model / NMF to commit matrix factorization onto our ratings matrix. Then we take the estimations of the missing data from when inverse transform the factorized matrix.

In [3]:

```
%%capture

nmf = NMF(n_components=5, random_state=42)
nmf_m = nmf.fit_transform(rating_matrix)
rating_matrix_p = nmf.inverse_transform(nmf_m)
```

In [4]:

```
rating_matrix_p
```

Out[4]:

```
array([[0.27898817, 0.04264456, 0.05027784, ..., 0.00735833, 0.00564499,
         0.0289247 ],
       [0.20343133, 0.07627137, 0.02264149, ..., 0.0188216 , 0.01288191,
         0.0712217 ],
       [0.4194923 , 0.08842252, 0.0532148 , ..., 0.00689331, 0.        ,
         0.00467821],
       ...,
       [0.08798879, 0.01713941, 0.02194985, ..., 0.00099939, 0.00080513,
         0.00558201],
       [0.20515588, 0.04216864, 0.05635916, ..., 0.01096005, 0.01031748,
         0.05347384],
       [0.66979071, 0.07099614, 0.01668803, ..., 0.06356899, 0.04919354,
         0.22332462]])
```

From our reconstructed matrix, we then find all of the missing values (denoted as 0) in our original matrix, and then input the equivalent value from the reconstructed matrix. As seen above, it seems that the value of the predictions are not exactly the same as our 1 to 5 rating, and thus seem like they are scaled to be based around the middle of the range of possible values. To combat this we adjust the prediction in order to cause it to be similar to the existing values. We also keep track of the existing values in the testing set in order to compare our predictions with real labels.

In [5]:

```
preds = rating_matrix.copy().astype(float)

rmse_actual = np.array([]).astype(float)
rmse_pred = np.array([]).astype(float)

for i in range(preds.shape[0]):
    for j in range(preds.shape[1]):
        if preds[i][j] == 0:
            preds[i][j] = rating_matrix_p[i][j] + 3
        else:
            rmse_actual = np.append(rmse_actual, preds[i][j])
            rmse_pred = np.append(rmse_pred, rating_matrix_p[i][j] + 3)

preds = np.round(preds, decimals = 2)
```

Our finished predictions.

In [6]:

```
preds
```

Out[6]:

```
array([[3.28, 3.04, 3.05, ..., 3.01, 3.01, 3.03],
       [3.2 , 3.08, 3.02, ..., 3.02, 3.01, 3.07],
       [3.42, 3.09, 3.05, ..., 3.01, 3.  , 3.  ],
       ...,
       [3.09, 3.02, 3.02, ..., 3.  , 3.  , 3.01],
       [3.21, 3.04, 3.06, ..., 3.01, 3.01, 3.05],
       [3.67, 3.07, 3.02, ..., 3.06, 3.05, 3.22]])
```

With our recorded data, we can also find the RMSE.

In [7]:

```
mean_squared_error(rmse_actual, rmse_pred)
```

Out[7]:

```
1.2090103575214424
```

It seems that our NMF results did not do as well compared to last week's similarity based methods as our RMSE is worse than before. NMF most likely does not function well when there are enourmous amount of missing or zero values, and one way to combat this would be to reduce the sparseness of the data with different matrix types, or to preprocess the data to make the data better suit NMF.

In [ ]:

```

```