# Collaborative Filtering

# Methods & Applications

Created by: Jonathan Ibrahim

NPM: 2206103421

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM

UNIVERSITAS INDONESIA

2022

## Table of Content

# Chapter 1 - Explore the MovieLen Latest Small Dataset

Before we start, we have to make sure that we already have our dependencies, that is 'recsys' folder

[1]:
```
import os
#Check if we already have the 'recsys' folder
if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    # If not then download directly from the source
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/
 ↪master/recsys.zip
    !unzip recsys.zip
```

```
--2023-01-03 20:57:08--  https://github.com/nzhinusoftcm/review-on-
collaborative-filtering/raw/master/recsys.zip
Resolving github.com (github.com)... 140.82.114.3
Connecting to github.com (github.com)|140.82.114.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/nzhinusoftcm/review-on-
collaborative-filtering/master/recsys.zip [following]
--2023-01-03 20:57:08--  https://raw.githubusercontent.com/nzhinusoftcm/review-
on-collaborative-filtering/master/recsys.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 15312323 (15M) [application/zip]
Saving to: ‘recsys.zip’

recsys.zip          100%[===================>]  14.60M  --.-KB/s    in 0.1s

2023-01-03 20:57:09 (128 MB/s) - ‘recsys.zip’ saved [15312323/15312323]

Archive:  recsys.zip
   creating: recsys/
  inflating: recsys/datasets.py
  inflating: recsys/preprocessing.py
  inflating: recsys/utils.py
  inflating: recsys/requirements.txt
```

```
 creating: recsys/.vscode/
inflating: recsys/.vscode/settings.json
 creating: recsys/__pycache__/
inflating: recsys/__pycache__/datasets.cpython-36.pyc
inflating: recsys/__pycache__/datasets.cpython-37.pyc
inflating: recsys/__pycache__/utils.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-37.pyc
inflating: recsys/__pycache__/datasets.cpython-38.pyc
inflating: recsys/__pycache__/preprocessing.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-38.pyc
 creating: recsys/memories/
inflating: recsys/memories/ItemToItem.py
inflating: recsys/memories/UserToUser.py
 creating: recsys/memories/__pycache__/
inflating: recsys/memories/__pycache__/UserToUser.cpython-36.pyc
inflating: recsys/memories/__pycache__/UserToUser.cpython-37.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-37.pyc
inflating: recsys/memories/__pycache__/user2user.cpython-36.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-36.pyc
 creating: recsys/models/
inflating: recsys/models/SVD.py
inflating: recsys/models/MatrixFactorization.py
inflating: recsys/models/ExplainableMF.py
inflating: recsys/models/NonnegativeMF.py
 creating: recsys/models/__pycache__/
inflating: recsys/models/__pycache__/SVD.cpython-36.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-37.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-36.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-37.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-36.pyc
 creating: recsys/metrics/
inflating: recsys/metrics/EvaluationMetrics.py
 creating: recsys/img/
inflating: recsys/img/MF-and-NNMF.png
inflating: recsys/img/svd.png
inflating: recsys/img/MF.png
 creating: recsys/predictions/
 creating: recsys/predictions/item2item/
 creating: recsys/weights/
 creating: recsys/weights/item2item/
 creating: recsys/weights/item2item/ml1m/
inflating: recsys/weights/item2item/ml1m/similarities.npy
inflating: recsys/weights/item2item/ml1m/neighbors.npy
 creating: recsys/weights/item2item/ml100k/
inflating: recsys/weights/item2item/ml100k/similarities.npy
inflating: recsys/weights/item2item/ml100k/neighbors.npy
```

# 1 Requirements

Other than the 'recsys' folder, we also have to make sure that the other required libs have already been installed

```
matplotlib==3.2.2
numpy==1.18.1
pandas==1.0.5
python==3.6.10
scikit-learn==0.23.1
scipy==1.5.0
```

(If we use Google Colab, these libs are already installed and up-to-date, so we're not required to double check)

Import all of the required libs

[2]:
```
from recsys.datasets import mlLatestSmall

import matplotlib.pyplot as plt
import pandas as pd
import zipfile
import urllib.request
import sys
import os
```

Load the dataset that we're going to use

[3]:
```
ratings, movies = mlLatestSmall.load()
```

```
Download data 100.5%
Successfully downloaded ml-latest-small.zip 978202 bytes.
Unzipping the ml-latest-small.zip zip file ...
```

# 2 Data visualisation

What's in the *ratings* data? Let's take a peek

[4]:
```
ratings.head()
```

[4]:

| | userid | itemid | rating | timestamp |
|---|---|---|---|---|
| 0 | 1 | 1 | 4.0 | 964982703 |
| 1 | 1 | 3 | 4.0 | 964981247 |
| 2 | 1 | 6 | 4.0 | 964982224 |
| 3 | 1 | 47 | 5.0 | 964983815 |
| 4 | 1 | 50 | 5.0 | 964982931 |

What's in the *movies* data? Let's take a peek

[5]:
```
movies.head()
```

```
[5]:    itemid                               title  \
     0       1                    Toy Story (1995)
     1       2                      Jumanji (1995)
     2       3             Grumpier Old Men (1995)
     3       4            Waiting to Exhale (1995)
     4       5  Father of the Bride Part II (1995)

                                             genres
     0  Adventure|Animation|Children|Comedy|Fantasy
     1                   Adventure|Children|Fantasy
     2                               Comedy|Romance
     3                         Comedy|Drama|Romance
     4                                       Comedy
```

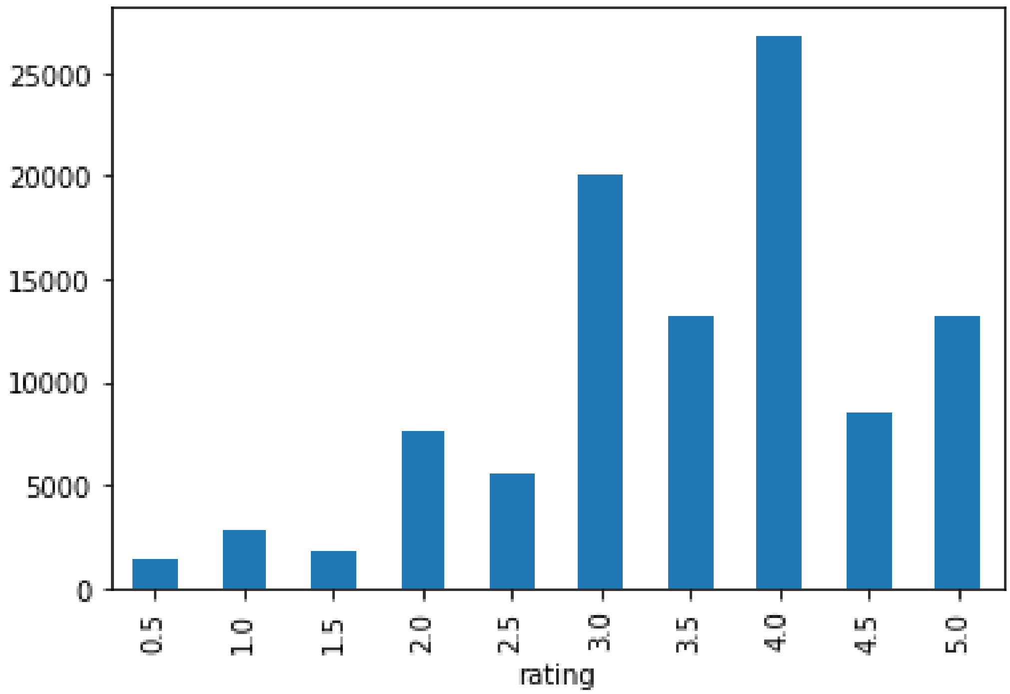
It seems that there's a connection between *ratings* and *movies* through *itemid*. Each row in *movies* data must be explaining about a certain movie, in which the *itemid* tells the ID of the movie, *title* tells the exact title with its released year, and *genres* tells the genres of the movie seperated by '|' Each row in *ratings* data must be explaining about the rating given by a certain user on a certain movie at a certain time, in which the *userid* tells the ID of the user, *itemid* tells the ID of the movie, *rating* tells the given rating, and *timestamp* tells the time that the rating is given.

## 3 Histogram of ratings

Let's see the distribution the *ratings* data

```
[6]: ratings.groupby('rating').size().plot(kind='bar')
```

```
[6]: <matplotlib.axes._subplots.AxesSubplot at 0x7f6b66a373a0>
```




Ratings range from 0.5 to 5.0, with a step of 0.5. The above histogram presents the repartition of ratings in the dataset. The two most common ratings are 4.0 and 3.0 and the less common ratings are 0.5 and 1.5

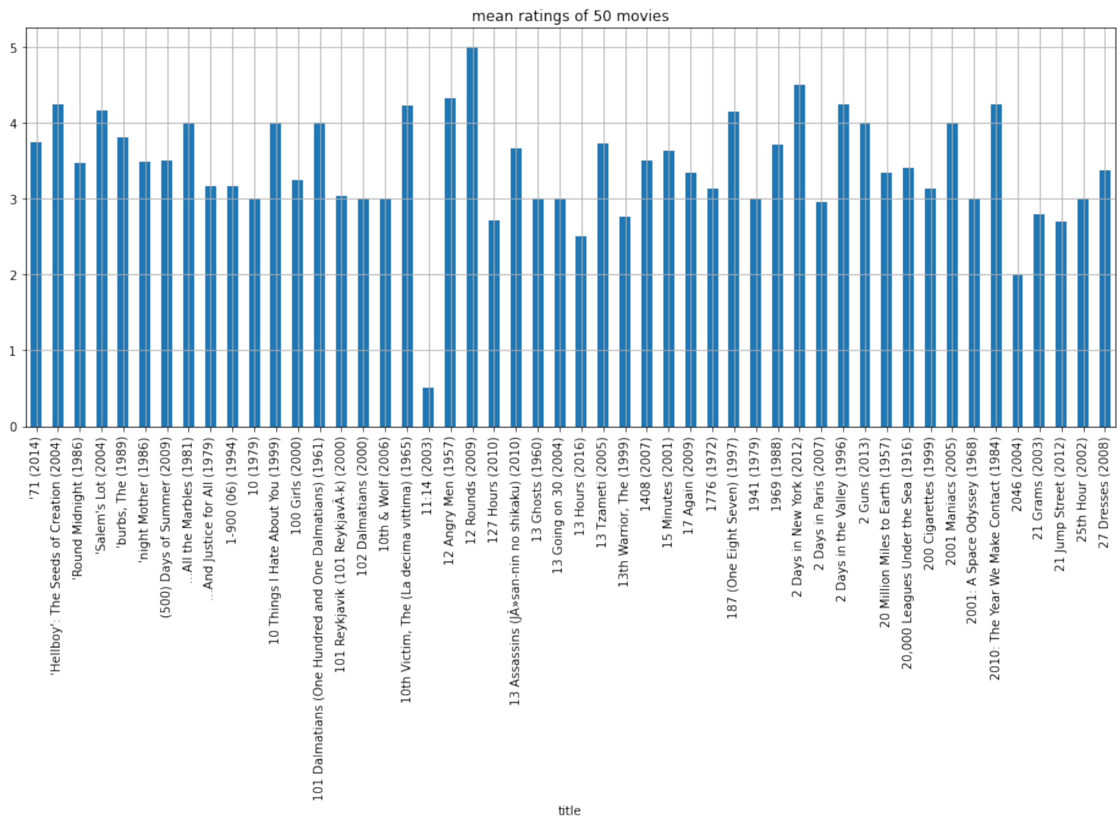
# 4 Average ratings of movies

Let's see the average ratings of each movie

For a clean visualization, we only show the average rating of 50 movies

[7]:
```
movie_means = ratings.join(movies['title'], on='itemid').groupby('title').rating.
 ↪mean()
movie_means[:50].plot(kind='bar', grid=True, figsize=(16,6), title="mean ratings␣
 ↪of 50 movies")
```

[7]: <matplotlib.axes._subplots.AxesSubplot at 0x7f6b669a40a0>



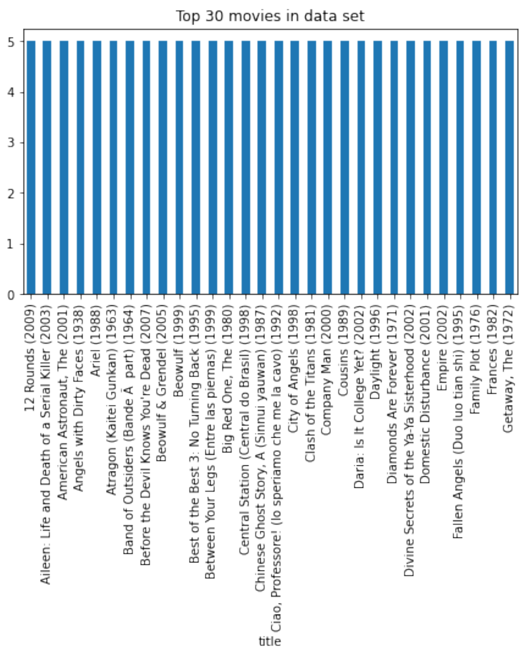

Most of the movies average rating are around the most common ratings, that is around 3.0 and 4.0. But there're still some movies with very low average ratings, for instance 11:14 (2003) with approx. 0.5 average rating

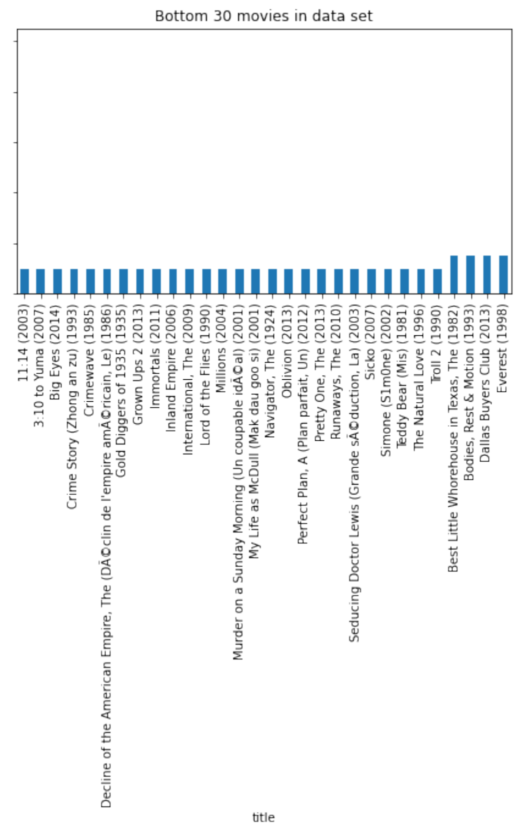
# 5 30 most rated movies vs. 30 less rated movies

Are there exist some movies with very high average rating approx. to 5.0? Are there exist some movies other than 11:14 (2003) with very low average rating approx. to 0.5? Let's take a look at the top 30 most rated movies and top 30 less rated movies

[8]:
```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(16,4), sharey=True)
movie_means.nlargest(30).plot(kind='bar', ax=ax1, title="Top 30 movies in data␣
↪set")
movie_means.nsmallest(30).plot(kind='bar', ax=ax2, title="Bottom 30 movies in␣
↪data set")
```

[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6b64b81790>
```




There exist some movies which have very high and very low average rating. The amount of movies which have very high and very low rating is inevitably not small, as there are at least 10 movies clustered in that category.

Throughout the practice in recommendation system, we're going to use this type of dataset, that is Movielens dataset.

# Chapter 2 - User-based Collaborative Filtering

Before we start, we have to make sure that we already have our dependencies, that is 'recsys' folder

[1]:
```
import os
#Check if we already have the 'recsys' folder
if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    # If not then download directly from the source
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/
 ↪master/recsys.zip
    !unzip recsys.zip
```

```
--2023-01-03 20:57:26--  https://github.com/nzhinusoftcm/review-on-
collaborative-filtering/raw/master/recsys.zip
Resolving github.com (github.com)... 140.82.113.3
Connecting to github.com (github.com)|140.82.113.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/nzhinusoftcm/review-on-
collaborative-filtering/master/recsys.zip [following]
--2023-01-03 20:57:26--  https://raw.githubusercontent.com/nzhinusoftcm/review-
on-collaborative-filtering/master/recsys.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 15312323 (15M) [application/zip]
Saving to: 'recsys.zip'

recsys.zip          100%[===================>]  14.60M  --.-KB/s    in 0.1s

2023-01-03 20:57:27 (132 MB/s) - 'recsys.zip' saved [15312323/15312323]

Archive:  recsys.zip
   creating: recsys/
  inflating: recsys/datasets.py
  inflating: recsys/preprocessing.py
  inflating: recsys/utils.py
  inflating: recsys/requirements.txt
```

```
 creating: recsys/.vscode/
inflating: recsys/.vscode/settings.json
 creating: recsys/__pycache__/
inflating: recsys/__pycache__/datasets.cpython-36.pyc
inflating: recsys/__pycache__/datasets.cpython-37.pyc
inflating: recsys/__pycache__/utils.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-37.pyc
inflating: recsys/__pycache__/datasets.cpython-38.pyc
inflating: recsys/__pycache__/preprocessing.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-38.pyc
 creating: recsys/memories/
inflating: recsys/memories/ItemToItem.py
inflating: recsys/memories/UserToUser.py
 creating: recsys/memories/__pycache__/
inflating: recsys/memories/__pycache__/UserToUser.cpython-36.pyc
inflating: recsys/memories/__pycache__/UserToUser.cpython-37.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-37.pyc
inflating: recsys/memories/__pycache__/user2user.cpython-36.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-36.pyc
 creating: recsys/models/
inflating: recsys/models/SVD.py
inflating: recsys/models/MatrixFactorization.py
inflating: recsys/models/ExplainableMF.py
inflating: recsys/models/NonnegativeMF.py
 creating: recsys/models/__pycache__/
inflating: recsys/models/__pycache__/SVD.cpython-36.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-37.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-36.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-37.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-36.pyc
 creating: recsys/metrics/
inflating: recsys/metrics/EvaluationMetrics.py
 creating: recsys/img/
inflating: recsys/img/MF-and-NNMF.png
inflating: recsys/img/svd.png
inflating: recsys/img/MF.png
 creating: recsys/predictions/
 creating: recsys/predictions/item2item/
 creating: recsys/weights/
 creating: recsys/weights/item2item/
 creating: recsys/weights/item2item/ml1m/
inflating: recsys/weights/item2item/ml1m/similarities.npy
inflating: recsys/weights/item2item/ml1m/neighbors.npy
 creating: recsys/weights/item2item/ml100k/
inflating: recsys/weights/item2item/ml100k/similarities.npy
inflating: recsys/weights/item2item/ml100k/neighbors.npy
```

# 1 Requirements

Other than the 'recsys' folder, we also have to make sure that the other required libs have already been installed

```
matplotlib==3.2.2
numpy==1.19.2
pandas==1.0.5
python==3.7
scikit-learn==0.24.1
scikit-surprise==1.1.1
scipy==1.6.2
```

(If we use Google Colab, most of these libs are already installed and up-to-date, except for *scikit-surprise* which is not pre-installed by Google Colab)

To install scikit-surprise on Google Colab, we must execute the code below

```
!pip install surprise
```

But this notebook doesn't require this library yet, so we're not going to install scikit-surprise at the moment

Import all of the required libs

[2]:
```python
from sklearn.neighbors import NearestNeighbors
from scipy.sparse import csr_matrix

from recsys.datasets import ml100k
from recsys.preprocessing import ids_encoder

import pandas as pd
import numpy as np
import zipfile
```

# 2 Load Movielens Data

Instead of using Movielens Latest Small Data, we're going to use Movielens 100K Data as it is a stable benchmark and won't change over time

[3]:
```python
ratings, movies = ml100k.load()
```

```
Download data 100.2%
Successfully downloaded ml-100k.zip 4924029 bytes.
Unzipping the ml-100k.zip zip file ...
```

What's the difference between *Movielens Latest Small Data* and *Movielens 100K Data*? According to GroupLens, *Movielens Latest Small Data* will change overtime while *Movielens 100K Data* is a stable benchmark released in 1998 and won't change over time

What's the difference between those two data in terms of content?

[4]:
```
ratings.head()
```

[4]:

| | userid | itemid | rating |
|---|---|---|---|
| 0 | 1 | 1 | 5 |
| 1 | 1 | 2 | 3 |
| 2 | 1 | 3 | 4 |
| 3 | 1 | 4 | 3 |
| 4 | 1 | 5 | 3 |

*Movielens Latest Small Data*'s *ratings* has a timestamp column while *Movielens 100K Data*'s *ratings* doesn't have that column

[5]:
```
movies.head()
```

[5]:

| | itemid | title |
|---|---|---|
| 0 | 1 | Toy Story (1995) |
| 1 | 2 | GoldenEye (1995) |
| 2 | 3 | Four Rooms (1995) |
| 3 | 4 | Get Shorty (1995) |
| 4 | 5 | Copycat (1995) |

*Movielens Latest Small Data*'s *movies* has a genres column while *Movielens 100K Data*'s *movies* doesn't have that column

## 3 Encoding of *userids* and *itemids*

All *userid* and *itemid* in *ratings* could have a non-consecutive sequence when ids are ordered, For convenience at the construction of the matrix, encode each of those to a consecutive sequence when ids are ordered through LabelEncoder

[6]:
```
# Encode userids and itemids in ratings through LabelEncoder
# uencoder -> LabelEncoder object of userids
# iencoder -> LabelEncoder object of itemids
ratings, uencoder, iencoder = ids_encoder(ratings)
```

## 4 Transform *ratings* dataframe to matrix

[7]:
```
def ratings_matrix(ratings):
    return csr_matrix(pd.crosstab(ratings.userid, ratings.itemid, ratings.
 ↪rating, aggfunc=sum).fillna(0).values)

R = ratings_matrix(ratings)
```

[8]:
```
display(R)
print('Total users: {}'.format(len(ratings['userid'].unique())))
print('Total movies: {}'.format(len(ratings['itemid'].unique())))
```

```
<943x1682 sparse matrix of type '<class 'numpy.float64'>'
with 100000 stored elements in Compressed Sparse Row format>
```

```
Total users: 943
Total movies: 1682
```

943 rows represents the users through userid, while 1682 columns represents the movies through itemid

# 5 Memory based collaborative filtering

Memory based collaborative filtering (CF) also known as nearest neighbors based CF makes recommendation based on similar behavious of users and items. There are two types of memory based CF : user-based and item-based CF. Both of these algorithm usually proceed in three stages :

1. Similarity computation (between users or items)
2. Rating prediction (using ratings of similar users or items)
3. Top-N recommendation

## 5.1 User-based Collaborative Filtering

### 5.1.1 Idea

Let $u$ be the user for which we plan to make recommendations.

1. Find other users whose past rating behavior is similar to that of $u$
2. Use their ratings on other items to predict what the current user will like

### 5.1.2 Algorithm : user-to-user collaborative filtering

The entire process of user-to-user CF algorithm is described as follow (J. Bobadilla et al. 2013): For an active user $u$,

First identify the set $G_u$ of $k$ most similar users. $G_u$ is the group users similar to the active user $u$. The similarity between two users $u$ and $v$ can be measured by the cosine similarity measure as follows :

$$w_{u,v} = \frac{\vec{r}_u \cdot \vec{r}_v}{\|\vec{r}_u\|_2 * \|\vec{r}_v\|_2} = \frac{\sum_{i \in I} r_{u,i} r_{v,i}}{\sqrt{\sum_{i \in I}(r_{u,i})^2}\sqrt{\sum_{i \in I}(r_{v,i})^2}} \tag{1}$$

$w_{u,v}$ is the degree of similarity between users $u$ and $v$. This term is computed for all $v \in U$, where $U$ is the set of all users. There remains the question of how many neighbors to select. As experimented by (Herlocker et al. 1999), $k \in [20, 50]$ is a reasonable starting point in many domains.

Find the set $C$ of candidate items, purchased by the group and not purchased by the active user $u$. Candidate items have to be the most frequent items purchased by the group.

Aggregate ratings of users in $G_u$ to make predictions for user $u$ on items he has not already purchased. Several aggregation approaches are often used such as average, weighted sum, ajusted weighted sum. By using weighted sum, the predicted rating of user $u$ on item $i$ denoted by $\hat{r}_{u,i}$ is computed as follow :

$$\hat{r}_{u,i} = \bar{r}_u + \frac{\sum_{v \in G_u} (r_{v,i} - \bar{r}_v) \cdot w_{u,v}}{\sum_{v \in G_u} |w_{u,v}|}. \tag{2}$$

Ratings of similar users are weighted by the corresponding similarity with the active user. Summation are made over all the users who have rated item $i$. Subtracting the user's mean rating $\bar{r}_v$ compensates for differences in users' use of the rating scale as some users will tend to give higher ratings than others (Michael D. Ekstrand, et al. 2011). This prediction is made for all items $i \in C$ not purchased by user $u$.

The Top-$N$ recommendations are obtained by choosing the $N$ items which provide most satisfaction to the user according to prediction.

**Step 1 - Identify $G_u$, the set of $k$ users similar to an active user $u$**

To find the $k$ most similar users to $u$, we use the cosine similarity and compute $w_{u,v}$ for all $v \in U$. Fortunately, libraries such as scikit-learn (sklearn) are very useful for such tasks :

1. First of all, we create a nearest neighbors model with sklearn through the function `create_model()`. This function creates and fit a nearest neighbors model with user's ratings. We can choose `cosine` or `euclidian` based similarity metric. `n_neighbors=21` define the number of neighbors to return. With $k = 20$ neighbors, $|G_u| = 21$ as $G_u$ contains 20 similar users added to the active user $u$. That is why `n_neighbors=21`. Each row $r_u$ of the rating matrix $R$ represents ratings of user $u$ on all items of the database. Missing ratings are replaced with 0.0.

```
R[u,:] # uth row of the rating matrix R. Ratings of user u on all items in the database
```

2. Function `nearest_neighbors()` returns the knn users for each user.

[9]:
```python
def create_model(rating_matrix, metric):
    """
    - create the nearest neighbors model with the corresponding similarity metric
    - fit the model
    """
    model = NearestNeighbors(metric=metric, n_neighbors=21, algorithm='brute')
    model.fit(rating_matrix)
    return model
```

[10]:
```python
def nearest_neighbors(rating_matrix, model):
    """
    :param rating_matrix : rating matrix of shape (nb_users, nb_items)
    :param model : nearest neighbors model
    :return
        - similarities : distances of the neighbors from the referenced user
```

```
        - neighbors : neighbors of the referenced user in decreasing order of␣
↪similarities
    """
    similarities, neighbors = model.kneighbors(rating_matrix)
    return similarities[:, 1:], neighbors[:, 1:]
```

Let's call functions `create_model()` and `nearest_neighbors()` to respectively create the $k$-NN model and compute the nearest neighbors for a given user

[11]:
```
model = create_model(rating_matrix=R, metric='cosine') # we can also use the␣
↪'euclidian' distance
similarities, neighbors = nearest_neighbors(R, model)
```

[12]:
```
print('similarities shape: ', similarities.shape)
print('neighbors shape: ', neighbors.shape)
```

```
similarities shape:  (943, 20)
neighbors shape:  (943, 20)
```

In *similarities* and *neighbors*, each row represents each user while its columns represent its neighbors with the highest similarities *similarities*'s values are the similarity values, and *neighbors*'s values are the index of the neighbors

**Step 2 - Find candidate items**

The set $C$ of candidate items are the most frequent ones purchased by users in $G_u$ for an active user $u$ and not purchased by $u$.

Function `find_candidate_items()` : find items purchased by these similar users as well as their frequency. Note that the frequency of the items in the set $C$ can be computed by just counting the actual occurrence frequency of that items.

1. `Gu_items` : frequent items of $G_u$ in decreasing order of frequency.
2. `active_items` : items already purchased by the active user
3. `candidates` : frequent items of $G_u$ not purchased by the active user $u$

[13]:
```
def find_candidate_items(userid):
    """
    Find candidate items for an active user

    :param userid : active user
    :param neighbors : users similar to the active user
    :return candidates : top 30 of candidate items
    """
    # find neighbors of user with index = userid
    user_neighbors = neighbors[userid]
    # find ratings activity of its neighbors
    activities = ratings.loc[ratings.userid.isin(user_neighbors)]

    # sort items in decreasing order of frequency
```

```
    frequency = activities.groupby('itemid')['rating'].count().
↪reset_index(name='count').sort_values(['count'],ascending=False)
    Gu_items = frequency.itemid
    active_items = ratings.loc[ratings.userid == userid].itemid.to_list()
    candidates = np.setdiff1d(Gu_items, active_items, assume_unique=True)[:30]

    return candidates
```

**Step 3 - Rating prediction**

Now it's time to predict what score the active user $u$ would have given to each of the top-30 candidate items.

To predict the score of $u$ on a candidate item $i$ ,we need : 1. Similarities between $u$ and all his neighbors $v \in G_u$ who rated item $i$ : function `nearest_neighbors()` returns similar users of a user as well as their corresponding similarities. 2. Normalized ratings of all $v \in G_u$ on item $i$. The normalized rating of user $v$ on item $i$ is defined by $r_{v,i} - \bar{r}_v$.

Next, let's compute the mean rating of each user and the normalized ratings for each item. The DataFrame `mean` contains mean rating for each user. With the mean rating of each user, we can add an extra column `norm_rating` to the `ratings`'s DataFrame which can be accessed to make predictions.

[14]:
```
# mean ratings for each user
mean = ratings.groupby(by='userid', as_index=False)['rating'].mean()
mean_ratings = pd.merge(ratings, mean, suffixes=('','_mean'), on='userid')

# normalized ratings for each items
mean_ratings['norm_rating'] = mean_ratings['rating'] -␣
↪mean_ratings['rating_mean']

mean = mean.to_numpy()[:, 1]
```

[15]:
```
np_ratings = mean_ratings.to_numpy()
```

Let us define function `predict` that predict rating between user $u$ and item $i$. Recall that the prediction formula is defined as follow :

$$\hat{r}_{u,i} = \bar{r}_u + \frac{\sum_{v \in G_u} (r_{v,i} - \bar{r}_v) \cdot w_{u,v}}{\sum_{v \in G_u} |w_{u,v}|}. \tag{3}$$

[16]:
```
def predict(userid, itemid):
    """
    predict what score userid would have given to itemid.

    :param
        - userid : user id for which we want to make prediction
        - itemid : item id on which we want to make prediction
```

```
    :return
        - r_hat : predicted rating of user userid on item itemid
    """
    user_similarities = similarities[userid]
    user_neighbors = neighbors[userid]
    # get mean rating of user userid
    user_mean = mean[userid]

    # find users who rated item 'itemid'
    iratings = np_ratings[np_ratings[:, 1].astype('int') == itemid]

    # find similar users to 'userid' who rated item 'itemid'
    suri = iratings[np.isin(iratings[:, 0], user_neighbors)]

    # similar users who rated current item (surci)
    normalized_ratings = suri[:,4]
    indexes = [np.where(user_neighbors == uid)[0][0] for uid in suri[:, 0].
↪astype('int')]
    sims = user_similarities[indexes]

    num = np.dot(normalized_ratings, sims)
    den = np.sum(np.abs(sims))

    if num == 0 or den == 0:
        return user_mean

    r_hat = user_mean + np.dot(normalized_ratings, sims) / np.sum(np.abs(sims))

    return r_hat
```

Now, we can make rating prediction for a given user on each item in his set of candidate items.

[17]:
```
def user2userPredictions(userid, pred_path):
    """
    Make rating prediction for the active user on each candidate item and save␣
↪in file prediction.csv

    :param
        - userid : id of the active user
        - pred_path : where to save predictions
    """
    # find candidate items for the active user
    candidates = find_candidate_items(userid)

    # loop over candidates items to make predictions
    for itemid in candidates:
```

```
        # prediction for userid on itemid
        r_hat = predict(userid, itemid)

        # save predictions
        with open(pred_path, 'a+') as file:
            line = '{},{},{}\n'.format(userid, itemid, r_hat)
            file.write(line)
```

[18]:
```
import sys

def user2userCF():
    """
    Make predictions for each user in the database.
    """
    # get list of users in the database
    users = ratings.userid.unique()

    def _progress(count):
        sys.stdout.write('\rRating predictions. Progress status : %.1f%%' %␣
 ↪(float(count/len(users))*100.0))
        sys.stdout.flush()

    saved_predictions = 'predictions.csv'
    if os.path.exists(saved_predictions):
        os.remove(saved_predictions)

    for count, userid in enumerate(users):
        # make rating predictions for the current user
        user2userPredictions(userid, saved_predictions)
        _progress(count)
```

[19]:
```
user2userCF()
```

```
Rating predictions. Progress status : 99.9%
```

**Step 4 - Top-N recommendation**

Function `user2userRecommendation()` reads predictions for a given user and return the list of items in decreasing order of predicted rating.

[20]:
```
def user2userRecommendation(userid):
    """
    """
    # encode the userid
    uid = uencoder.transform([userid])[0]
    saved_predictions = 'predictions.csv'
```

```
    predictions = pd.read_csv(saved_predictions, sep=',', names=['userid',␣
↪'itemid', 'predicted_rating'])
    predictions = predictions[predictions.userid==uid]
    List = predictions.sort_values(by=['predicted_rating'], ascending=False)

    List.userid = uencoder.inverse_transform(List.userid.tolist())
    List.itemid = iencoder.inverse_transform(List.itemid.tolist())

    List = pd.merge(List, movies, on='itemid', how='inner')

    return List
```

For example, we want to see the recommendation for user with id 212

[21]:
```
user2userRecommendation(212)
```

[21]:
```
    userid  itemid  predicted_rating  \
0      212     483          4.871495
1      212     357          4.764547
2      212      50          4.660002
3      212      98          4.613636
4      212      64          4.550733
5      212     194          4.522336
6      212     174          4.521300
7      212     134          4.414819
8      212     187          4.344531
9      212     196          4.303696
10     212     523          4.281802
11     212     216          4.278246
12     212     100          4.260087
13     212     168          4.206139
14     212     435          4.122984
15     212     135          4.115228
16     212      83          4.106995
17     212      69          4.086366
18     212      70          4.086328
19     212     275          3.985037
20     212     153          3.981619
21     212     514          3.956640
22     212     521          3.937792
23     212      97          3.906106
24     212     173          3.879325
25     212     660          3.847897
26     212     215          3.709920
27     212     258          3.583718
28     212     202          3.508617
29     212     237          3.039041
```

```
                                  title
0                      Casablanca (1942)
1     One Flew Over the Cuckoo's Nest (1975)
2                       Star Wars (1977)
3            Silence of the Lambs, The (1991)
4            Shawshank Redemption, The (1994)
5                      Sting, The (1973)
6              Raiders of the Lost Ark (1981)
7                    Citizen Kane (1941)
8              Godfather: Part II, The (1974)
9                 Dead Poets Society (1989)
10                  Cool Hand Luke (1967)
11             When Harry Met Sally... (1989)
12                          Fargo (1996)
13     Monty Python and the Holy Grail (1974)
14  Butch Cassidy and the Sundance Kid (1969)
15               2001: A Space Odyssey (1968)
16              Much Ado About Nothing (1993)
17                   Forrest Gump (1994)
18          Four Weddings and a Funeral (1994)
19               Sense and Sensibility (1995)
20                Fish Called Wanda, A (1988)
21                     Annie Hall (1977)
22                 Deer Hunter, The (1978)
23                Dances with Wolves (1990)
24                Princess Bride, The (1987)
25               Fried Green Tomatoes (1991)
26                  Field of Dreams (1989)
27                        Contact (1997)
28                   Groundhog Day (1993)
29                   Jerry Maguire (1996)
```

We can see that the above movies are the movies which predicted to be high rated by user with id 212 So, we'll recommend those movies to the user with id 212

**Stage 5. Evaluation with Mean Absolute Error (MAE)**

[22]:
```python
from recsys.preprocessing import train_test_split, get_examples

# get examples as tuples of userids and itemids and labels from normalize ratings
raw_examples, raw_labels = get_examples(ratings, labels_column='rating')

# train test split
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)

def evaluate(x_test, y_test):
```

```
    print('Evaluate the model on {} test data ...'.format(x_test.shape[0]))
    preds = list(predict(u,i) for (u,i) in x_test)
    mae = np.sum(np.absolute(y_test - np.array(preds))) / x_test.shape[0]
    print('\nMAE :', mae)
    return mae
```

[23]:
```
evaluate(x_test, y_test)
```

```
Evaluate the model on 10000 test data ...

MAE : 0.7505910931068639
```

[23]:
```
0.7505910931068639
```

According to the result, the MAE is < 1.0, which is quite okay

**Summary** For Convenience, User-based Collaborative Filtering can be used by calling UserToUser class from *recsys* On how to use that class, the process is described as below

**Evaluation on the ML-100K dataset**

[24]:
```
from recsys.memories.UserToUser import UserToUser

# load ml100k ratings
ratings, movies = ml100k.load()

# prepare data
ratings, uencoder, iencoder = ids_encoder(ratings)

# get examples as tuples of userids and itemids and labels from normalize ratings
raw_examples, raw_labels = get_examples(ratings, labels_column='rating')

# train test split
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
 ↪labels=raw_labels)
```

[25]:
```
# create the user-based CF
usertouser = UserToUser(ratings, movies, metric='cosine')
```

```
Normalize users ratings ...
Initialize the similarity model ...
Compute nearest neighbors ...
User to user recommendation model created with success ...
```

[26]:
```
# evaluate the user-based CF on the ml100k test data
usertouser.evaluate(x_test, y_test)
```

```
Evaluate the model on 10000 test data ...

MAE : 0.7505910931068639
```

[26]: 0.7505910931068639

**Evaluation on the ML-1M dataset (this may take some time)**

[27]:
```
from recsys.datasets import ml1m
from recsys.preprocessing import ids_encoder, get_examples, train_test_split
from recsys.memories.UserToUser import UserToUser

# load ml100k ratings
ratings, movies = ml1m.load()

# prepare data
ratings, uencoder, iencoder = ids_encoder(ratings)

# get examples as tuples of userids and itemids and labels from normalize ratings
raw_examples, raw_labels = get_examples(ratings, labels_column='rating')

# train test split
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
 ↪labels=raw_labels)

# create the user-based CF
usertouser = UserToUser(ratings, movies, k=20, metric='cosine')

# evaluate the user-based CF on the ml1m test data
print("==========================")
usertouser.evaluate(x_test, y_test)
```

```
Download data 100.1%
Successfully downloaded ml-1m.zip 5917549 bytes.
Unzipping the ml-1m.zip zip file ...
Normalize users ratings ...
Initialize the similarity model ...
Compute nearest neighbors ...
User to user recommendation model created with success ...
==========================
Evaluate the model on 100021 test data ...

MAE : 0.732267005840993
```

[27]: 0.732267005840993

# 6 Limitations of user-based CF

1. Sparsity : In general, users interact with less than 20% of items. This leads the rating matrix to be highly sparse. For example, the movielen-100k contains 100k ratings from 943 users on 1682 items. The pourcentage of sparsity in this case is around 94%. A recommender system based on nearest neighbor algorithms may be unable to make any item recommendations

for a particular user. As a result the accuracy of recommendations may be poor (Sarwar et al. 2001).

2. Stability of user's ratings : As a user rates and re-rates items, their rating vector will change along with their similarity to other users. A user's neighborhood is determined not only by their ratings but also by the ratings of other users, so their neighborhood can change as a result of new ratings supplied by any user in the system (Michael D. Ekstrand, et al. 2011).

3. Scalability : Due to the non-stability of users ratings, finding similar users in advance is complicated. For this reason, most user-based CF systems find neighborhoods each time predictions or recommendations are needed. However, these are huge computations that grows with both the number of users and the number of items. With millions of users and items, a typical web-based recommender system running existing algorithms will suffer serious scalability concerns (Sarwar et al. 2001), (Michael D. Ekstrand, et al. 2011).

# 7 References

1. Herlocker et al. (1999) An Algorithmic Framework for Performing Collaborative Filtering
2. Sarwar et al. (2001) Item-based collaborative filtering recommendation algorithms
3. Michael D. Ekstrand, et al. (2011). Collaborative Filtering Recommender Systems
4. J. Bobadilla et al. (2013) Recommender systems survey

# 8 Author

Carmel WENGA, PhD student at Université de la Polynésie Française, Applied Machine Learning Research Engineer, ShoppingList, NzhinuSoft.

# Chapter 3 - Item-based Collaborative Filtering

Before we start, we have to make sure that we already have our dependencies, that is 'recsys' folder

[1]:
```
import os
#Check if we already have the 'recsys' folder
if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    # If not then download directly from the source
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/
↪master/recsys.zip
    !unzip recsys.zip
```

```
--2023-01-03 20:58:12--  https://github.com/nzhinusoftcm/review-on-
collaborative-filtering/raw/master/recsys.zip
Resolving github.com (github.com)... 140.82.114.3
Connecting to github.com (github.com)|140.82.114.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/nzhinusoftcm/review-on-
collaborative-filtering/master/recsys.zip [following]
--2023-01-03 20:58:12--  https://raw.githubusercontent.com/nzhinusoftcm/review-
on-collaborative-filtering/master/recsys.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 15312323 (15M) [application/zip]
Saving to: 'recsys.zip'

recsys.zip          100%[===================>]  14.60M  --.-KB/s    in 0.05s

2023-01-03 20:58:12 (317 MB/s) - 'recsys.zip' saved [15312323/15312323]

Archive:  recsys.zip
   creating: recsys/
  inflating: recsys/datasets.py
  inflating: recsys/preprocessing.py
  inflating: recsys/utils.py
  inflating: recsys/requirements.txt
```

```
 creating: recsys/.vscode/
inflating: recsys/.vscode/settings.json
 creating: recsys/__pycache__/
inflating: recsys/__pycache__/datasets.cpython-36.pyc
inflating: recsys/__pycache__/datasets.cpython-37.pyc
inflating: recsys/__pycache__/utils.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-37.pyc
inflating: recsys/__pycache__/datasets.cpython-38.pyc
inflating: recsys/__pycache__/preprocessing.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-38.pyc
 creating: recsys/memories/
inflating: recsys/memories/ItemToItem.py
inflating: recsys/memories/UserToUser.py
 creating: recsys/memories/__pycache__/
inflating: recsys/memories/__pycache__/UserToUser.cpython-36.pyc
inflating: recsys/memories/__pycache__/UserToUser.cpython-37.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-37.pyc
inflating: recsys/memories/__pycache__/user2user.cpython-36.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-36.pyc
 creating: recsys/models/
inflating: recsys/models/SVD.py
inflating: recsys/models/MatrixFactorization.py
inflating: recsys/models/ExplainableMF.py
inflating: recsys/models/NonnegativeMF.py
 creating: recsys/models/__pycache__/
inflating: recsys/models/__pycache__/SVD.cpython-36.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-37.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-36.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-37.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-36.pyc
 creating: recsys/metrics/
inflating: recsys/metrics/EvaluationMetrics.py
 creating: recsys/img/
inflating: recsys/img/MF-and-NNMF.png
inflating: recsys/img/svd.png
inflating: recsys/img/MF.png
 creating: recsys/predictions/
 creating: recsys/predictions/item2item/
 creating: recsys/weights/
 creating: recsys/weights/item2item/
 creating: recsys/weights/item2item/ml1m/
inflating: recsys/weights/item2item/ml1m/similarities.npy
inflating: recsys/weights/item2item/ml1m/neighbors.npy
 creating: recsys/weights/item2item/ml100k/
inflating: recsys/weights/item2item/ml100k/similarities.npy
inflating: recsys/weights/item2item/ml100k/neighbors.npy
```

# 1 Requirements

Other than the 'recsys' folder, we also have to make sure that the other required libs have already been installed

```
matplotlib==3.2.2
numpy==1.19.2
pandas==1.0.5
python==3.7
scikit-learn==0.24.1
scikit-surprise==1.1.1
scipy==1.6.2
```

(If we use Google Colab, most of these libs are already installed and up-to-date, except for *scikit-surprise* which is not pre-installed by Google Colab)

To install scikit-surprise on Google Colab, we must execute the code below

```
!pip install surprise
```

But this notebook doesn't require this library yet, so we're not going to install scikit-surprise at the moment

Import all of the required libs

[2]:
```
from sklearn.neighbors import NearestNeighbors
from scipy.sparse import csr_matrix

from recsys.datasets import ml1m, ml100k
from recsys.preprocessing import ids_encoder

import pandas as pd
import numpy as np
import os
import sys
```

# 2 Load Movielens Data

Instead of using Movielens Latest Small Data, we're going to use Movielens 100K Data as it is a stable benchmark and won't change over time

[3]:
```
ratings, movies = ml100k.load()
```

```
Download data 100.2%
Successfully downloaded ml-100k.zip 4924029 bytes.
Unzipping the ml-100k.zip zip file ...
```

What's the difference between *Movielens Latest Small Data* and *Movielens 100K Data*? According to GroupLens, *Movielens Latest Small Data* will change overtime while *Movielens 100K Data* is a stable benchmark released in 1998 and won't change over time

What's the difference between those two data in terms of content?

[4]:
```
ratings.head()
```

[4]:

| | userid | itemid | rating |
|---|---|---|---|
| 0 | 1 | 1 | 5 |
| 1 | 1 | 2 | 3 |
| 2 | 1 | 3 | 4 |
| 3 | 1 | 4 | 3 |
| 4 | 1 | 5 | 3 |

*Movielens Latest Small Data*'s *ratings* has a timestamp column while *Movielens 100K Data*'s *ratings* doesn't have that column

[5]:
```
movies.head()
```

[5]:

| | itemid | title |
|---|---|---|
| 0 | 1 | Toy Story (1995) |
| 1 | 2 | GoldenEye (1995) |
| 2 | 3 | Four Rooms (1995) |
| 3 | 4 | Get Shorty (1995) |
| 4 | 5 | Copycat (1995) |

*Movielens Latest Small Data*'s *movies* has a genres column while *Movielens 100K Data*'s *movies* doesn't have that column

# 3 Encoding of *userids* and *itemids*

All *userid* and *itemid* in *ratings* could have a non-consecutive sequence when ids are ordered, For convenience at the construction of the matrix, encode each of those to a consecutive sequence when ids are ordered through LabelEncoder

[6]:
```
# Encode userids and itemids in ratings through LabelEncoder
# uencoder -> LabelEncoder object of userids
# iencoder -> LabelEncoder object of itemids
ratings, uencoder, iencoder = ids_encoder(ratings)
```

Now that the initial configuration is done, let's implement the item-based collaborative filtering algorithm

Item-based collaborative filtering is still a part of memory-based collaborative filtering. The user-based collaborative filtering has already been explained in the previous chapter, so now we'll take a look at the item-based collaborative filtering

# 4 Item-to-Item Collaborative Filtering

## 4.1 Idea

Let $u$ be the active user and $i$ the referenced item 1. If $u$ liked items similar to $i$, he will probably like item $i$. 2. If he hated or disliked items similar to $i$, he will also hate item $i$.

The idea is therefore to look at how an active user $u$ rated items similar to $i$ to know how he would have rated item $i$

### 4.2 Algorithm : item-to-item collaborative filtering

The algorithm that defines item-based CF is described as follow (B. Sarwar et al. 2001)(George Karypis 2001) :

First identify the $k$ most similar items for each item in the catalogue and record the corresponding similarities. To compute similarity between two items we can user the Adjusted Cosine Similarity that has proven to be more efficient than the basic Cosine similarity measure used for user-based collaborative as described in (B. Sarwar et al. 2001). The Adjusted Cosine distance between two items $i$ and $j$ is computed as follow

$$w_{i,j} = \frac{\sum_{u \in U}(r_{u,i} - \bar{r}_u)(r_{u,j} - \bar{r}_u)}{\sqrt{\sum_{u \in U}(r_{u,i} - \bar{r}_u)^2}\sqrt{\sum_{u \in U}(r_{u,j} - \bar{r}_u)^2}} \tag{1}$$

$w_{i,j}$ is the degree of similarity between items $i$ and $j$. This term is computed for all users $u \in U$, where $U$ is the set of users that rated both items $i$ and $j$. Let's denote by $S^{(i)}$ the set of the $k$ most similar items to item $i$.

To produce top-N recommendations for a given user $u$ that has already purchased a set $I_u$ of items, do the following :

Find the set $C$ of candidate items by taking the union of all $S^{(i)}, \forall i \in I_u$ and removing each of the items in the set $I_u$.

$$C = \bigcup_{i \in I_u} \{S^{(i)}\} \setminus I_u \tag{2}$$

$\forall c \in C$, compute similarity between c and the set $I_u$ as follows:

$$w_{c,I_u} = \sum_{i \in I_u} w_{c,i}, \forall c \in C \tag{3}$$

Sort items in $C$ in decreasing order of $w_{c,I_u}, \forall c \in C$, and return the first $N$ items as the Top-N recommendation list.

</li>

Before returning the first $N$ items as top-N recommendation list, we can make predictions about what user $u$ would have given to each items in the top-N recommendation list, rearrange the list in descending order of predicted ratings and return the rearranged list as the final recommendation list. Rating prediction for item-based CF is given by the following formular (B. Sarwar et al. 2001):

$$\hat{r}_{u,i} = \frac{\sum_{i \in S^{(i)}} r_{u,j} \cdot w_{i,j}}{\sum_{j \in S^{(i)}} |w_{i,j}|} \tag{4}$$

### 4.2.1 Step 1 - Find similarities for each of the items

To compute similarity between two items *i* and *j*, we need to :

1. Find all users who rated both of them,
2. Normalize their ratings on items *i* and *j*
3. Apply the cosine metric to the normalized ratings to compute similarity between *i* and *j*

Function `normalize()` process the rating dataframe to normalize ratings of all users

[7]:
```
def normalize():
    # compute mean rating for each user
    mean = ratings.groupby(by='userid', as_index=False)['rating'].mean()
    norm_ratings = pd.merge(ratings, mean, suffixes=('','_mean'), on='userid')

    # normalize each rating by substracting the mean rating of the corresponding␣
↪user
    norm_ratings['norm_rating'] = norm_ratings['rating'] -␣
↪norm_ratings['rating_mean']
    return mean.to_numpy()[:, 1], norm_ratings
```

[8]:
```
mean, norm_ratings = normalize()
np_ratings = norm_ratings.to_numpy()
norm_ratings.head()
```

[8]:

| | userid | itemid | rating | rating_mean | norm_rating |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 5 | 3.610294 | 1.389706 |
| 1 | 0 | 1 | 3 | 3.610294 | -0.610294 |
| 2 | 0 | 2 | 4 | 3.610294 | 0.389706 |
| 3 | 0 | 3 | 3 | 3.610294 | -0.610294 |
| 4 | 0 | 4 | 3 | 3.610294 | -0.610294 |

Now that each rating has been normalized, we can represent each item by a vector of its normalized ratings

[9]:
```
def item_representation(ratings):
    return csr_matrix(
        pd.crosstab(ratings.itemid, ratings.userid, ratings.norm_rating,␣
↪aggfunc=sum).fillna(0).values
    )
```

[10]:
```
R = item_representation(norm_ratings)
```

[11]:
```
display(R)
print('Total users: {}'.format(len(ratings['userid'].unique())))
print('Total movies: {}'.format(len(ratings['itemid'].unique())))
```

```
<1682x943 sparse matrix of type '<class 'numpy.float64'>'
with 99650 stored elements in Compressed Sparse Row format>
```

```
Total users: 943
Total movies: 1682
```

1682 rows represents the movies through movieid, while 943 columns represents the users through userid

Let's build and fit our $k$-NN model using sklearn

[12]:
```
def create_model(rating_matrix, k=20, metric="cosine"):
    """
    :param R : numpy array of item representations
    :param k : number of nearest neighbors to return
    :return model : our knn model
    """
    model = NearestNeighbors(metric=metric, n_neighbors=k+1, algorithm='brute')
    model.fit(rating_matrix)
    return model
```

**Similarities computation** Similarities between items can be measured with the *Cosine* or *Eucliedian* distance. The ***NearestNeighbors*** class from the sklearn library simplifies the computation of neighbors. We just need to specify the metric (e.g. cosine or euclidian) that will be used to compute similarities.

The above method, `create_model`, creates the kNN model and the following `nearest_neighbors` method uses the created model to kNN items. It returns nearest neighbors as well as similarities measures for each items.

`nearest_neighbors` returns : - `similarities` : numpy array of shape $(n, k)$ - `neighbors` : numpy array of shape $(n, k)$

where $n$ is the total number of items and $k$ is the number of neighbors to return, specified when creating the kNN model.

[13]:
```
def nearest_neighbors(rating_matrix, model):
    """
    compute the top n similar items for each item.
    :param rating_matrix : items representations
    :param model : nearest neighbors model
    :return similarities, neighbors
    """
    similarities, neighbors = model.kneighbors(rating_matrix)
    return similarities[:,1:], neighbors[:,1:]
```

**Adjusted Cosine Similarity** In the context of item-based collaborative filtering, the adjusted cosine similarity has shown to be more efficient that the cosine or the euclidian distance. Here is the formula to compute the adjusted cosine weight between two items $i$ and $j$ :

$$w_{i,j} = \frac{\sum_{u \in U}(r_{u,i} - \bar{r}_u)(r_{u,j} - \bar{r}_u)}{\sqrt{\sum_{u \in U}(r_{u,i} - \bar{r}_u)^2}\sqrt{\sum_{u \in U}(r_{u,j} - \bar{r}_u)^2}}. \tag{5}$$

This term is computed for all users $u \in U$, where $U$ is the set of users that rated both items $i$ and $j$. Since the *sklearn* library do not directly implement the adjusted cosine similarity metric, we will implement it with the method `adjusted_cosine`, with some helper function :

- `save_similarities` : since the computation of the adjusted cosine similarity is time consuming, around 5 mins for the ml100k dataset, we use this method to save the computed similarities for lated usage.
- `load_similarities` : load the saved similarities
- `cosine` : cosine distance between two vectors.

[14]:
```
def save_similarities(similarities, neighbors, dataset_name):
    base_dir = 'recsys/weights/item2item'
    save_dir = os.path.join(base_dir, dataset_name)
    os.makedirs(save_dir, exist_ok=True)
    similarities_file_name = os.path.join(save_dir, 'similarities.npy')
    neighbors_file_name = os.path.join(save_dir, 'neighbors.npy')
    try:
        np.save(similarities_file_name, similarities)
        np.save(neighbors_file_name, neighbors)
    except ValueError as error:
        print(f"An error occured when saving similarities, due to : \n␣
↪ValueError : {error}")


def load_similarities(dataset_name, k=20):
    base_dir = 'recsys/weights/item2item'
    save_dir = os.path.join(base_dir, dataset_name)
    similiraties_file = os.path.join(save_dir, 'similarities.npy')
    neighbors_file = os.path.join(save_dir, 'neighbors.npy')
    similarities = np.load(similiraties_file)
    neighbors = np.load(neighbors_file)
    return similarities[:,:k], neighbors[:,:k]


def cosine(x, y):
    return np.dot(x, y) / (np.linalg.norm(x) * np.linalg.norm(y))


def adjusted_cosine(np_ratings, nb_items, dataset_name):
    similarities = np.zeros(shape=(nb_items, nb_items))
    similarities.fill(-1)

    def _progress(count):
        sys.stdout.write('\rComputing similarities. Progress status : %.1f%%' %␣
↪(float(count / nb_items)*100.0))
        sys.stdout.flush()

    items = sorted(ratings.itemid.unique())
```

```
    for i in items[:-1]:
        for j in items[i+1:]:
            scores = np_ratings[(np_ratings[:, 1] == i) | (np_ratings[:, 1] ==␣
↪j), :]
            vals, count = np.unique(scores[:,0], return_counts = True)
            scores = scores[np.isin(scores[:,0], vals[count > 1]),:]

            if scores.shape[0] > 2:
                x = scores[scores[:, 1].astype('int') == i, 4]
                y = scores[scores[:, 1].astype('int') == j, 4]
                w = cosine(x, y)

                similarities[i, j] = w
                similarities[j, i] = w
        _progress(i)
    _progress(nb_items)

    # get neighbors by their neighbors in decreasing order of similarities
    neighbors = np.flip(np.argsort(similarities), axis=1)

    # sort similarities in decreasing order
    similarities = np.flip(np.sort(similarities), axis=1)

    # save similarities to disk
    save_similarities(similarities, neighbors, dataset_name=dataset_name)

    return similarities, neighbors
```

Now, we can call the `adjusted_cosine` function to compute and save items similarities and neighbors based on the adjusted cosine metric.

Run these lines to compute the adjusted cosine between all items.

```
nb_items = ratings.itemid.nunique()
similarities, neighbors = adjusted_cosine(np_ratings, nb_items=nb_items, dataset_name='ml100k')
```

As the precomputed version is already available in *recsys*, we don't have to run those lines again. Just load the precomputed version instead for further use.

Among the following similarity metrics, choose the one you wish to use for the item-based collaborative filtering :

- **euclidian** or **cosine** : choose *euclidian* or *cosine* to initialize the similarity model through the sklearn library.
- **adjusted_cosine** : choose the *adjusted_cosine* metric to load similarities computed and saved through the `adjusted_cosine` function.

In this case, we will use the *adjusted_cosine* metric.

[15]:
```
metric = 'adjusted_cosine' # [cosine, euclidean, adjusted_cosine]

if metric == 'adjusted_cosine':
    similarities, neighbors = load_similarities('ml100k')
else:
    model = create_model(R, k=21, metric=metric)
    similarities, neighbors = nearest_neighbors(R, model)
```

[16]:
```
print('neighbors shape : ', neighbors.shape)
print('similarities shape : ', similarities.shape)
```

```
neighbors shape :  (1682, 20)
similarities shape :  (1682, 20)
```

`neighbors` and `similarities` are numpy array, were each entries are list of 20 neighbors with their corresponding similarities

### 4.2.2 Step 2 - Top-N recommendation for a given user

For a user $u$ who has already rated a set of items $I_u$, give top-N movies recommendation

**Finding candidate items** To find candidate items for user $u$, we need to :

1. Find the set $I_u$ of items already rated by user $u$,
2. Take the union of similar items as $C$ for all items in $I_u$
3. Exclude from the set $C$ all items in $I_u$, to avoid recommend to a user items he has already purchased.

These are done in function `candidate_items()`

[17]:
```
def candidate_items(userid):
    """
    :param userid : user id for which we wish to find candidate items
    :return : I_u, candidates
    """

    # 1. Finding the set I_u of items already rated by user userid
    I_u = np_ratings[np_ratings[:, 0] == userid]
    I_u = I_u[:, 1].astype('int')

    # 2. Taking the union of similar items for all items in I_u to form the set␣
 ↪of candidate items
    c = set()

    for iid in I_u:
        # add the neighbors of item iid in the set of candidate items
        c.update(neighbors[iid])

    c = list(c)
```

```
    # 3. exclude from the set C all items in I_u.
    candidates = np.setdiff1d(c, I_u, assume_unique=True)

    return I_u, candidates
```

[18]:
```
#Take a user of testing later
test_user = uencoder.transform([1])[0]
#Find items purchased and candidates items for that user
i_u, u_candidates = candidate_items(test_user)
```

[19]:
```
print('number of items purchased by user 1 : ', len(i_u))
print('number of candidate items for user 1 : ', len(u_candidates))
```

```
number of items purchased by user 1 :  272
number of candidate items for user 1 :  893
```

**Find similarity between each candidate item and the set $I_u$**

[20]:
```
def similarity_with_Iu(c, I_u):
    """
    compute similarity between an item c and a set of items I_u. For each item i␣
 ↪in I_u, get similarity between
    i and c, if c exists in the set of items similar to itemid.
    :param c : itemid of a candidate item
    :param I_u : set of items already purchased by a given user
    :return w : similarity between c and I_u
    """
    w = 0
    for iid in I_u :
        # get similarity between itemid and c, if c is one of the k nearest␣
 ↪neighbors of itemid
        if c in neighbors[iid] :
            w = w + similarities[iid, neighbors[iid] == c][0]
    return w
```

**Rank candidate items according to their similarities to $I_u$**

[21]:
```
def rank_candidates(candidates, I_u):
    """
    rank candidate items according to their similarities with i_u
    :param candidates : list of candidate items
    :param I_u : list of items purchased by the user
    :return ranked_candidates : dataframe of candidate items, ranked in␣
 ↪descending order of similarities with I_u
    """

    # list of candidate items mapped to their corresponding similarities to I_u
    sims = [similarity_with_Iu(c, I_u) for c in candidates]
    candidates = iencoder.inverse_transform(candidates)
```

```
    mapping = list(zip(candidates, sims))

    ranked_candidates = sorted(mapping, key=lambda couple:couple[1],␣
↪reverse=True)
    return ranked_candidates
```

**Putting all together**

Now that we defined all functions necessary to build our item to item top-N recommendation, let's define function `topn_recommendation()` that makes top-$N$ recommendations for a given user

[22]:
```
def topn_recommendation(userid, N=30):
    """
    Produce top-N recommendation for a given user
    :param userid : user for which we produce top-N recommendation
    :param n : length of the top-N recommendation list
    :return topn
    """
    # find candidate items
    I_u, candidates = candidate_items(userid)

    # rank candidate items according to their similarities with I_u
    ranked_candidates = rank_candidates(candidates, I_u)

    # get the first N row of ranked_candidates to build the top N recommendation␣
↪list
    topn = pd.DataFrame(ranked_candidates[:N],␣
↪columns=['itemid','similarity_with_Iu'])
    topn = pd.merge(topn, movies, on='itemid', how='inner')
    return topn
```

[23]:
```
#Show top-30 movies recommendation for the test user
topn_recommendation(test_user)
```

[23]:
```
    itemid  similarity_with_Iu  \
0     1356           52.867173
1     1189           50.362199
2     1516           31.133267
3     1550           31.031738
4     1554           27.364494
5     1600           27.287712
6     1223           26.631850
7     1388           26.624397
8      766           26.590175
9      691           26.461802
10    1378           25.787842
11    1664           25.327445
12    1261           24.785660
```

| | | |
|---|---|---|
| 13 | 1123 | 24.524028 |
| 14 | 1538 | 24.492453 |
| 15 | 1485 | 24.345312 |
| 16 | 1450 | 24.262120 |
| 17 | 909 | 23.357301 |
| 18 | 359 | 22.973658 |
| 19 | 1369 | 22.710078 |
| 20 | 1506 | 22.325504 |
| 21 | 1537 | 22.061914 |
| 22 | 1474 | 21.877034 |
| 23 | 1467 | 21.861203 |
| 24 | 1255 | 21.750924 |
| 25 | 1499 | 21.529748 |
| 26 | 1466 | 21.063269 |
| 27 | 1448 | 20.846909 |
| 28 | 927 | 20.730153 |
| 29 | 1375 | 20.627152 |

| | title |
|---|---|
| 0 | Ed's Next Move (1996) |
| 1 | Prefontaine (1997) |
| 2 | Wedding Gift, The (1994) |
| 3 | Destiny Turns on the Radio (1995) |
| 4 | Safe Passage (1994) |
| 5 | Guantanamera (1994) |
| 6 | King of the Hill (1993) |
| 7 | Gabbeh (1996) |
| 8 | Man of the Year (1995) |
| 9 | Dark City (1998) |
| 10 | Rhyme & Reason (1997) |
| 11 | 8 Heads in a Duffel Bag (1997) |
| 12 | Run of the Country, The (1995) |
| 13 | Last Time I Saw Paris, The (1954) |
| 14 | All Over Me (1997) |
| 15 | Colonel Chabert, Le (1994) |
| 16 | Golden Earrings (1947) |
| 17 | Dangerous Beauty (1998) |
| 18 | Assignment, The (1997) |
| 19 | Forbidden Christ, The (Cristo proibito, Il) (1... |
| 20 | Nelly & Monsieur Arnaud (1995) |
| 21 | Cosi (1996) |
| 22 | Nina Takes a Lover (1994) |
| 23 | Saint of Fort Washington, The (1993) |
| 24 | Broken English (1996) |
| 25 | Grosse Fatigue (1994) |
| 26 | Margaret's Museum (1995) |
| 27 | My Favorite Season (1993) |

```
28  Flower of My Secret, The (Flor de mi secreto, ...
29                            Cement Garden, The (1993)
```

This dataframe represents the top N recommendation list a user. These items are sorted in decreasing order of similarities with $I_u$.

**Observation** : The recommended items are the most similar to the set $I_u$ of items already purchased by the user.

### 4.2.3 Extra: Top-N recommendation with rating *predictions*

Before recommending the previous list to the user, we can go further and predict the ratings the user would have given to each of these items, sort them in descending order of prediction and return the reordered list as the new top N recommendation list.

**Rating prediction** As stated earlier, the predicted rating $\hat{r}_{u,i}$ for a given user $u$ on an item $i$ is obtained by aggregating ratings given by $u$ on items similar to $i$ as follows:

$$\hat{r}_{u,i} = \frac{\sum_{j \in S^{(i)}} r_{u,j} \cdot w_{i,j}}{\sum_{j \in S^{(i)}} |w_{i,j}|} \tag{6}$$

[24]:
```
def predict(userid, itemid):
    """
    Make rating prediction for user userid on item itemid
    :param userid : id of the active user
    :param itemid : id of the item for which we are making prediction
    :return r_hat : predicted rating
    """

    # Get items similar to item itemid with their corresponding similarities
    item_neighbors = neighbors[itemid]
    item_similarities = similarities[itemid]

    # get ratings of user with id userid
    uratings = np_ratings[np_ratings[:, 0].astype('int') == userid]

    # similar items rated by item the user of i
    siru = uratings[np.isin(uratings[:, 1], item_neighbors)]
    scores = siru[:, 2]
    indexes = [np.where(item_neighbors == iid)[0][0] for iid in siru[:,1].
↪astype('int')]
    sims = item_similarities[indexes]

    dot = np.dot(scores, sims)
    som = np.sum(np.abs(sims))

    if dot == 0 or som == 0:
```

```
        return mean[userid]

    return dot / som
```

Now let's use our predict() function to predict what ratings the user would have given to the previous top-$N$ list and return the reorganised list (in decreasing order of predictions) as the new top-$N$ list

[25]:
```
def topn_prediction(userid):
    """
    :param userid : id of the active user
    :return topn : initial topN recommendations returned by the function␣
↪topn_recommendation
    :return topn_predict : topN recommendations reordered according to rating␣
↪predictions
    """
    # make top N recommendation for the active user
    topn = topn_recommendation(userid)

    # get list of items of the top N list
    itemids = topn.itemid.to_list()

    predictions = []

    # make prediction for each item in the top N list
    for itemid in itemids:
        r = predict(userid, itemid)

        predictions.append((itemid,r))

    predictions = pd.DataFrame(predictions, columns=['itemid','prediction'])

    # merge the predictions to topN_list and rearrange the list according to␣
↪predictions
    topn_predict = pd.merge(topn, predictions, on='itemid', how='inner')
    topn_predict = topn_predict.sort_values(by=['prediction'], ascending=False)

    return topn, topn_predict
```

Now, let's make recommendation for test user and compare both list

[26]:
```
topn, topn_predict = topn_prediction(userid=test_user)
# topn -> first list
# topn_predict -> second list
```

[27]:
```
topn_predict
```

[27]:

| | itemid | similarity_with_Iu |
|---|---|---|
| 7 | 1388 | 26.624397 |
| 18 | 359 | 22.973658 |
| 4 | 1554 | 27.364494 |
| 14 | 1538 | 24.492453 |
| 27 | 1448 | 20.846909 |
| 29 | 1375 | 20.627152 |
| 26 | 1466 | 21.063269 |
| 2 | 1516 | 31.133267 |
| 23 | 1467 | 21.861203 |
| 21 | 1537 | 22.061914 |
| 10 | 1378 | 25.787842 |
| 19 | 1369 | 22.710078 |
| 3 | 1550 | 31.031738 |
| 1 | 1189 | 50.362199 |
| 20 | 1506 | 22.325504 |
| 15 | 1485 | 24.345312 |
| 11 | 1664 | 25.327445 |
| 9 | 691 | 26.461802 |
| 6 | 1223 | 26.631850 |
| 5 | 1600 | 27.287712 |
| 17 | 909 | 23.357301 |
| 12 | 1261 | 24.785660 |
| 24 | 1255 | 21.750924 |
| 13 | 1123 | 24.524028 |
| 16 | 1450 | 24.262120 |
| 22 | 1474 | 21.877034 |
| 8 | 766 | 26.590175 |
| 0 | 1356 | 52.867173 |
| 28 | 927 | 20.730153 |
| 25 | 1499 | 21.529748 |

\

| | title | prediction |
|---|---|---|
| 7 | Gabbeh (1996) | 4.666667 |
| 18 | Assignment, The (1997) | 4.600000 |
| 4 | Safe Passage (1994) | 4.500000 |
| 14 | All Over Me (1997) | 4.500000 |
| 27 | My Favorite Season (1993) | 4.490052 |
| 29 | Cement Garden, The (1993) | 4.333333 |
| 26 | Margaret's Museum (1995) | 4.271915 |
| 2 | Wedding Gift, The (1994) | 4.000000 |
| 23 | Saint of Fort Washington, The (1993) | 4.000000 |
| 21 | Cosi (1996) | 4.000000 |
| 10 | Rhyme & Reason (1997) | 4.000000 |
| 19 | Forbidden Christ, The (Cristo proibito, Il) (1... | 4.000000 |
| 3 | Destiny Turns on the Radio (1995) | 3.777778 |
| 1 | Prefontaine (1997) | 3.666528 |

```
20                      Nelly & Monsieur Arnaud (1995)    3.610294
15                          Colonel Chabert, Le (1994)    3.610294
11                      8 Heads in a Duffel Bag (1997)    3.610294
9                                  Dark City (1998)    3.610294
6                           King of the Hill (1993)    3.610294
5                               Guantanamera (1994)    3.610294
17                          Dangerous Beauty (1998)    3.500000
12                    Run of the Country, The (1995)    3.333333
24                            Broken English (1996)    3.265749
13                 Last Time I Saw Paris, The (1954)    3.200000
16                           Golden Earrings (1947)    3.142978
22                        Nina Takes a Lover (1994)    3.000000
8                            Man of the Year (1995)    3.000000
0                             Ed's Next Move (1996)    2.280926
28  Flower of My Secret, The (Flor de mi secreto, ...    1.665010
25                            Grosse Fatigue (1994)    1.122032
```

As you will have noticed, the two lists are sorted in different ways. The second list is organized according to the predictions made for the user.

Note: When making predictions for user $u$ on item $i$, user $u$ may not have rated any of the $k$ most similar items to i. In this case, we consider the mean rating of $u$ as the predicted value.

**Evaluation with Mean Absolute Error**

[28]:
```python
from recsys.preprocessing import train_test_split, get_examples

# get examples as tuples of userids and itemids and labels from normalize ratings
raw_examples, raw_labels = get_examples(ratings, labels_column='rating')

# train test split
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)

def evaluate(x_test, y_test):
    print('Evaluate the model on {} test data ...'.format(x_test.shape[0]))
    preds = list(predict(u,i) for (u,i) in x_test)
    mae = np.sum(np.absolute(y_test - np.array(preds))) / x_test.shape[0]
    print('\nMAE :', mae)
    return mae
```

[29]:
```python
evaluate(x_test, y_test)
```

```
Evaluate the model on 10000 test data ...

MAE : 0.672389703640273
```

[29]: 0.672389703640273

According to the result, the MAE is < 1.0, which is quite okay

# 5 Summary

For Convenience, Item-based Collaborative Filtering can be used by calling ItemToItem class from *recsys* On how to use that class, the process is described as below

**Evaluation on the ML-100K dataset**

[30]:
```python
from recsys.memories.ItemToItem import ItemToItem
from recsys.preprocessing import ids_encoder, train_test_split, get_examples
from recsys.datasets import ml100k

# load data
ratings, movies = ml100k.load()

# prepare data
ratings, uencoder, iencoder = ids_encoder(ratings)

# get examples as tuples of userids and itemids and labels from normalize ratings
raw_examples, raw_labels = get_examples(ratings, labels_column='rating')

# train test split
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)
```

Instantiate the ItemToItem Collaborative Filtering

Parameters : - `k` : number of neighbors to consider for each item - `metric` : metric to use when computing similarities : let's use **cosine** - `dataset_name` : in this example, we use the ml100k dataset

[31]:
```python
# create the Item-based CF
item2item = ItemToItem(ratings, movies, k=20, metric='cosine',␣
↪dataset_name='ml100k')
```

```
Normalize ratings ...
Create the similarity model ...
Compute nearest neighbors ...
Item to item recommendation model created with success ...
```

[32]:
```python
# evaluate the algorithm on test dataset
item2item.evaluate(x_test, y_test)
```

```
Evaluate the model on 10000 test data ...

MAE : 0.507794195659005
```

[32]: `0.507794195659005`

**Evaluation on the ML-1M dataset**

[33]:
```python
from recsys.memories.ItemToItem import ItemToItem
from recsys.preprocessing import ids_encoder, train_test_split, get_examples
from recsys.datasets import ml1m

# load data
ratings, movies = ml1m.load()

# prepare data
ratings, uencoder, iencoder = ids_encoder(ratings)

# get examples as tuples of userids and itemids and labels from normalize ratings
raw_examples, raw_labels = get_examples(ratings, labels_column='rating')

# train test split
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)

# create the Item-based CF
item2item = ItemToItem(ratings, movies, k=20, metric='cosine',␣
↪dataset_name='ml1m')

# evaluate the algorithm on test dataset
print("=========================")
item2item.evaluate(x_test, y_test)
```

```
Download data 100.1%
Successfully downloaded ml-1m.zip 5917549 bytes.
Unzipping the ml-1m.zip zip file ...
Normalize ratings ...
Create the similarity model ...
Compute nearest neighbors ...
Item to item recommendation model created with success ...
=========================
Evaluate the model on 100021 test data ...

MAE : 0.42514728655396045
```

[33]: 0.42514728655396045

## 6 Advantages over user-based CF

1. Stability : Items ratings are more stable than users ratings. New ratings on items are unlikely to significantly change the similarity between two items, particularly when the items have many ratings (Michael D. Ekstrand, et al. 2011).
2. Scalability : with stable item's ratings, it is reasonable to pre-compute similarities between items in an item-item similarity matrix (similarity between items can be computed offline). This will reduce the scalability concern of the algorithm. (Sarwar et al. 2001), (Michael D.

Ekstrand, et al. 2011).

# 7 References

1. George Karypis (2001)Evaluation of Item-Based Top-N Recommendation Algorithms
2. Sarwar et al. (2001) Item-based collaborative filtering recommendation algorithms
3. Michael D. Ekstrand, et al. (2011). Collaborative Filtering Recommender Systems
4. J. Bobadilla et al. (2013) Recommender systems survey
5. Greg Linden, Brent Smith, and Jeremy York (2003) Amazon.com Recommendations : Item-to-Item Collaborative Filtering

# 8 Author

Carmel WENGA, PhD student at Université de la Polynésie Française, Applied Machine Learning Research Engineer, ShoppingList, NzhinuSoft.

# Chapter 4 - SVD in Collaborative Filtering

Before we start, we have to make sure that we already have our dependencies, that is 'recsys' folder

[1]:
```
import os
#Check if we already have the 'recsys' folder
if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    # If not then download directly from the source
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/
↪master/recsys.zip
    !unzip recsys.zip
```

```
--2023-01-03 21:06:24--  https://github.com/nzhinusoftcm/review-on-
collaborative-filtering/raw/master/recsys.zip
Resolving github.com (github.com)... 192.30.255.112
Connecting to github.com (github.com)|192.30.255.112|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/nzhinusoftcm/review-on-
collaborative-filtering/master/recsys.zip [following]
--2023-01-03 21:06:25--  https://raw.githubusercontent.com/nzhinusoftcm/review-
on-collaborative-filtering/master/recsys.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.110.133, 185.199.109.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 15312323 (15M) [application/zip]
Saving to: 'recsys.zip'

recsys.zip          100%[===================>]  14.60M  --.-KB/s    in 0.1s

2023-01-03 21:06:25 (152 MB/s) - 'recsys.zip' saved [15312323/15312323]

Archive:  recsys.zip
   creating: recsys/
  inflating: recsys/datasets.py
  inflating: recsys/preprocessing.py
  inflating: recsys/utils.py
  inflating: recsys/requirements.txt
```

```
 creating: recsys/.vscode/
inflating: recsys/.vscode/settings.json
 creating: recsys/__pycache__/
inflating: recsys/__pycache__/datasets.cpython-36.pyc
inflating: recsys/__pycache__/datasets.cpython-37.pyc
inflating: recsys/__pycache__/utils.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-37.pyc
inflating: recsys/__pycache__/datasets.cpython-38.pyc
inflating: recsys/__pycache__/preprocessing.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-38.pyc
 creating: recsys/memories/
inflating: recsys/memories/ItemToItem.py
inflating: recsys/memories/UserToUser.py
 creating: recsys/memories/__pycache__/
inflating: recsys/memories/__pycache__/UserToUser.cpython-36.pyc
inflating: recsys/memories/__pycache__/UserToUser.cpython-37.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-37.pyc
inflating: recsys/memories/__pycache__/user2user.cpython-36.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-36.pyc
 creating: recsys/models/
inflating: recsys/models/SVD.py
inflating: recsys/models/MatrixFactorization.py
inflating: recsys/models/ExplainableMF.py
inflating: recsys/models/NonnegativeMF.py
 creating: recsys/models/__pycache__/
inflating: recsys/models/__pycache__/SVD.cpython-36.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-37.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-36.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-37.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-36.pyc
 creating: recsys/metrics/
inflating: recsys/metrics/EvaluationMetrics.py
 creating: recsys/img/
inflating: recsys/img/MF-and-NNMF.png
inflating: recsys/img/svd.png
inflating: recsys/img/MF.png
 creating: recsys/predictions/
 creating: recsys/predictions/item2item/
 creating: recsys/weights/
 creating: recsys/weights/item2item/
 creating: recsys/weights/item2item/ml1m/
inflating: recsys/weights/item2item/ml1m/similarities.npy
inflating: recsys/weights/item2item/ml1m/neighbors.npy
 creating: recsys/weights/item2item/ml100k/
inflating: recsys/weights/item2item/ml100k/similarities.npy
inflating: recsys/weights/item2item/ml100k/neighbors.npy
```

# 1 Requirements

Other than the 'recsys' folder, we also have to make sure that the other required libs have already been installed

```
matplotlib==3.2.2
numpy==1.19.2
pandas==1.0.5
python==3.7
scikit-learn==0.24.1
scikit-surprise==1.1.1
scipy==1.6.2
```

(If we use Google Colab, most of these libs are already installed and up-to-date, except for *scikit-surprise* which is not pre-installed by Google Colab)

To install scikit-surprise on Google Colab, we must execute the code below

```
!pip install surprise
```

But this notebook doesn't require this library yet, so we're not going to install scikit-surprise at the moment

Import all of the required libs

[2]:
```
from recsys.datasets import mlLatestSmall, ml100k, ml1m
from sklearn.preprocessing import LabelEncoder
from scipy.sparse import csr_matrix

import pandas as pd
import numpy as np
import os
```

# 2 Load Movielens Data

This time we'll be using Movielens Latest Small Data, which is similar data that we used on Chapter 1

[3]:
```
ratings, movies = mlLatestSmall.load()
```

```
Download data 100.5%
Successfully downloaded ml-latest-small.zip 978202 bytes.
Unzipping the ml-latest-small.zip zip file ...
```

Now that the initial configuration is done, let's implement the SVD-based collaborative filtering algorithm

# 3 SVD-based Collaborative Filtering

Due to the high level sparsity of the rating matrix $R$, **user-based** and **item-based** collaborative filtering suffer from **data sparsity** and **scalability**. These cause user and item-based collaborative filtering to be less effective and highly affect their performences.

To address the high level sparsity problem, Sarwar et al. (2000) proposed to reduce the dimensionality of the rating $R$ using the *Singular Value Decomposition (SVD)* algorithm.

---

### 3.1 How do SVD works ?

As described is Figure 1, SVD factors the rating matrix $R$ of size $m \times n$ into three matrices $P$, $\Sigma$ and $Q$ as follows :

$$R = P \Sigma Q^{\top}. \tag{1}$$

Here, $P$ and $Q$ are two orthogonal matrices of size $m \times \hat{k}$ and $n \times \hat{k}$ respectively and $\Sigma$ is a diagonal matrix of size $\$ \times \hat{k}\$(with\$\hat{k}\$therankofmatrixR)$ having all singular values of the rating matrix R as its diagonal entries (Billsus and Pazzani, 1998, Sarwar et al. (2000)).

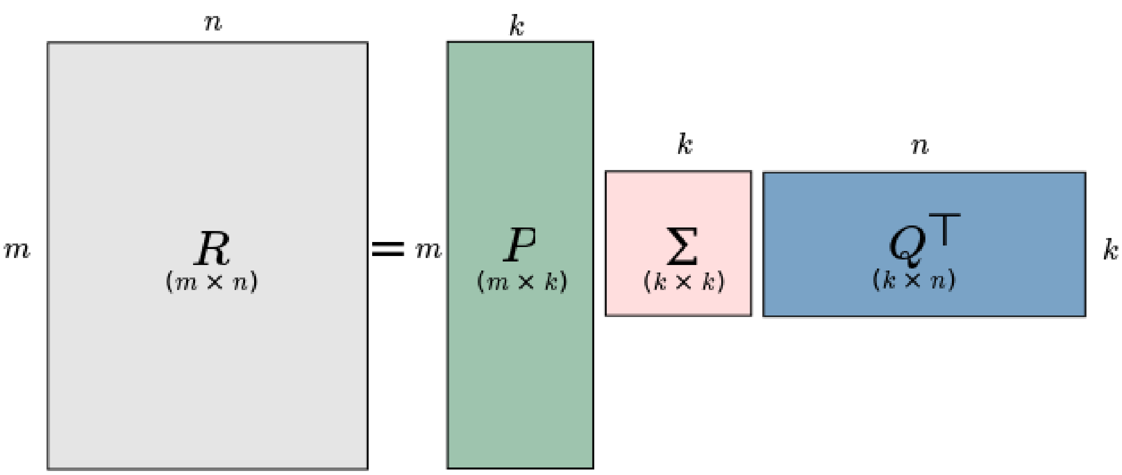



Figure 1 : Singular value decomposition of rating matrix $R$

After having choosen $k$, the dimension of factors that will represent users and items, we can truncate matrix $\Sigma$ by only retaining its $k$ largest singular values to yield $\Sigma_k$ and reduce matrices $P$ and $Q$ accordingly to obtain $P_k$ and $Q_k$. The rating matrix will then be estimated as

$$R_k = P_k \Sigma_k Q_k^{\top}. \tag{2}$$

Once these matrices are known, they can be used for rating predictions ant top-N recommendations. $P_k \Sigma_k^{\frac{1}{2}}$ represents the latent space of users and $\Sigma_k^{\frac{1}{2}} Q_k^{\top}$ the latent space of items. Rating prediction for user $u$ on $i$ is done by the following formular

$$\hat{R}_{u,i} = \left[ P_k \Sigma_k^{\frac{1}{2}} \right]_u \left[ \Sigma_k^{\frac{1}{2}} Q_k^{\top} \right]_i . \tag{3}$$

Before applying SVD, its important to fill in missing values of the rating matrix $R$. Sarwar et al. (2000) found the item's mean rating to be useful default values. The user's average rating can also be used but the former shown better performances. Adding ratings normalization by subtracting the user mean rating or other baseline predictor can improve accuracy.

---

### 3.2 SVD algorithm

1. Factor the normalize rating matrix $R_{norm}$ to obtain matrices $P$, $\Sigma$ and $Q$
2. Reduce $\Sigma$ to dimension $k$ to obtain $\Sigma_k$
3. Compute the square-root of $\Sigma_k$ to obtain $\Sigma_k^{\frac{1}{2}}$
4. Compute the resultant matrices $P_k\Sigma_k^{\frac{1}{2}}$ and $\Sigma_k^{\frac{1}{2}}Q_k^\top$ that will be used to compute recommendation scores for any user and items.

---

### 3.3 Implementation details

SVD can easily be implemented using python library such as `numpy`, `scipy` or `sklearn`. As described by Andrew Ng in his Machine Learning course, it's not recommended to implement the standard SVD by ourselves. Instead, we can take advantage of matrix libraries (such as those listed before) that are optimized for matrix computations and vectorization.

Now let's implement the SVD collaborative filtering

For starter, let's see how our rating matrix looks like

[4]:
```
pd.crosstab(ratings.userid, ratings.itemid, ratings.rating, aggfunc=sum)
```

[4]:
```
itemid  1       2       3       4       5       6       7       8       \
userid
1          4.0     NaN     4.0     NaN     NaN     4.0     NaN     NaN
2          NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
3          NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
4          NaN     NaN     NaN     NaN     NaN     NaN     NaN     NaN
5          4.0     NaN     NaN     NaN     NaN     NaN     NaN     NaN
...        ...     ...     ...     ...     ...     ...     ...     ...
606        2.5     NaN     NaN     NaN     NaN     NaN     2.5     NaN
607        4.0     NaN     NaN     NaN     NaN     NaN     NaN     NaN
608        2.5     2.0     2.0     NaN     NaN     NaN     NaN     NaN
609        3.0     NaN     NaN     NaN     NaN     NaN     NaN     NaN
610        5.0     NaN     NaN     NaN     NaN     5.0     NaN     NaN

itemid  9       10      ...  193565  193567  193571  193573  193579  193581  \
userid                  ...
1          NaN     NaN  ...     NaN     NaN     NaN     NaN     NaN     NaN
2          NaN     NaN  ...     NaN     NaN     NaN     NaN     NaN     NaN
3          NaN     NaN  ...     NaN     NaN     NaN     NaN     NaN     NaN
4          NaN     NaN  ...     NaN     NaN     NaN     NaN     NaN     NaN
5          NaN     NaN  ...     NaN     NaN     NaN     NaN     NaN     NaN
```

```
...         ...    ...   ...     ...    ...    ...    ...    ...    ...
606         NaN    NaN   ...     NaN    NaN    NaN    NaN    NaN    NaN
607         NaN    NaN   ...     NaN    NaN    NaN    NaN    NaN    NaN
608         NaN    4.0   ...     NaN    NaN    NaN    NaN    NaN    NaN
609         NaN    4.0   ...     NaN    NaN    NaN    NaN    NaN    NaN
610         NaN    NaN   ...     NaN    NaN    NaN    NaN    NaN    NaN

itemid  193583  193585  193587  193609
userid
1          NaN     NaN     NaN     NaN
2          NaN     NaN     NaN     NaN
3          NaN     NaN     NaN     NaN
4          NaN     NaN     NaN     NaN
5          NaN     NaN     NaN     NaN
...        ...     ...     ...     ...
606        NaN     NaN     NaN     NaN
607        NaN     NaN     NaN     NaN
608        NaN     NaN     NaN     NaN
609        NaN     NaN     NaN     NaN
610        NaN     NaN     NaN     NaN

[610 rows x 9724 columns]
```

We can observe that our rating matrix has many of unobserved value. However, as we described earlier, the SVD algorithm requires that all inputs in the matrix must be defined. Let's initialize the unobserved ratings with item's average that led to better performances compared to the user's average or even a null initialization (Sarwar et al. (2000)).

We can go further and subtract from each rating the corresponding user mean to normalize the data. This helps to improve the accuracy of the model.

[5]:
```python
# get user's mean rating
umean = ratings.groupby(by='userid')['rating'].mean()
```

[6]:
```python
def rating_matrix(ratings):
    """
    1. Fill NaN values with item's average ratings
    2. Normalize ratings by subtracting user's mean ratings

    :param ratings : DataFrame of ratings data
    :return
        - R : Numpy array of normalized ratings
        - df : DataFrame of normalized ratings
    """

    # fill missing values with item's average ratings
    df = pd.crosstab(ratings.userid, ratings.itemid, ratings.rating, aggfunc=sum)
    df = df.fillna(df.mean(axis=0))
```

```
    # subtract user's mean ratings to normalize data
    df = df.subtract(umean, axis=0)

    # convert our dataframe to numpy array
    R = df.to_numpy()

    return R, df

# generate rating matrix by calling function rating_matrix
R, df = rating_matrix(ratings)
```

*R* is our final rating matrix. This is how the final rating matrix looks like

[7]:
```
df
```

[7]:
```
itemid    1         2         3         4         5         6         7         \
userid
1        -0.366379 -0.934561 -0.366379 -2.009236 -1.294951 -0.366379 -1.181194
2        -0.027346 -0.516458 -0.688660 -1.591133 -0.876847 -0.002197 -0.763091
3         1.485033  0.995921  0.823718 -0.078755  0.635531  1.510181  0.749288
4         0.365375 -0.123737 -0.295940 -1.198413 -0.484127  0.390523 -0.370370
5         0.363636 -0.204545 -0.376748 -1.279221 -0.564935  0.309715 -0.451178
...            ...       ...       ...       ...       ...       ...       ...
606      -1.157399 -0.225581 -0.397784 -1.300256 -0.585971  0.288679 -1.157399
607       0.213904 -0.354278 -0.526481 -1.428953 -0.714668  0.159982 -0.600911
608      -0.634176 -1.134176 -1.134176 -0.777033 -0.062747  0.811903  0.051009
609      -0.270270  0.161548 -0.010655 -0.913127 -0.198842  0.675808 -0.085085
610       1.311444 -0.256738 -0.428941 -1.331413 -0.617127  1.311444 -0.503371

itemid    8         9         10        ...    193565    193567    193571  \
userid                                  ...
1        -1.491379 -1.241379 -0.870167  ... -0.866379 -1.366379 -0.366379
2        -1.073276 -0.823276 -0.452064  ... -0.448276 -0.948276  0.051724
3         0.439103  0.689103  1.060315  ...  1.064103  0.564103  1.564103
4        -0.680556 -0.430556 -0.059343  ... -0.055556 -0.555556  0.444444
5        -0.761364 -0.511364 -0.140152  ... -0.136364 -0.636364  0.363636
...            ...       ...       ...  ...       ...       ...       ...
606      -0.782399 -0.532399 -0.161187  ... -0.157399 -0.657399  0.342601
607      -0.911096 -0.661096 -0.289884  ... -0.286096 -0.786096  0.213904
608      -0.259176 -0.009176  0.865824  ...  0.365824 -0.134176  0.865824
609      -0.395270 -0.145270  0.729730  ...  0.229730 -0.270270  0.729730
610      -0.813556 -0.563556 -0.192344  ... -0.188556 -0.688556  0.311444

itemid    193573    193579    193581    193583    193585    193587    193609
userid
1        -0.366379 -0.866379 -0.366379 -0.866379 -0.866379 -0.866379 -0.366379
2         0.051724 -0.448276  0.051724 -0.448276 -0.448276 -0.448276  0.051724
```

```
3        1.564103  1.064103  1.564103  1.064103  1.064103  1.064103  1.564103
4        0.444444 -0.055556  0.444444 -0.055556 -0.055556 -0.055556  0.444444
5        0.363636 -0.136364  0.363636 -0.136364 -0.136364 -0.136364  0.363636
...           ...       ...       ...       ...       ...       ...       ...
606      0.342601 -0.157399  0.342601 -0.157399 -0.157399 -0.157399  0.342601
607      0.213904 -0.286096  0.213904 -0.286096 -0.286096 -0.286096  0.213904
608      0.865824  0.365824  0.865824  0.365824  0.365824  0.365824  0.865824
609      0.729730  0.229730  0.729730  0.229730  0.229730  0.229730  0.729730
610      0.311444 -0.188556  0.311444 -0.188556 -0.188556 -0.188556  0.311444

[610 rows x 9724 columns]
```

## 4 Encoding of *userids* and *itemids*

All *userid* and *itemid* in *ratings* could have a non-consecutive sequence when ids are ordered, For convenience at the construction of the matrix, encode each of those to a consecutive sequence when ids are ordered through LabelEncoder

Let's encode users and items ids such that their values range from 0 to 609 (for users) and from 0 to 9723 (for items)

[8]:
```python
users = sorted(ratings['userid'].unique())
items = sorted(ratings['itemid'].unique())

# create our id encoders
uencoder = LabelEncoder()
iencoder = LabelEncoder()

# fit our label encoder
uencoder.fit(users)
iencoder.fit(items)
```

[8]:
```
LabelEncoder()
```

## 5 SVD Algorithm

Now that our rating data has been normalize and that missing values has been filled, we can apply the SVD algorithm. Several libraries may be useful such as `numpy`, `scipy`, `sklearn`, … Let's try it with `numpy`.

In our SVD class we provide the following function :

1. `fit()` : compute the svd of the rating matrix and save the resultant matrices P, S and Qh (Q transpose) as attributs of the SVD class.
2. `predict()`: use matrices P, S and Qh to make rating prediction for a given $u$ user on an item $i$. Computations are made over encoded values of userid and itemid. The predicted value is the dot product between $u^{th}$ row of $P.\sqrt{S}$ and the $i^{th}$ column of $\sqrt{S}.Qh$. **Note** that since we normalized rating before applying SVD, the predicted value will also be normalize. So, to

get the final predicted rating, we have to add to the predicted value the mean rating of user $u$.

3. `recommend()`: use matrices P, S and Qh to make recommendations to a given user. The recommended items are those that where not rated by the user and received a high score according to the svd model.

[9]:
```python
class SVD:

    def __init__(self, umeam):
        """
        :param
            - umean : mean ratings of users
        """
        self.umean = umean.to_numpy()

        # init svd resultant matrices
        self.P = np.array([])
        self.S = np.array([])
        self.Qh = np.array([])

        # init users and items latent factors
        self.u_factors = np.array([])
        self.i_factors = np.array([])

    def fit(self, R):
        """
        Fit the SVD model with rating matrix R
        """
        P, s, Qh = np.linalg.svd(R, full_matrices=False)

        self.P = P
        self.S = np.diag(s)
        self.Qh = Qh

        # latent factors of users (u_factors) and items (i_factors)
        self.u_factors = np.dot(self.P, np.sqrt(self.S))
        self.i_factors = np.dot(np.sqrt(self.S), self.Qh)

    def predict(self, userid, itemid):
        """
        Make rating prediction for a given user on an item

        :param
            - userid : user's id
            - itemid : item's id

        :return
```

```
            - r_hat : predicted rating
        """
        # encode user and item ids
        u = uencoder.transform([userid])[0]
        i = iencoder.transform([itemid])[0]

        # the predicted rating is the dot product between the uth row
        # of u_factors and the ith column of i_factors
        r_hat = np.dot(self.u_factors[u,:], self.i_factors[:,i])

        # add the mean rating of user u to the predicted value
        r_hat += self.umean[u]

        return r_hat


    def recommend(self, userid):
        """
        :param
            - userid : user's id
        """
        # encode user
        u = uencoder.transform([userid])[0]

        # the dot product between the uth row of u_factors and i_factors returns
        # the predicted value for user u on all items
        predictions = np.dot(self.u_factors[u,:], self.i_factors) + self.umean[u]

        # sort item ids in decreasing order of predictions
        top_idx = np.flip(np.argsort(predictions))

        # decode indices to get their corresponding itemids
        top_items = iencoder.inverse_transform(top_idx)

        # sorted predictions
        preds = predictions[top_idx]

        return top_items, preds
```

Now let's create our SVD model and provide to it user's mean rating Fit the model with the normalized rating matrix $R$.

[10]:
```
# create our svd model
svd = SVD(umean)

# fit our model with normalized ratings
```

```
svd.fit(R)
```

## 6 Rating prediction

That our model has been fitted, let's make some predictions for users using function `predict` of our SVD class. Here are some truth ratings

[11]:
```
ratings.head(10)
```

[11]:

| | userid | itemid | rating | timestamp |
|---|---|---|---|---|
| 0 | 1 | 1 | 4.0 | 964982703 |
| 1 | 1 | 3 | 4.0 | 964981247 |
| 2 | 1 | 6 | 4.0 | 964982224 |
| 3 | 1 | 47 | 5.0 | 964983815 |
| 4 | 1 | 50 | 5.0 | 964982931 |
| 5 | 1 | 70 | 3.0 | 964982400 |
| 6 | 1 | 101 | 5.0 | 964980868 |
| 7 | 1 | 110 | 4.0 | 964982176 |
| 8 | 1 | 151 | 5.0 | 964984041 |
| 9 | 1 | 157 | 5.0 | 964984100 |

Let's apply our model to make see if our predictions make sense. We will make predictions for user 1 on the 10 items listed above.

[12]:
```
# user for which we make predictions
userid = 1

# list of items for which we are making predictions for user 1
items = [1,3,6,47,50,70,101,110,151,157]

# predictions
for itemid in items:
    r = svd.predict(userid=userid, itemid=itemid)
    print('prediction for userid={} and itemid={} : {}'.format(userid, itemid,␣
 ↪r))
```

```
prediction for userid=1 and itemid=1 : 3.9999999999999996
prediction for userid=1 and itemid=3 : 4.0000000000000036
prediction for userid=1 and itemid=6 : 3.9999999999999867
prediction for userid=1 and itemid=47 : 5.0
prediction for userid=1 and itemid=50 : 4.9999999999999964
prediction for userid=1 and itemid=70 : 2.999999999999981
prediction for userid=1 and itemid=101 : 5.000000000000006
prediction for userid=1 and itemid=110 : 3.9999999999999862
prediction for userid=1 and itemid=151 : 5.0000000000000115
prediction for userid=1 and itemid=157 : 5.000000000000028
```

Our prediction error is less than 0.00001

## 7 Top-N recommendations

The `recommend` function makes recommendations for a given user.

[13]:
```python
# For example, we want to recommend user with id = 1
userid = 1

# items sorted in decreasing order of predictions for user with id = 1
sorted_items, preds = svd.recommend(userid=userid)

##
# Now let's exclude from that sorted list items already purchased by the user
##

# list of items rated by the user
uitems = ratings.loc[ratings.userid == userid].itemid.to_list()

# remove from sorted_items items already in uitems and pick the top 30 ones
# as recommendation list
top30 = np.setdiff1d(sorted_items, uitems, assume_unique=True)[:30]

# get corresponding predictions from the top30 items
top30_idx = list(np.where(sorted_items == idx)[0][0] for idx in top30)
top30_predictions = preds[top30_idx]

# find corresponding movie titles
zipped_top30 = list(zip(top30,top30_predictions))
top30 = pd.DataFrame(zipped_top30, columns=['itemid','predictions'])
List = pd.merge(top30, movies, on='itemid', how='inner')

# show the list
List
```

[13]:
```
    itemid  predictions                                              title  \
0      148          5.0                   Awfully Big Adventure, An (1995)
1     6086          5.0                                 I, the Jury (1982)
2   136445          5.0                  George Carlin: Back in Town (1996)
3     6201          5.0                                  Lady Jane (1986)
4     2075          5.0                                   Mephisto (1981)
5     6192          5.0            Open Hearts (Elsker dig for evigt) (2002)
6   117531          5.0                                  Watermark (2014)
7   158398          5.0                             World of Glory (1991)
8     6021          5.0    American Friend, The (Amerikanische Freund, De...
9   136556          5.0          Kung Fu Panda: Secrets of the Masters (2011)
10  136447          5.0           George Carlin: You Are All Diseased (1999)
11  136503          5.0             Tom and Jerry: Shiver Me Whiskers (2006)
12  134095          5.0                                    My Love (2006)
13    3851          5.0                     I'm the One That I Want (2000)
```

```
14  136469           5.0           Larry David: Curb Your Enthusiasm (1999)
15  158882           5.0                                    All Yours (2016)
16  134004           5.0                                 What Love Is (2007)
17   67618           5.0                             Strictly Sexual (2008)
18    3567           5.0                                  Bossa Nova (2000)
19  158027           5.0                   SORI: Voice from the Heart (2016)
20   59814           5.0                                  Ex Drummer (2007)
21    5745           5.0                           Four Seasons, The (1981)
22  118894           5.0                 Scooby-Doo! Abracadabra-Doo (2010)
23    5746           5.0                   Galaxy of Terror (Quest) (1981)
24  118834           5.0                   National Lampoon's Bag Boy (2007)
25    3940           5.0                   Slumber Party Massacre III (1990)
26   95311           5.0                                      Presto (2008)
27    3496           5.0                            Madame Sousatzka (1988)
28  156025           5.0              Ice Age: The Great Egg-Scapade (2016)
29    2196           5.0                                   Knock Off (1998)

                              genres
0                              Drama
1               Crime|Drama|Thriller
2                             Comedy
3                      Drama|Romance
4                          Drama|War
5                            Romance
6                        Documentary
7                             Comedy
8       Crime|Drama|Mystery|Thriller
9                 Animation|Children
10                            Comedy
11         Animation|Children|Comedy
12                   Animation|Drama
13                            Comedy
14                            Comedy
15              Comedy|Drama|Romance
16                    Comedy|Romance
17              Comedy|Drama|Romance
18              Comedy|Drama|Romance
19                      Drama|Sci-Fi
20         Comedy|Crime|Drama|Horror
21                      Comedy|Drama
22        Animation|Children|Mystery
23      Action|Horror|Mystery|Sci-Fi
24                            Comedy
25                            Horror
26  Animation|Children|Comedy|Fantasy
27                             Drama
28  Adventure|Animation|Children|Comedy
```

```
29                                              Action
```

These 30 items will be the movie recommendations for the user with id = 1 as these movies are predicted to be high rated by that user

# 8 Improving memory based collaborative filtering

SVD can be applied to improve user and item-based collaborative filtering. Instead of computing similarities between user's or item's ratings, we can represent users and items by their corresponding latent factors extracted from the SVD algorithm.

# 9 Reference


1. Daniel Billsus and Michael J. Pazzani (1998). Learning Collaborative Information Filters
2. Sarwar et al. (2000). Application of Dimensionality Reduction in Recommender System – A Case Study


# 10 Author


Carmel WENGA, PhD student at Université de la Polynésie Française, Applied Machine Learning Research Engineer, ShoppingList, NzhinuSoft.

# Chapter 5 - Matrix Factorization in Collaborative Filtering

Before we start, we have to make sure that we already have our dependencies, that is 'recsys' folder

[1]:
```
import os
#Check if we already have the 'recsys' folder
if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    # If not then download directly from the source
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/
 ↪master/recsys.zip
    !unzip recsys.zip
```

```
--2023-01-03 21:07:19--  https://github.com/nzhinusoftcm/review-on-
collaborative-filtering/raw/master/recsys.zip
Resolving github.com (github.com)... 140.82.121.3
Connecting to github.com (github.com)|140.82.121.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/nzhinusoftcm/review-on-
collaborative-filtering/master/recsys.zip [following]
--2023-01-03 21:07:19--  https://raw.githubusercontent.com/nzhinusoftcm/review-
on-collaborative-filtering/master/recsys.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 15312323 (15M) [application/zip]
Saving to: ‘recsys.zip’

recsys.zip          100%[===================>]  14.60M  --.-KB/s    in 0.06s

2023-01-03 21:07:21 (252 MB/s) - ‘recsys.zip’ saved [15312323/15312323]

Archive:  recsys.zip
   creating: recsys/
  inflating: recsys/datasets.py
  inflating: recsys/preprocessing.py
  inflating: recsys/utils.py
  inflating: recsys/requirements.txt
```

```
 creating: recsys/.vscode/
inflating: recsys/.vscode/settings.json
 creating: recsys/__pycache__/
inflating: recsys/__pycache__/datasets.cpython-36.pyc
inflating: recsys/__pycache__/datasets.cpython-37.pyc
inflating: recsys/__pycache__/utils.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-37.pyc
inflating: recsys/__pycache__/datasets.cpython-38.pyc
inflating: recsys/__pycache__/preprocessing.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-38.pyc
 creating: recsys/memories/
inflating: recsys/memories/ItemToItem.py
inflating: recsys/memories/UserToUser.py
 creating: recsys/memories/__pycache__/
inflating: recsys/memories/__pycache__/UserToUser.cpython-36.pyc
inflating: recsys/memories/__pycache__/UserToUser.cpython-37.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-37.pyc
inflating: recsys/memories/__pycache__/user2user.cpython-36.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-36.pyc
 creating: recsys/models/
inflating: recsys/models/SVD.py
inflating: recsys/models/MatrixFactorization.py
inflating: recsys/models/ExplainableMF.py
inflating: recsys/models/NonnegativeMF.py
 creating: recsys/models/__pycache__/
inflating: recsys/models/__pycache__/SVD.cpython-36.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-37.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-36.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-37.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-36.pyc
 creating: recsys/metrics/
inflating: recsys/metrics/EvaluationMetrics.py
 creating: recsys/img/
inflating: recsys/img/MF-and-NNMF.png
inflating: recsys/img/svd.png
inflating: recsys/img/MF.png
 creating: recsys/predictions/
 creating: recsys/predictions/item2item/
 creating: recsys/weights/
 creating: recsys/weights/item2item/
 creating: recsys/weights/item2item/ml1m/
inflating: recsys/weights/item2item/ml1m/similarities.npy
inflating: recsys/weights/item2item/ml1m/neighbors.npy
 creating: recsys/weights/item2item/ml100k/
inflating: recsys/weights/item2item/ml100k/similarities.npy
inflating: recsys/weights/item2item/ml100k/neighbors.npy
```

# 1 Requirements

Other than the 'recsys' folder, we also have to make sure that the other required libs have already been installed

```
matplotlib==3.2.2
numpy==1.19.2
pandas==1.0.5
python==3.7
scikit-learn==0.24.1
scikit-surprise==1.1.1
scipy==1.6.2
```

(If we use Google Colab, most of these libs are already installed and up-to-date, except for *scikit-surprise* which is not pre-installed by Google Colab)

To install scikit-surprise on Google Colab, we must execute the code below

```
!pip install surprise
```

But this notebook doesn't require this library yet, so we're not going to install scikit-surprise at the moment

Import all of the required libs

[2]:
```
from recsys.preprocessing import mean_ratings
from recsys.preprocessing import normalized_ratings
from recsys.preprocessing import ids_encoder
from recsys.preprocessing import train_test_split
from recsys.preprocessing import rating_matrix
from recsys.preprocessing import get_examples
from recsys.preprocessing import scale_ratings

from recsys.datasets import ml100k
from recsys.datasets import ml1m

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

import os
```

# 2 Matrix Factorization

The Matrix Factorization algorithm is a variant of SVD. Also known as Regularized SVD, it uses the Gradient Descent optimizer to optimize the cost function while training the model

User-based and Item-based collaborative Filtering recommender systems suffer from data sparsity and scalability for online recommendations. Matrix Factorization helps to address these drawbacks of memory-based collaborative filtering by reducing the dimension of the rating matrix $R$.

The movielen lasted small dataset has 100k ratings of $m = 610$ users on $n = 9724$ items. The rating matrix in then a $m \times n$ matrix (i.e $R \in \mathbb{R}^{m \times n}$). The fact that users usually interact with less than 1% of items leads the rating matrix $R$ to be highly sparse. For example, the degree of sparsity of the movielen lasted small dataset is

$$sparsity = 100 - \frac{\text{total ratings}}{m \times n} = 100 - \frac{100000}{610 \times 9724} = 98,3\% \tag{1}$$

This means that in this dataset, a user has interacted with less than 2% of items. To reduce the dimension of the rating matrix $R$, Matrix Factorization (MF) mappes both users and items to a joint latent factor space of dimensionality $k$ such that user-item interactions are modeled as inner products in that space (Yehuda Koren et al., 2009). MF then decomposes $R$ in two matrices as follows :

$$R = Q^{\top} P \tag{2}$$

Where $P \in \mathbb{R}^{m \times k}$ represents latent factors of users and $Q \in \mathbb{R}^{n \times k}$ is the latent factors of items. Each line of $P$, say $p_u \in \mathbb{R}^k$ denotes the taste of user $u$ and each $q_i \in \mathbb{R}^k$ the features of item $i$. The dot product between $p_u$ and $q_i$ will be the rating prediction of user $u$ on item $i$ :

$$\hat{r}_{u,i} = q_i^{\top} p_u. \tag{3}$$

Figure 1 presents an example of decomposition of $R$ into two matrices $P$ and $Q$.

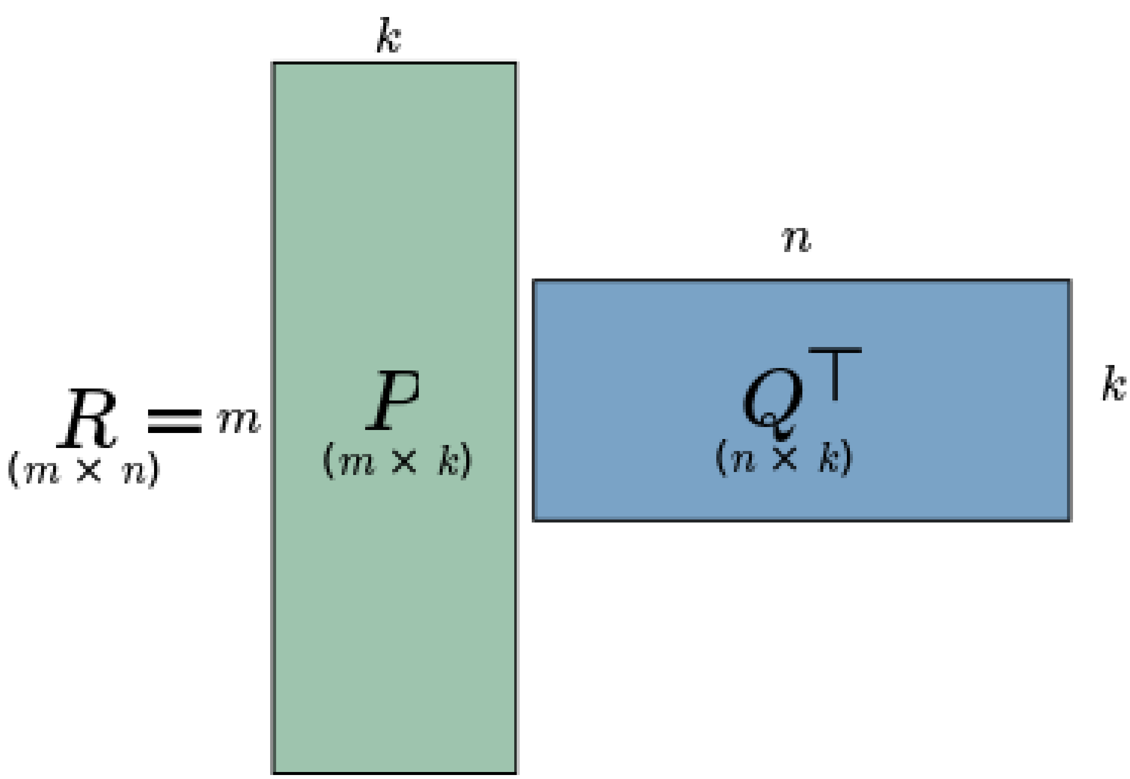



Figure 1: Decomposition of $R$ into $P$ and $Q$

To learn the latent factors $p_u$ and $q_i$, the system minimizes the regularized squared error on the set of known ratings. The cost function $J$ is defined as follows :

$$J = \frac{1}{2} \sum_{(u,i) \in \kappa} (r_{ui} - q_i^\top p_u)^2 + \lambda(||p_u||^2 + ||q_i||^2) \tag{4}$$

where $\kappa$ is the set of $(u, i)$ pairs for which $r_{u,i}$ is known (the training set), and $\lambda$ is the regularizer parameter.

### 2.1 Learning Algorithms

As described in (Yehuda Koren et al., 2009), to minimize the cost function $J$, the matrix factorization algorithm predicts $\hat{r}_{u,i}$ for each given training case (existing $r_{u,i}$), and computes the associated error defined by the Mean Absolute Error (MAE) as :

$$e_{u,i} = |r_{ui} - q_i^\top p_u|. \tag{5}$$

Note : The overall error $E$ is defined as :

$$E = \frac{1}{M} \sum_{(u,i) \in \kappa} e_{u,i} \tag{6}$$

Where $M$ is the number of example. The update rules for parameters $p_u$ and $q_i$ are defined as follows :

$$q_i \leftarrow q_i - \alpha \frac{\partial}{\partial q_i} J_{u,i}, \tag{7}$$

$$p_u \leftarrow p_u - \alpha \frac{\partial}{\partial p_u} J_{u,i} \tag{8}$$

where $\alpha$ is the learning rate and $\frac{\partial}{\partial p_u} J_{u,i}$ is the partial derivative of the cost function $J$ according to $p_u$. It computes the extent to which $p_u$ contributes to the total error.

#### 2.1.1 How to compute $\frac{\partial}{\partial q_i} J_{u,i}$ ?

$$\begin{aligned} \frac{\partial}{\partial q_i} J_{u,i} &= \frac{1}{2} \frac{\partial}{\partial q_i} \left[ (r_{ui} - q_i^\top p_u)^2 + \lambda(||p_u||^2 + ||q_i||^2) \right] && (9) \\ &= -(r_{u,i} - q_i^\top p_u) \cdot p_u + \lambda \cdot q_i && (10) \\ &= -e_{u,i} \cdot p_u + \lambda \cdot q_i && (11) \end{aligned}$$

The update rules are then given by :

$$q_i \leftarrow q_i + \alpha \cdot (e_{u,i} \cdot p_u - \lambda \cdot q_i), \tag{12}$$

$$p_u \leftarrow p_u + \alpha \cdot (e_{u,i} \cdot q_i - \lambda \cdot p_u) \tag{13}$$

### 2.2 Matrix Factorization: The Algorithm

Initialize $P$ and $Q$ with random values

For each training example $(u,i) \in \kappa$ with the corresponding rating $r_{u,i}$ :

compute $\hat{r}_{u,i}$ as $\hat{r}_{u,i} = q_i^\top p_u$

compute the error : $e_{u,i} = |r_{ui} - \hat{r}_{u,i}|$

update $p_u$ and $q_i$:

$p_u \leftarrow p_u + \alpha \cdot (e_{u,i} \cdot q_i - \lambda \cdot p_u)$

$q_i \leftarrow q_i + \alpha \cdot (e_{u,i} \cdot p_u - \lambda \cdot q_i)$

```
    </ul>
<li> Repeat step 2 until the optimal parameters are reached.
```

### 2.3 Model definition

[3]:
```
class MatrixFactorization:

    def __init__(self, m, n, k=10, alpha=0.001, lamb=0.01):
        """
        Initialization of the model
        : param
            - m : number of users
            - n : number of items
            - k : length of latent factor, both for users and items. 50 by
 ↪default
            - alpha : learning rate. 0.001 by default
            - lamb : regularizer parameter. 0.02 by default
        """
        np.random.seed(32)

        # initialize the latent factor matrices P and Q (of shapes (m,k) and
 ↪(n,k) respectively) that will be learnt
        self.k = k
        self.P = np.random.normal(size=(m, k))
        self.Q = np.random.normal(size=(n, k))

        # hyperparameter initialization
        self.alpha = alpha
        self.lamb = lamb

        # training history
        self.history = {
```

```
            "epochs":[],
            "loss":[],
            "val_loss":[],
            "lr":[]
        }

    def print_training_parameters(self):
        print('Training Matrix Factorization Model ...')
        print(f'k={self.k} \t alpha={self.alpha} \t lambda={self.lamb}')

    def update_rule(self, u, i, error):
        self.P[u] = self.P[u] + self.alpha * (error * self.Q[i] - self.lamb *␣
↪self.P[u])
        self.Q[i] = self.Q[i] + self.alpha * (error * self.P[u] - self.lamb *␣
↪self.Q[i])

    def mae(self,  x_train, y_train):
        """
        returns the Mean Absolute Error
        """
        # number of training exemples
        M = x_train.shape[0]
        error = 0
        for pair, r in zip(x_train, y_train):
            u, i = pair
            error += abs(r - np.dot(self.P[u], self.Q[i]))
        return error/M

    def print_training_progress(self, epoch, epochs, error, val_error, steps=5):
        if epoch == 1 or epoch % steps == 0 :
                print("epoch {}/{} - loss : {} - val_loss : {}".format(epoch,␣
↪epochs, round(error,3), round(val_error,3)))

    def learning_rate_schedule(self, epoch, target_epochs = 20):
        if (epoch >= target_epochs) and (epoch % target_epochs == 0):
                factor = epoch // target_epochs
                self.alpha = self.alpha * (1 / (factor * 20))
                print("\nLearning Rate : {}\n".format(self.alpha))

    def fit(self, x_train, y_train, validation_data, epochs=1000):
        """
        Train latent factors P and Q according to the training set

        :param
            - x_train : training pairs (u,i) for which rating r_ui is known
            - y_train : set of ratings r_ui for all training pairs (u,i)
            - validation_data : tuple (x_test, y_test)
```

```
            - epochs : number of time to loop over the entire training set.
            1000 epochs by default

        Note that u and i are encoded values of userid and itemid
        """
        self.print_training_parameters()

        # validation data
        x_test, y_test = validation_data

        # loop over the number of epochs
        for epoch in range(1, epochs+1):

            # for each pair (u,i) and the corresponding rating r
            for pair, r in zip(x_train, y_train):

                # get encoded values of userid and itemid from pair
                u,i = pair

                # compute the predicted rating r_hat
                r_hat = np.dot(self.P[u], self.Q[i])

                # compute the prediction error
                e = abs(r - r_hat)

                # update rules
                self.update_rule(u, i, e)

            # training and validation error  after this epochs
            error = self.mae(x_train, y_train)
            val_error = self.mae(x_test, y_test)

            # update history
            self.history['epochs'].append(epoch)
            self.history['loss'].append(error)
            self.history['val_loss'].append(val_error)

            # update history
            self.update_history(epoch, error, val_error)

            # print training progress after each steps epochs
            self.print_training_progress(epoch, epochs, error, val_error,␣
↪steps=1)

            # leaning rate scheduler : redure the learning rate as we go deeper␣
↪in the number of epochs
            # self.learning_rate_schedule(epoch)
```

```
        return self.history

    def update_history(self, epoch, error, val_error):
        self.history['epochs'].append(epoch)
        self.history['loss'].append(error)
        self.history['val_loss'].append(val_error)
        self.history['lr'].append(self.alpha)

    def evaluate(self, x_test, y_test):
        """
        compute the global error on the test set
        :param x_test : test pairs (u,i) for which rating r_ui is known
        :param y_test : set of ratings r_ui for all test pairs (u,i)
        """
        error = self.mae(x_test, y_test)
        print(f"validation error : {round(error,3)}")

        return error

    def predict(self, userid, itemid):
        """
        Make rating prediction for a user on an item
        :param userid
        :param itemid
        :return r : predicted rating
        """
        # encode user and item ids to be able to access their latent factors in
        # matrices P and Q
        u = uencoder.transform([userid])[0]
        i = iencoder.transform([itemid])[0]

        # rating prediction using encoded ids. Dot product between P_u and Q_i
        r = np.dot(self.P[u], self.Q[i])
        return r

    def recommend(self, userid, N=30):
        """
        make to N recommendations for a given user

        :return(top_items,preds) : top N items with the highest predictions
        with their corresponding predictions
        """
        # encode the userid
        u = uencoder.transform([userid])[0]

        # predictions for users userid on all product
```

```
        predictions = np.dot(self.P[u], self.Q.T)

        # get the indices of the top N predictions
        top_idx = np.flip(np.argsort(predictions))[:N]

        # decode indices to get their corresponding itemids
        top_items = iencoder.inverse_transform(top_idx)

        # take corresponding predictions for top N indices
        preds = predictions[top_idx]

        return top_items, preds
```

Define the number of epoch for training process

```
[4]: epochs = 10
```

# 3 Dataset 1: MovieLens 100k

## 3.1 Evaluation on raw ratings

```
[5]: # load the ml100k dataset
ratings, movies = ml100k.load()

# Encode the userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

m = ratings.userid.nunique()   # total number of users
n = ratings.itemid.nunique()   # total number of items

# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings)

# Split dataset to train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
 ↪labels=raw_labels)
```

```
Download data 100.2%
Successfully downloaded ml-100k.zip 4924029 bytes.
Unzipping the ml-100k.zip zip file ...
```

```
[6]: # Create the model
MF = MatrixFactorization(m, n, k=10, alpha=0.01, lamb=1.5)

# Fit the model on the training set
history = MF.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
 ↪y_test))
```

```
Training Matrix Factorization Model ...
k=10     alpha=0.01      lambda=1.5
epoch 1/10 - loss : 2.734 - val_loss : 2.779
epoch 2/10 - loss : 1.764 - val_loss : 1.794
epoch 3/10 - loss : 1.592 - val_loss : 1.614
epoch 4/10 - loss : 1.538 - val_loss : 1.556
epoch 5/10 - loss : 1.515 - val_loss : 1.531
epoch 6/10 - loss : 1.503 - val_loss : 1.517
epoch 7/10 - loss : 1.496 - val_loss : 1.509
epoch 8/10 - loss : 1.491 - val_loss : 1.504
epoch 9/10 - loss : 1.488 - val_loss : 1.5
epoch 10/10 - loss : 1.486 - val_loss : 1.497
```

[7]:
```
# Evaluate model on test set
MF.evaluate(x_test, y_test)
```

```
validation error : 1.497
```

[7]:
```
1.4973507972141993
```

It shows a pretty bad result as the MAE score is more than 1.0. How about we normalize the ratings first?

### 3.2 Evaluation on normalized ratings

[8]:
```
# Load the ml100k dataset
ratings, movies = ml100k.load()

# Encode the userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

m = ratings['userid'].nunique()   # total number of users
n = ratings['itemid'].nunique()   # total number of items

# Normalize ratings by subtracting means
normalized_column_name = "norm_rating"
ratings = normalized_ratings(ratings, norm_column=normalized_column_name)

# Get examples as tuples of userids and itemids and labels from normalized␣
 ↪ratings
raw_examples, raw_labels = get_examples(ratings,␣
 ↪labels_column=normalized_column_name)

# Split dataset to train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
 ↪labels=raw_labels)
```

[9]:
```
# Create the model
MF = MatrixFactorization(m, n, k=10, alpha=0.01, lamb=1.5)

# Fit the model on the training set
history = MF.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
↪y_test))
```

```
Training Matrix Factorization Model ...
k=10     alpha=0.01      lambda=1.5
epoch 1/10 - loss : 0.851 - val_loss : 0.847
epoch 2/10 - loss : 0.831 - val_loss : 0.831
epoch 3/10 - loss : 0.828 - val_loss : 0.829
epoch 4/10 - loss : 0.827 - val_loss : 0.828
epoch 5/10 - loss : 0.827 - val_loss : 0.828
epoch 6/10 - loss : 0.826 - val_loss : 0.828
epoch 7/10 - loss : 0.826 - val_loss : 0.828
epoch 8/10 - loss : 0.826 - val_loss : 0.828
epoch 9/10 - loss : 0.826 - val_loss : 0.828
epoch 10/10 - loss : 0.826 - val_loss : 0.828
```

[10]:
```
# Evaluate model on test set
MF.evaluate(x_test, y_test)
```

```
validation error : 0.828
```

[10]:
```
0.8276982643684648
```

The result seems better than before as now that the MAE score is less than 1.0

# 4 Dataset 2: MovieLens 1M

## 4.1 Evaluation on raw ratings

[11]:
```
# Load the ml1m dataset
ratings, movies = ml1m.load()

# Encode the userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

m = ratings.userid.nunique()   # total number of users
n = ratings.itemid.nunique()   # total number of items

# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings)

# Split dataset to train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)
```

```
Download data 100.1%
Successfully downloaded ml-1m.zip 5917549 bytes.
Unzipping the ml-1m.zip zip file ...
```

[12]:
```
# Create the model
MF = MatrixFactorization(m, n, k=10, alpha=0.01, lamb=1.5)

# Fit the model on the training set
history = MF.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
↪y_test))
```

```
Training Matrix Factorization Model ...
k=10     alpha=0.01      lambda=1.5
epoch 1/10 - loss : 1.713 - val_loss : 1.718
epoch 2/10 - loss : 1.523 - val_loss : 1.526
epoch 3/10 - loss : 1.496 - val_loss : 1.498
epoch 4/10 - loss : 1.489 - val_loss : 1.489
epoch 5/10 - loss : 1.485 - val_loss : 1.486
epoch 6/10 - loss : 1.484 - val_loss : 1.484
epoch 7/10 - loss : 1.483 - val_loss : 1.483
epoch 8/10 - loss : 1.483 - val_loss : 1.483
epoch 9/10 - loss : 1.482 - val_loss : 1.482
epoch 10/10 - loss : 1.482 - val_loss : 1.482
```

[13]:
```
# Evaluate model on test set
MF.evaluate(x_test, y_test)
```

```
validation error : 1.482
```

[13]: 1.4820034560467208

It shows a pretty bad result as the MAE score is more than 1.0. How about we normalize the ratings first?

### 4.2 Evaluation on normalized ratings

[14]:
```
# Load the ml1m dataset
ratings, movies = ml1m.load()

# Encode the userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

m = ratings['userid'].nunique()    # total number of users
n = ratings['itemid'].nunique()    # total number of items

# Normalize ratings by subtracting means
normalized_column_name = "norm_rating"
ratings = normalized_ratings(ratings, norm_column=normalized_column_name)
```

```
# Get examples as tuples of userids and itemids and labels from normalized␣
↪ratings
raw_examples, raw_labels = get_examples(ratings,␣
↪labels_column=normalized_column_name)

# Split dataset to train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)
```

[15]:
```
# Create the model
MF = MatrixFactorization(m, n, k=10, alpha=0.01, lamb=1.5)

# Fit the model on the training set
history = MF.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
↪y_test))
```

```
Training Matrix Factorization Model ...
k=10     alpha=0.01      lambda=1.5
epoch 1/10 - loss : 0.826 - val_loss : 0.827
epoch 2/10 - loss : 0.824 - val_loss : 0.825
epoch 3/10 - loss : 0.823 - val_loss : 0.825
epoch 4/10 - loss : 0.823 - val_loss : 0.825
epoch 5/10 - loss : 0.823 - val_loss : 0.825
epoch 6/10 - loss : 0.823 - val_loss : 0.825
epoch 7/10 - loss : 0.823 - val_loss : 0.825
epoch 8/10 - loss : 0.823 - val_loss : 0.825
epoch 9/10 - loss : 0.823 - val_loss : 0.825
epoch 10/10 - loss : 0.823 - val_loss : 0.825
```

[16]:
```
# Evaluate model on test set
MF.evaluate(x_test, y_test)
```

```
validation error : 0.825
```

[16]: 0.8250208634455388

The result seems better than before as now that the MAE score is less than 1.0

# 5 Predictions

Now that the latent factors $P$ and $Q$, we can use them to make predictions and recommendations. Let's call the `predict` function of the `Matrix Factorization` class to make prediction for a given.

Rating prediction for user with id = 1 on movie with id = 1 for which the truth rating $r = 5.0$

[17]:
```
ratings.userid = uencoder.inverse_transform(ratings.userid.to_list())
ratings.itemid = uencoder.inverse_transform(ratings.itemid.to_list())
ratings.head(5)
```

[17]:

| | userid | itemid | rating | rating_mean | norm_rating |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 5 | 4.188679 | 0.811321 |
| 1 | 1 | 48 | 5 | 4.188679 | 0.811321 |
| 2 | 1 | 145 | 5 | 4.188679 | 0.811321 |
| 3 | 1 | 254 | 4 | 4.188679 | -0.188679 |
| 4 | 1 | 514 | 5 | 4.188679 | 0.811321 |

[18]:

```
4.188679 + MF.predict(userid=1, itemid=1) # add the mean because we have used␣
↪the normalised ratings for training
```

[18]:

```
4.188679163563357
```

Thus, the predicted rating of user with id = 1 for movie with id = 1 is 4.188679

# 6 Reference


1. Yehuda Koren et al. (2009). Matrix Factorization Techniques for Recommender Systems


# 7 Author


Carmel WENGA, PhD student at Université de la Polynésie Française, Applied Machine Learning Research Engineer, ShoppingList, NzhinuSoft.

# Chapter 6 - Non-Negative Matrix Factorization in Collaborative Filtering

Before we start, we have to make sure that we already have our dependencies, that is 'recsys' folder

[1]:
```
import os
#Check if we already have the 'recsys' folder
if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    # If not then download directly from the source
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/
 ↪master/recsys.zip
    !unzip recsys.zip
```

```
--2023-01-03 21:08:21--  https://github.com/nzhinusoftcm/review-on-
collaborative-filtering/raw/master/recsys.zip
Resolving github.com (github.com)... 140.82.121.3
Connecting to github.com (github.com)|140.82.121.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/nzhinusoftcm/review-on-
collaborative-filtering/master/recsys.zip [following]
--2023-01-03 21:08:21--  https://raw.githubusercontent.com/nzhinusoftcm/review-
on-collaborative-filtering/master/recsys.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 15312323 (15M) [application/zip]
Saving to: 'recsys.zip'

recsys.zip          100%[===================>]  14.60M  --.-KB/s    in 0.06s

2023-01-03 21:08:21 (252 MB/s) - 'recsys.zip' saved [15312323/15312323]

Archive:  recsys.zip
   creating: recsys/
  inflating: recsys/datasets.py
  inflating: recsys/preprocessing.py
```

```
inflating: recsys/utils.py
inflating: recsys/requirements.txt
 creating: recsys/.vscode/
inflating: recsys/.vscode/settings.json
 creating: recsys/__pycache__/
inflating: recsys/__pycache__/datasets.cpython-36.pyc
inflating: recsys/__pycache__/datasets.cpython-37.pyc
inflating: recsys/__pycache__/utils.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-37.pyc
inflating: recsys/__pycache__/datasets.cpython-38.pyc
inflating: recsys/__pycache__/preprocessing.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-38.pyc
 creating: recsys/memories/
inflating: recsys/memories/ItemToItem.py
inflating: recsys/memories/UserToUser.py
 creating: recsys/memories/__pycache__/
inflating: recsys/memories/__pycache__/UserToUser.cpython-36.pyc
inflating: recsys/memories/__pycache__/UserToUser.cpython-37.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-37.pyc
inflating: recsys/memories/__pycache__/user2user.cpython-36.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-36.pyc
 creating: recsys/models/
inflating: recsys/models/SVD.py
inflating: recsys/models/MatrixFactorization.py
inflating: recsys/models/ExplainableMF.py
inflating: recsys/models/NonnegativeMF.py
 creating: recsys/models/__pycache__/
inflating: recsys/models/__pycache__/SVD.cpython-36.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-37.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-36.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-37.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-36.pyc
 creating: recsys/metrics/
inflating: recsys/metrics/EvaluationMetrics.py
 creating: recsys/img/
inflating: recsys/img/MF-and-NNMF.png
inflating: recsys/img/svd.png
inflating: recsys/img/MF.png
 creating: recsys/predictions/
 creating: recsys/predictions/item2item/
 creating: recsys/weights/
 creating: recsys/weights/item2item/
 creating: recsys/weights/item2item/ml1m/
inflating: recsys/weights/item2item/ml1m/similarities.npy
inflating: recsys/weights/item2item/ml1m/neighbors.npy
 creating: recsys/weights/item2item/ml100k/
inflating: recsys/weights/item2item/ml100k/similarities.npy
inflating: recsys/weights/item2item/ml100k/neighbors.npy
```

# 1 Requirements

Other than the 'recsys' folder, we also have to make sure that the other required libs have already been installed

```
matplotlib==3.2.2
numpy==1.19.2
pandas==1.0.5
python==3.7
scikit-learn==0.24.1
scikit-surprise==1.1.1
scipy==1.6.2
```

(If we use Google Colab, most of these libs are already installed and up-to-date, except for *scikit-surprise* which is not pre-installed by Google Colab)

Because *scikit-surprise* is required by current notebook, we're going to install *scikit-surprise* right away

[2]:
```
!pip install surprise
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting surprise
  Downloading surprise-0.1-py2.py3-none-any.whl (1.8 kB)
Collecting scikit-surprise
  Downloading scikit-surprise-1.1.3.tar.gz (771 kB)
      772.0/772.0 KB
25.8 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: joblib>=1.0.0 in /usr/local/lib/python3.8/dist-
packages (from scikit-surprise->surprise) (1.2.0)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.8/dist-
packages (from scikit-surprise->surprise) (1.21.6)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.8/dist-
packages (from scikit-surprise->surprise) (1.7.3)
Building wheels for collected packages: scikit-surprise
  Building wheel for scikit-surprise (setup.py) ... done
  Created wheel for scikit-surprise:
filename=scikit_surprise-1.1.3-cp38-cp38-linux_x86_64.whl size=2626456
sha256=0ce016051fb68798c19e8396c7c3bf405e080ba98c37f6e04646c0c17ae6315a
  Stored in directory: /root/.cache/pip/wheels/af/db/86/2c18183a80ba05da35bf0fb7
417aac5cddbd93bcb1b92fd3ea
Successfully built scikit-surprise
Installing collected packages: scikit-surprise, surprise
Successfully installed scikit-surprise-1.1.3 surprise-0.1
```

Import all of the required libs

[3]:
```python
from recsys.preprocessing import mean_ratings
from recsys.preprocessing import normalized_ratings
from recsys.preprocessing import ids_encoder
from recsys.preprocessing import train_test_split
from recsys.preprocessing import rating_matrix
from recsys.preprocessing import get_examples
from recsys.preprocessing import scale_ratings

from recsys.datasets import ml1m
from recsys.datasets import ml100k

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

import os
```

## 2 Load and Preprocess the Movielens-100K Dataset

[4]:
```python
# Load ml100k dataset
ratings, movies = ml100k.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

# Convert ratings from dataframe to numpy array
np_ratings = ratings.to_numpy()

# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings, labels_column="rating")

# Split the dataset to train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)
```

```
Download data 100.2%
Successfully downloaded ml-100k.zip 4924029 bytes.
Unzipping the ml-100k.zip zip file ...
```

Now that the data is prepared, let's move to the implementation of Non-negative Matrix Factorization

## 3 Non-negative Matrix Factorization for Recommendations

Jusl like Matrix Factorization (MF) (Yehuda Koren et al., 2009), Non-negative Matrix Factorization (NMF in short) factors the rating matrix $R$ in two matrices in such a way that $R = PQ^\top$.

### 3.1 One limitation of Matrix Factorization

$P$ and $Q$ values in MF are non interpretable since their components can take arbitrary (positive and negative) values.

### 3.2 Particulariy of Non-negative Matrix Factorization

NMF (Lee and Seung, 1999) allows the reconstruction of $P$ and $Q$ in such a way that $P, Q \geq 0$. Constraining $P$ and $Q$ values to be taken from $[0, 1]$ allows a probabilistic interpretation

- Latent factors represent groups of users who share the same tastes,
- The value $P_{u,l}$ represents the probability that user $u$ belongs to the group $l$ of users and
- The value $Q_{l,i}$ represents the probability that users in the group $l$ likes item $i$.

### 3.3 Objective function

With the Euclidian distance, the NMF objective function is defined by

$$J = \frac{1}{2} \sum_{(u,i) \in \kappa} ||R_{u,i} - P_u Q_i^\top||^2 + \lambda_P ||P_u||^2 + \lambda_Q ||Q_i||^2 \tag{1}$$

The goal is to minimize the cost function $J$ by optimizing parameters $P$ and $Q$, with $\lambda_P$ and $\lambda_Q$ the regularizer parameters.

### 3.4 Multiplicative update rule

According (Lee and Seung, 1999), to the multiplicative update rule for $P$ and $Q$ are as follows :

$$P \leftarrow P \cdot \frac{RQ}{PQ^\top Q} \tag{2}$$

$$Q \leftarrow Q \cdot \frac{R^\top P}{QP^\top P} \tag{3}$$

However, since $R$ is a sparse matrix, we need to update each $P_u$ according to existing ratings of user $u$. Similarily, we need to update $Q_i$ according to existing ratings on item $i$. Hence :

$$P_{u,k} \leftarrow P_{u,k} \cdot \frac{\sum_{i \in I_u} Q_{i,k} \cdot r_{u,i}}{\sum_{i \in I_u} Q_{i,k} \cdot \hat{r}_{u,i} + \lambda_P |I_u| P_{u,k}} \tag{4}$$

$$Q_{i,k} \leftarrow Q_{i,k} \cdot \frac{\sum_{u \in U_i} P_{u,k} \cdot r_{u,i}}{\sum_{u \in U_i} P_{u,k} \cdot \hat{r}_{u,i} + \lambda_Q |U_i| Q_{i,k}} \tag{5}$$

Where - $P_{u,k}$ is the $k^{th}$ latent factor of $P_u$ - $Q_{i,k}$ is the $k^{th}$ latent factor of $Q_i$ - $I_u$ the of items rated by user $u$ - $U_i$ the set of users who rated item $i$

# 4 Non-negative Matrix Factorization Model

[5]:
```
class NMF:

    def __init__(self, ratings, m, n, uencoder, iencoder, K=10, lambda_P=0.01,␣
 ↪lambda_Q=0.01):

        np.random.seed(32)

        # initialize the latent factor matrices P and Q (of shapes (m,k) and␣
 ↪(n,k) respectively) that will be learnt
        self.ratings = ratings
        self.np_ratings = ratings.to_numpy()
        self.K = K
        self.P = np.random.rand(m, K)
        self.Q = np.random.rand(n, K)

        # hyper parameter initialization
        self.lambda_P = lambda_P
        self.lambda_Q = lambda_Q

        # initialize encoders
        self.uencoder = uencoder
        self.iencoder = iencoder

        # training history
        self.history = {
            "epochs": [],
            "loss": [],
            "val_loss": [],
        }

    def print_training_parameters(self):
        print('Training NMF ...')
        print(f'k={self.K}')

    def mae(self, x_train, y_train):
        """
        returns the Mean Absolute Error
        """
        # number of training examples
        m = x_train.shape[0]
        error = 0
```

```python
        for pair, r in zip(x_train, y_train):
            u, i = pair
            error += abs(r - np.dot(self.P[u], self.Q[i]))
        return error / m

    def update_rule(self, u, i, error):
        I = self.np_ratings[self.np_ratings[:, 0] == u][:, [1, 2]]
        U = self.np_ratings[self.np_ratings[:, 1] == i][:, [0, 2]]

        num = self.P[u] * np.dot(self.Q[I[:, 0]].T, I[:, 1])
        dem = np.dot(self.Q[I[:, 0]].T, np.dot(self.P[u], self.Q[I[:, 0]].T)) + self.lambda_P * len(I) * self.P[u]
        self.P[u] = num / dem

        num = self.Q[i] * np.dot(self.P[U[:, 0]].T, U[:, 1])
        dem = np.dot(self.P[U[:, 0]].T, np.dot(self.P[U[:, 0]], self.Q[i].T)) + self.lambda_Q * len(U) * self.Q[i]
        self.Q[i] = num / dem

    @staticmethod
    def print_training_progress(epoch, epochs, error, val_error, steps=5):
        if epoch == 1 or epoch % steps == 0:
            print(f"epoch {epoch}/{epochs} - loss : {round(error, 3)} - val_loss : {round(val_error, 3)}")

    def fit(self, x_train, y_train, validation_data, epochs=10):

        self.print_training_parameters()
        x_test, y_test = validation_data
        for epoch in range(1, epochs+1):
            for pair, r in zip(x_train, y_train):
                u, i = pair
                r_hat = np.dot(self.P[u], self.Q[i])
                e = abs(r - r_hat)
                self.update_rule(u, i, e)
            # training and validation error  after this epochs
            error = self.mae(x_train, y_train)
            val_error = self.mae(x_test, y_test)
            self.update_history(epoch, error, val_error)
            self.print_training_progress(epoch, epochs, error, val_error, steps=1)

        return self.history

    def update_history(self, epoch, error, val_error):
        self.history['epochs'].append(epoch)
        self.history['loss'].append(error)
```

```
        self.history['val_loss'].append(val_error)

    def evaluate(self, x_test, y_test):
        error = self.mae(x_test, y_test)
        print(f"validation error : {round(error,3)}")
        print('MAE : ', error)
        return error

    def predict(self, userid, itemid):
        u = self.uencoder.transform([userid])[0]
        i = self.iencoder.transform([itemid])[0]
        r = np.dot(self.P[u], self.Q[i])
        return r

    def recommend(self, userid, N=30):
        # encode the userid
        u = self.uencoder.transform([userid])[0]

        # predictions for users userid on all product
        predictions = np.dot(self.P[u], self.Q.T)

        # get the indices of the top N predictions
        top_idx = np.flip(np.argsort(predictions))[:N]

        # decode indices to get their corresponding itemids
        top_items = self.iencoder.inverse_transform(top_idx)

        # take corresponding predictions for top N indices
        preds = predictions[top_idx]

        return top_items, preds
```

## 5 Train the NMF model

Selected model parameters :

- $k = 10$ : (number of factors)
- $\lambda_P = 0.6$
- $\lambda_Q = 0.6$
- epochs = 10

*Note that it may take some time to complete the training on 10 epochs (around 7 minutes).*

[6]:
```
m = ratings['userid'].nunique()   # total number of users
n = ratings['itemid'].nunique()   # total number of items

# Create and train the model using train set
nmf = NMF(ratings, m, n, uencoder, iencoder, K=10, lambda_P=0.6, lambda_Q=0.6)
```

```
history = nmf.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
```

```
Training NMF ...
k=10
epoch 1/10 - loss : 0.916 - val_loss : 0.917
epoch 2/10 - loss : 0.915 - val_loss : 0.917
epoch 3/10 - loss : 0.915 - val_loss : 0.917
epoch 4/10 - loss : 0.915 - val_loss : 0.917
epoch 5/10 - loss : 0.915 - val_loss : 0.917
epoch 6/10 - loss : 0.915 - val_loss : 0.917
epoch 7/10 - loss : 0.915 - val_loss : 0.917
epoch 8/10 - loss : 0.915 - val_loss : 0.917
epoch 9/10 - loss : 0.915 - val_loss : 0.917
epoch 10/10 - loss : 0.915 - val_loss : 0.917
```

Evaluate the model using test set

[7]:
```
nmf.evaluate(x_test, y_test)
```

```
validation error : 0.917
MAE :  0.9165041343019539
```

[7]:
```
0.9165041343019539
```

The result seems okay because the MAE score is less than 1.0, but it's pretty near to the 1.0 so it can't be called a good result

## 6 Evaluation of NMF with Scikit-suprise

We can use the scikit-suprise package to train the NMF model. It is an easy-to-use Python scikit for recommender systems.

1. Import the NMF class from the suprise scikit.
2. Load the data with the built-in function
3. Instanciate NMF with `k=10` (`n_factors`) and we use 10 epochs (`n_epochs`)
4. Evaluate the model using cross-validation with 5 folds.

**Dataset 1: ML-100K**

[8]:
```
from surprise import NMF
from surprise import Dataset
from surprise.model_selection import cross_validate

# Load the movielens-100k dataset (download it if needed).
data = Dataset.load_builtin('ml-100k')

# Use the NMF algorithm.
nmf = NMF(n_factors=10, n_epochs=10)

# Run 5-fold cross-validation and print results.
```

```
history = cross_validate(nmf, data, measures=['MAE'], cv=5, verbose=True)
```

```
Dataset ml-100k could not be found. Do you want to download it? [Y/n] Y
Trying to download dataset from
https://files.grouplens.org/datasets/movielens/ml-100k.zip...
Done! Dataset ml-100k has been saved to /root/.surprise_data/ml-100k
Evaluating MAE of algorithm NMF on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
MAE (testset)     0.9626  0.9435  0.9777  0.9607  0.9510  0.9591  0.0116
Fit time          0.38    0.43    0.37    0.42    0.43    0.40    0.03
Test time         0.12    0.21    0.14    0.21    0.12    0.16    0.04
```

As result, the mean MAE on the test set is **mae = 0.9591** which is pretty close to the result we have obtained on *ml-100k* with our own implementation **mae = 0.9165**

It has a different MAE score due to the use of cross validation. The use of cross validation is to ensure that the model is properly validated, so that the model doesn't easily show overfitting result

**Dataset 2: ML-1M** *This may take arount 2 minutes*

[9]:
```
data = Dataset.load_builtin('ml-1m')
nmf = NMF(n_factors=10, n_epochs=10)
history = cross_validate(nmf, data, measures=['MAE'], cv=5, verbose=True)
```

```
Dataset ml-1m could not be found. Do you want to download it? [Y/n] Y
Trying to download dataset from
https://files.grouplens.org/datasets/movielens/ml-1m.zip...
Done! Dataset ml-1m has been saved to /root/.surprise_data/ml-1m
Evaluating MAE of algorithm NMF on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
MAE (testset)     0.9559  0.9580  0.9566  0.9639  0.9490  0.9567  0.0047
Fit time          3.53    3.84    3.86    3.83    3.79    3.77    0.12
Test time         2.60    2.37    2.33    2.34    2.03    2.33    0.18
```

The mean MAE on a 5-fold cross-validation is **mae = 0.9567**

## 7 Reference


1. Daniel D. Lee & H. Sebastian Seung (1999). Learning the parts of objects by non-negative matrix factorization
2. Deng Cai et al. (2008). Non-negative Matrix Factorization on Manifold
3. Yu-Xiong Wang and Yu-Jin Zhang (2011). Non-negative Matrix Factorization: a Comprehensive Review
4. Nicolas Gillis (2014). The Why and How of Nonnegative Matrix Factorization

# 8 Author

Carmel WENGA, PhD student at Université de la Polynésie Française, Applied Machine Learning Research Engineer, ShoppingList, NzhinuSoft.

# Chapter 7 - Explainable Matrix Factorization

Before we start, we have to make sure that we already have our dependencies, that is 'recsys' folder

[1]:
```
import os
#Check if we already have the 'recsys' folder
if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    # If not then download directly from the source
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/
 ↪master/recsys.zip
    !unzip recsys.zip
```

## 1 Requirements

Other than the 'recsys' folder, we also have to make sure that the other required libs have already been installed

```
matplotlib==3.2.2
numpy==1.18.1
pandas==1.0.5
python==3.6.10
scikit-learn==0.23.1
scipy==1.5.0
```

(If we use Google Colab, most of these libs are already installed and up-to-date, except for *scikit-surprise* which is not pre-installed by Google Colab)

Import all of the required libs

[2]:
```
from recsys.memories.UserToUser import UserToUser

from recsys.preprocessing import mean_ratings
from recsys.preprocessing import normalized_ratings
from recsys.preprocessing import ids_encoder
from recsys.preprocessing import train_test_split
from recsys.preprocessing import rating_matrix
from recsys.preprocessing import get_examples

from recsys.datasets import ml100k, ml1m
```

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

import sys
import os
```

# 2 Explainable Matrix Factorization (EMF)

## 2.1 How to quantify explainability ?

- Use the rating distribution within the active user's neighborhood.
- If many neighbors have rated the recommended item, then this can provide a basis upon which to explain the recommendations, using neighborhood style explanation mechanisms

According to (Abdollahi and Nasraoui, 2016), an item $i$ is consider to be explainable for user $u$ if a considerable number of its neighbors rated item $i$. The explainability score $E_{ui}$ is the percentage of user $u$'s neighbors who have rated item $i$.

$$E_{ui} = \frac{|N_k^{(i)}(u)|}{|N_k(u)|}, \tag{1}$$

where $N_k(u)$ is the set of $k$ nearest neighbors of user $u$ and $N_k^{(i)}(u)$ is the set of user $u$'s neighbors who have rated item $i$. However, only explainable scores above an optimal threshold $\theta$ are accepted.

$$W_{ui} = \begin{cases} E_{ui} \; if \; E_{ui} > \theta \\ 0 \; otherwise \end{cases}, \tag{2}$$

By including explainability weight in the training algorithm, the new objective function, to be minimized over the set of known ratings, has been formulated by (Abdollahi and Nasraoui, 2016) as:

$$J = \sum_{(u,i)\in\kappa} (R_{ui} - \hat{R}_{ui})^2 + \frac{\beta}{2}(||P_u||^2 + ||Q_i||^2) + \frac{\lambda}{2}(P_u - Q_i)^2 W_{ui}, \tag{3}$$

here, $\frac{\beta}{2}(||P_u||^2 + ||Q_i||^2)$ is the $L_2$ regularization term weighted by the coefficient $\beta$, and $\lambda$ is an explainability regularization coefficient that controls the smoothness of the new representation and tradeoff between explainability and accuracy. The idea here is that if item $i$ is explainable for user $u$, then their representations in the latent space, $Q_i$ and $P_u$, should be close to each other. Stochastic Gradient descent can be used to optimize the objectve function.

$$P_u \leftarrow P_u + \alpha \left( 2(R_{u,i} - P_u Q_i^\top) Q_i - \beta P_u - \lambda (P_u - Q_i) W_{ui} \right) \tag{4}$$

$$Q_i \leftarrow Q_i + \alpha \left( 2(R_{u,i} - P_u Q_i^\top) P_u - \beta Q_i + \lambda (P_u - Q_i) W_{ui} \right) \tag{5}$$

### 2.2 Compute Explainable Scores

Explainable score are computed using neighborhood based similarities. Here, we are using the user based algorithm to compute similarities

[3]:
```
def explainable_score(user2user, users, items, theta=0):

    def _progress(count):
        sys.stdout.write('\rCompute Explainable score. Progress status : %.
↪1f%%'%(float(count/len(users))*100.0))
        sys.stdout.flush()
    # initialize explainable score to zeros
    W = np.zeros((len(users), len(items)))

    for count, u in enumerate(users):
        candidate_items = user2user.find_user_candidate_items(u)
        for i in candidate_items:
            user_who_rated_i, similar_user_who_rated_i = \
                user2user.similar_users_who_rated_this_item(u, i)
            if user_who_rated_i.shape[0] == 0:
                w = 0.0
            else:
                w = similar_user_who_rated_i.shape[0] / user_who_rated_i.shape[0]
            W[u,i] =  w  if w > theta else 0.0
        _progress(count)
    return W
```

## 3 Explainable Matrix Factorization Model

[4]:
```
class ExplainableMatrixFactorization:

    def __init__(self, m, n, W, alpha=0.001, beta=0.01, lamb=0.1, k=10):
        """
            - R : Rating matrix of shape (m,n)
            - W : Explainability Weights of shape (m,n)
            - k : number of latent factors
            - beta : L2 regularization parameter
            - lamb : explainability regularization coefficient
            - theta : threshold above which an item is explainable for a user
        """
        self.W = W
        self.m = m
        self.n = n

        np.random.seed(64)
```

```
        # initialize the latent factor matrices P and Q (of shapes (m,k) and␣
↪(n,k) respectively) that will be learnt
        self.k = k
        self.P = np.random.normal(size=(self.m,k))
        self.Q = np.random.normal(size=(self.n,k))

        # hyperparameter initialization
        self.alpha = alpha
        self.beta = beta
        self.lamb = lamb

        # training history
        self.history = {
            "epochs":[],
            "loss":[],
            "val_loss":[],
        }

    def print_training_parameters(self):
        print('Training EMF')
        print(f'k={self.k} \t alpha={self.alpha} \t beta={self.beta} \t␣
↪lambda={self.lamb}')

    def update_rule(self, u, i, error):
        self.P[u] = self.P[u] + \
            self.alpha*(2 * error*self.Q[i] - self.beta*self.P[u] - self.
↪lamb*(self.P[u] - self.Q[i]) * self.W[u,i])

        self.Q[i] = self.Q[i] + \
            self.alpha*(2 * error*self.P[u] - self.beta*self.Q[i] + self.
↪lamb*(self.P[u] - self.Q[i]) * self.W[u,i])

    def mae(self,  x_train, y_train):
        """
        returns the Mean Absolute Error
        """
        # number of training exemples
        M = x_train.shape[0]
        error = 0
        for pair, r in zip(x_train, y_train):
            u, i = pair
            error += np.absolute(r - np.dot(self.P[u], self.Q[i]))
        return error/M

    def print_training_progress(self, epoch, epochs, error, val_error, steps=5):
        if epoch == 1 or epoch % steps == 0 :
```

```
                print(f"epoch {epoch}/{epochs} - loss : {round(error,3)} -␣
↪val_loss : {round(val_error,3)}")

    def learning_rate_schedule(self, epoch, target_epochs = 20):
        if (epoch >= target_epochs) and (epoch % target_epochs == 0):
                factor = epoch // target_epochs
                self.alpha = self.alpha * (1 / (factor * 20))
                print("\nLearning Rate : {}\n".format(self.alpha))

    def fit(self, x_train, y_train, validation_data, epochs=10):
        """
        Train latent factors P and Q according to the training set

        :param
            - x_train : training pairs (u,i) for which rating r_ui is known
            - y_train : set of ratings r_ui for all training pairs (u,i)
            - validation_data : tuple (x_test, y_test)
            - epochs : number of time to loop over the entire training set.
            10 epochs by default

        Note that u and i are encoded values of userid and itemid
        """
        self.print_training_parameters()

        # get validation data
        x_test, y_test = validation_data

        for epoch in range(1, epochs+1):
            for pair, r in zip(x_train, y_train):
                u,i = pair
                r_hat = np.dot(self.P[u], self.Q[i])
                e = r - r_hat
                self.update_rule(u, i, error=e)

            # training and validation error  after this epochs
            error = self.mae(x_train, y_train)
            val_error = self.mae(x_test, y_test)
            self.update_history(epoch, error, val_error)
            self.print_training_progress(epoch, epochs, error, val_error,␣
↪steps=1)

        return self.history

    def update_history(self, epoch, error, val_error):
        self.history['epochs'].append(epoch)
        self.history['loss'].append(error)
        self.history['val_loss'].append(val_error)
```

```
    def evaluate(self, x_test, y_test):
        """
        compute the global error on the test set

        :param
            - x_test : test pairs (u,i) for which rating r_ui is known
            - y_test : set of ratings r_ui for all test pairs (u,i)
        """
        error = self.mae(x_test, y_test)
        print(f"validation error : {round(error,3)}")

    def predict(self, userid, itemid):
        """
        Make rating prediction for a user on an item

        :param
        - userid
        - itemid

        :return
        - r : predicted rating
        """
        # encode user and item ids to be able to access their latent factors in
        # matrices P and Q
        u = uencoder.transform([userid])[0]
        i = iencoder.transform([itemid])[0]

        # rating prediction using encoded ids. Dot product between P_u and Q_i
        r = np.dot(self.P[u], self.Q[i])

        return r

    def recommend(self, userid, N=30):
        """
        make to N recommendations for a given user

        :return
        - (top_items,preds) : top N items with the highest predictions
        """
        # encode the userid
        u = uencoder.transform([userid])[0]

        # predictions for this user on all product
        predictions = np.dot(self.P[u], self.Q.T)

        # get the indices of the top N predictions
```

```
        top_idx = np.flip(np.argsort(predictions))[:N]

        # decode indices to get their corresponding itemids
        top_items = iencoder.inverse_transform(top_idx)

        # take corresponding predictions for top N indices
        preds = predictions[top_idx]

        return top_items, preds
```

Define the number of epoch for training process

[5]:
```
epochs = 10
```

# 4 Model Evaluation

**Dataset 1: MovieLens 100K**

**Evaluation on raw ratings**

[6]:
```
# Load ml100k dataset
ratings, movies = ml100k.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

users = sorted(ratings.userid.unique())
items = sorted(ratings.itemid.unique())

m = len(users) # total number of users
n = len(items) # total number of items

# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings)

# Split dataset into train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
 ↪labels=raw_labels)
```

[7]:
```
# Create the user to user model for similarity measure
usertouser = UserToUser(ratings, movies)

# Compute explainable score
W = explainable_score(usertouser, users, items)
```

```
Normalize users ratings ...
Initialize the similarity model ...
Compute nearest neighbors ...
```

```
User to user recommendation model created with success ...
Compute Explainable score. Progress status : 99.9%
```

[8]:
```
# Construct the model
EMF = ExplainableMatrixFactorization(m, n, W, alpha=0.01, beta=0.4, lamb=0.01,␣
 ↪k=10)
# Train the model with training data
history = EMF.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
 ↪y_test))
```

```
Training EMF
k=10     alpha=0.01      beta=0.4        lambda=0.01
epoch 1/10 - loss : 0.922 - val_loss : 1.036
epoch 2/10 - loss : 0.79 - val_loss : 0.873
epoch 3/10 - loss : 0.766 - val_loss : 0.837
epoch 4/10 - loss : 0.757 - val_loss : 0.822
epoch 5/10 - loss : 0.753 - val_loss : 0.814
epoch 6/10 - loss : 0.751 - val_loss : 0.808
epoch 7/10 - loss : 0.749 - val_loss : 0.805
epoch 8/10 - loss : 0.748 - val_loss : 0.802
epoch 9/10 - loss : 0.746 - val_loss : 0.799
epoch 10/10 - loss : 0.745 - val_loss : 0.797
```

Evaluate the model with testing data

[9]:
```
EMF.evaluate(x_test, y_test)
```

```
validation error : 0.797
```

The result is pretty okay because the MAE score is less than 1.0

#####Evaluation on normalized ratings

[10]:
```
# load ml100k dataset
ratings, movies = ml100k.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

users = sorted(ratings.userid.unique())
items = sorted(ratings.itemid.unique())

m = len(users) # total number of users
n = len(items) # total number of items

# Normalize ratings by subtracting means
normalized_column_name = "norm_rating"
ratings = normalized_ratings(ratings, norm_column=normalized_column_name)
```

```
# Get examples as tuples of userids and itemids and labels from normalized␣
↪ratings
raw_examples, raw_labels = get_examples(ratings,␣
↪labels_column=normalized_column_name)

# Split dataset into train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)
```

[11]:
```
# Construct the model
EMF = ExplainableMatrixFactorization(m, n, W, alpha=0.022, beta=0.65, lamb=0.01,␣
↪k=10)
# Train the model with training data
history = EMF.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
↪y_test))
```

```
Training EMF
k=10     alpha=0.022     beta=0.65       lambda=0.01
epoch 1/10 - loss : 0.809 - val_loss : 0.842
epoch 2/10 - loss : 0.809 - val_loss : 0.829
epoch 3/10 - loss : 0.807 - val_loss : 0.821
epoch 4/10 - loss : 0.799 - val_loss : 0.811
epoch 5/10 - loss : 0.789 - val_loss : 0.8
epoch 6/10 - loss : 0.782 - val_loss : 0.793
epoch 7/10 - loss : 0.778 - val_loss : 0.789
epoch 8/10 - loss : 0.776 - val_loss : 0.786
epoch 9/10 - loss : 0.774 - val_loss : 0.784
epoch 10/10 - loss : 0.773 - val_loss : 0.783
```

Evaluate the model with testing data

[12]:
```
EMF.evaluate(x_test, y_test)
```

```
validation error : 0.783
```

The result is pretty okay because the MAE score is less than 1.0 It also has a better score on normalized data than on raw data

### Dataset 2: MovieLens 1M

#### Evaluation on raw ratings

[13]:
```
# Load ml1m dataset
ratings, movies = ml1m.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

users = sorted(ratings.userid.unique())
```

```python
items = sorted(ratings.itemid.unique())

m = len(users) # total number of users
n = len(items) # total number of items

# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings)

# Split dataset into train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,
 labels=raw_labels)
```

[14]:
```python
# Create the user to user model for similarity measure
usertouser = UserToUser(ratings, movies)

# Compute explainable score
W = explainable_score(usertouser, users, items)
```

```
Normalize users ratings ...
Initialize the similarity model ...
Compute nearest neighbors ...
User to user recommendation model created with success ...
Compute Explainable score. Progress status : 100.0%
```

[15]:
```python
# Construct the model
EMF = ExplainableMatrixFactorization(m, n, W, alpha=0.01, beta=0.4, lamb=0.01,
 k=10)
# Train the model with training data
history = EMF.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,
 y_test))
```

```
Training EMF
k=10     alpha=0.01      beta=0.4        lambda=0.01
epoch 1/10 - loss : 0.782 - val_loss : 0.807
epoch 2/10 - loss : 0.762 - val_loss : 0.781
epoch 3/10 - loss : 0.76 - val_loss : 0.775
epoch 4/10 - loss : 0.758 - val_loss : 0.771
epoch 5/10 - loss : 0.757 - val_loss : 0.769
epoch 6/10 - loss : 0.756 - val_loss : 0.767
epoch 7/10 - loss : 0.754 - val_loss : 0.764
epoch 8/10 - loss : 0.752 - val_loss : 0.762
epoch 9/10 - loss : 0.751 - val_loss : 0.761
epoch 10/10 - loss : 0.75 - val_loss : 0.76
```

Evaluate the model with testing data

[16]:
```python
EMF.evaluate(x_test, y_test)
```

```
validation error : 0.76
```

The result is pretty okay because the MAE score is less than 1.0

**Evaluation on normalized ratings**

[17]:
```
# Load ml1m dataset
ratings, movies = ml1m.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

# Normalize ratings by subtracting means
normalized_column_name = "norm_rating"
ratings = normalized_ratings(ratings, norm_column=normalized_column_name)

# Get examples as tuples of userids and itemids and labels from normalized␣
↪ratings
raw_examples, raw_labels = get_examples(ratings,␣
↪labels_column=normalized_column_name)

# Split dataset into train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)
```

[18]:
```
# Construct the model
EMF = ExplainableMatrixFactorization(m, n, W, alpha=0.023, beta=0.59, lamb=0.01,␣
↪k=10)
# Train the model with training data
history = EMF.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
↪y_test))
```

```
Training EMF
k=10     alpha=0.023     beta=0.59       lambda=0.01
epoch 1/10 - loss : 0.805 - val_loss : 0.814
epoch 2/10 - loss : 0.764 - val_loss : 0.77
epoch 3/10 - loss : 0.756 - val_loss : 0.762
epoch 4/10 - loss : 0.755 - val_loss : 0.759
epoch 5/10 - loss : 0.754 - val_loss : 0.759
epoch 6/10 - loss : 0.754 - val_loss : 0.758
epoch 7/10 - loss : 0.754 - val_loss : 0.758
epoch 8/10 - loss : 0.753 - val_loss : 0.758
epoch 9/10 - loss : 0.753 - val_loss : 0.758
epoch 10/10 - loss : 0.753 - val_loss : 0.758
```

Evaluate the model with testing data

[19]:
```
EMF.evaluate(x_test, y_test)
```

```
validation error : 0.758
```

The result is pretty okay because the MAE score is less than 1.0 It also has a better score on normalized data than on raw data

## 5 Ratings prediction

Find the predicted rating of the user of id = 42 for each movies and select the top-30 movies with highest predicted rating

[20]:
```
# get list of top N items with their corresponding predicted ratings
userid = 42
recommended_items, predictions = EMF.recommend(userid=userid)

# find corresponding movie titles
top_N = list(zip(recommended_items,predictions))
top_N = pd.DataFrame(top_N, columns=['itemid','predictions'])
top_N.predictions = top_N.predictions + ratings.loc[ratings.userid==userid].
 ↪rating_mean.values[0]
List = pd.merge(top_N, movies, on='itemid', how='inner')

# show the list
List
```

[20]:

| | itemid | predictions | title \ |
|---|---|---|---|
| 0 | 3460 | 4.364036 | Hillbillys in a Haunted House (1967) |
| 1 | 701 | 4.324177 | Daens (1992) |
| 2 | 3057 | 4.307404 | Where's Marlowe? (1999) |
| 3 | 2214 | 4.304979 | Number Seventeen (1932) |
| 4 | 1145 | 4.299559 | Snowriders (1996) |
| 5 | 2258 | 4.292125 | Master Ninja I (1984) |
| 6 | 3353 | 4.281912 | Closer You Get, The (2000) |
| 7 | 868 | 4.278937 | Death in Brunswick (1991) |
| 8 | 826 | 4.269901 | Diebinnen (1995) |
| 9 | 3305 | 4.266769 | Bluebeard (1944) |
| 10 | 2619 | 4.265997 | Mascara (1999) |
| 11 | 763 | 4.264092 | Last of the High Kings, The (a.k.a. Summer Fli... |
| 12 | 1852 | 4.262517 | Love Walked In (1998) |
| 13 | 642 | 4.260353 | Roula (1995) |
| 14 | 682 | 4.258829 | Tigrero: A Film That Was Never Made (1994) |
| 15 | 792 | 4.253339 | Hungarian Fairy Tale, A (1987) |
| 16 | 1316 | 4.252915 | Anna (1996) |
| 17 | 3228 | 4.245526 | Wirey Spindell (1999) |
| 18 | 853 | 4.240745 | Dingo (1992) |
| 19 | 3172 | 4.238188 | Ulysses (Ulisse) (1954) |
| 20 | 2254 | 4.238008 | Choices (1981) |
| 21 | 2503 | 4.234547 | Apple, The (Sib) (1998) |
| 22 | 2905 | 4.224974 | Sanjuro (1962) |
| 23 | 744 | 4.224278 | Brothers in Trouble (1995) |
| 24 | 757 | 4.224226 | Ashes of Time (1994) |

```
25    858    4.223665                        Godfather, The (1972)
26    789    4.220788   I, Worst of All (Yo, la peor de todas) (1990)
27   3748    4.216508                            Match, The (1999)
28    790    4.216455              An Unforgettable Summer (1994)
29    745    4.215986                       Close Shave, A (1995)

                      genres
0                     Comedy
1                      Drama
2                     Comedy
3                   Thriller
4                Documentary
5                     Action
6             Comedy|Romance
7                     Comedy
8                      Drama
9          Film-Noir|Horror
10                     Drama
11                     Drama
12            Drama|Thriller
13                     Drama
14         Documentary|Drama
15                   Fantasy
16                     Drama
17                    Comedy
18                     Drama
19                 Adventure
20                     Drama
21                     Drama
22          Action|Adventure
23                     Drama
24                     Drama
25        Action|Crime|Drama
26                     Drama
27            Comedy|Romance
28                     Drama
29  Animation|Comedy|Thriller
```

**Note**: The recommendation list may content items already purchased by the user. This is just an illustration of how to implement matrix factorization recommender system. You can optimize the recommended list and return the top rated items that the user has not already purchased.

# 6 Reference


1. Yehuda Koren et al. (2009). Matrix Factorization Techniques for Recommender Systems
2. Abdollahi and Nasraoui (2016). Explainable Matrix Factorization for Collaborative Filtering
3. Abdollahi and Nasraoui (2017). Using Explainability for Constrained Matrix Factorization

4. Shuo Wang et al, (2018). Explainable Matrix Factorization with Constraints on Neighborhood in the Latent Space

# 7 Author

Carmel WENGA, PhD student at Université de la Polynésie Française, Applied Machine Learning Research Engineer, ShoppingList, NzhinuSoft.

# Chapter 8 - Performance Measure

Before we start, we have to make sure that we already have our dependencies, that is 'recsys' folder

[1]:
```
import os
#Check if we already have the 'recsys' folder
if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    # If not then download directly from the source
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/
↪master/recsys.zip
    !unzip recsys.zip
```

```
--2023-01-03 20:49:26--  https://github.com/nzhinusoftcm/review-on-
collaborative-filtering/raw/master/recsys.zip
Resolving github.com (github.com)... 192.30.255.112
Connecting to github.com (github.com)|192.30.255.112|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/nzhinusoftcm/review-on-
collaborative-filtering/master/recsys.zip [following]
--2023-01-03 20:49:27--  https://raw.githubusercontent.com/nzhinusoftcm/review-
on-collaborative-filtering/master/recsys.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.111.133, 185.199.109.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 15312323 (15M) [application/zip]
Saving to: 'recsys.zip'

recsys.zip          100%[===================>]  14.60M  --.-KB/s    in 0.08s

2023-01-03 20:49:27 (186 MB/s) - 'recsys.zip' saved [15312323/15312323]

Archive:  recsys.zip
   creating: recsys/
  inflating: recsys/datasets.py
  inflating: recsys/preprocessing.py
  inflating: recsys/utils.py
  inflating: recsys/requirements.txt
```

```
 creating: recsys/.vscode/
inflating: recsys/.vscode/settings.json
 creating: recsys/__pycache__/
inflating: recsys/__pycache__/datasets.cpython-36.pyc
inflating: recsys/__pycache__/datasets.cpython-37.pyc
inflating: recsys/__pycache__/utils.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-37.pyc
inflating: recsys/__pycache__/datasets.cpython-38.pyc
inflating: recsys/__pycache__/preprocessing.cpython-36.pyc
inflating: recsys/__pycache__/preprocessing.cpython-38.pyc
 creating: recsys/memories/
inflating: recsys/memories/ItemToItem.py
inflating: recsys/memories/UserToUser.py
 creating: recsys/memories/__pycache__/
inflating: recsys/memories/__pycache__/UserToUser.cpython-36.pyc
inflating: recsys/memories/__pycache__/UserToUser.cpython-37.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-37.pyc
inflating: recsys/memories/__pycache__/user2user.cpython-36.pyc
inflating: recsys/memories/__pycache__/ItemToItem.cpython-36.pyc
 creating: recsys/models/
inflating: recsys/models/SVD.py
inflating: recsys/models/MatrixFactorization.py
inflating: recsys/models/ExplainableMF.py
inflating: recsys/models/NonnegativeMF.py
 creating: recsys/models/__pycache__/
inflating: recsys/models/__pycache__/SVD.cpython-36.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-37.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-36.pyc
inflating: recsys/models/__pycache__/ExplainableMF.cpython-37.pyc
inflating: recsys/models/__pycache__/MatrixFactorization.cpython-36.pyc
 creating: recsys/metrics/
inflating: recsys/metrics/EvaluationMetrics.py
 creating: recsys/img/
inflating: recsys/img/MF-and-NNMF.png
inflating: recsys/img/svd.png
inflating: recsys/img/MF.png
 creating: recsys/predictions/
 creating: recsys/predictions/item2item/
 creating: recsys/weights/
 creating: recsys/weights/item2item/
 creating: recsys/weights/item2item/ml1m/
inflating: recsys/weights/item2item/ml1m/similarities.npy
inflating: recsys/weights/item2item/ml1m/neighbors.npy
 creating: recsys/weights/item2item/ml100k/
inflating: recsys/weights/item2item/ml100k/similarities.npy
inflating: recsys/weights/item2item/ml100k/neighbors.npy
```

# 1 Requirements

Other than the 'recsys' folder, we also have to make sure that the other required libs have already been installed

```
matplotlib==3.2.2
numpy==1.19.2
pandas==1.0.5
python==3.7
scikit-learn==0.24.1
scikit-surprise==1.1.1
scipy==1.6.2
```

(If we use Google Colab, most of these libs are already installed and up-to-date, except for *scikit-surprise* which is not pre-installed by Google Colab)

Because *scikit-surprise* is required by current notebook, we're going to install *scikit-surprise* right away

[2]:
```
!pip install surprise
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting surprise
  Downloading surprise-0.1-py2.py3-none-any.whl (1.8 kB)
Collecting scikit-surprise
  Downloading scikit-surprise-1.1.3.tar.gz (771 kB)
      772.0/772.0 KB
15.4 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: joblib>=1.0.0 in /usr/local/lib/python3.8/dist-
packages (from scikit-surprise->surprise) (1.2.0)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.8/dist-
packages (from scikit-surprise->surprise) (1.21.6)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.8/dist-
packages (from scikit-surprise->surprise) (1.7.3)
Building wheels for collected packages: scikit-surprise
  Building wheel for scikit-surprise (setup.py) ... done
  Created wheel for scikit-surprise:
filename=scikit_surprise-1.1.3-cp38-cp38-linux_x86_64.whl size=2626484
sha256=6a83221df4245477e57cba3d14e9c59c26f18e4c4de33cb236f3cd7cd16362ea
  Stored in directory: /root/.cache/pip/wheels/af/db/86/2c18183a80ba05da35bf0fb7
417aac5cddbd93bcb1b92fd3ea
Successfully built scikit-surprise
Installing collected packages: scikit-surprise, surprise
Successfully installed scikit-surprise-1.1.3 surprise-0.1
```

Import all of the required libs

```
[3]: from recsys.memories.UserToUser import UserToUser
from recsys.memories.ItemToItem import ItemToItem

from recsys.models.MatrixFactorization import MF
from recsys.models.ExplainableMF import EMF, explainable_score

from recsys.preprocessing import normalized_ratings
from recsys.preprocessing import train_test_split
from recsys.preprocessing import rating_matrix
from recsys.preprocessing import scale_ratings
from recsys.preprocessing import mean_ratings
from recsys.preprocessing import get_examples
from recsys.preprocessing import ids_encoder

from recsys.datasets import ml100k
from recsys.datasets import ml1m

from sklearn.preprocessing import LabelEncoder

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

import os
```

# 2 Results on MovieLens 100k Dataset

## 2.1 User-based CF

```
[4]: # Load data
ratings, movies = ml100k.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings, labels_column='rating')

# Split dataset into train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,
 ↪labels=raw_labels)
```

```
Download data 100.2%
Successfully downloaded ml-100k.zip 4924029 bytes.
Unzipping the ml-100k.zip zip file ...
```

### 2.1.1 Evaluation with Euclidean Distance

[5]:
```
# Evaluate with Euclidean distance
usertouser = UserToUser(ratings, movies, metric='euclidean')
print("==========================")
usertouser.evaluate(x_test, y_test)
```

```
Normalize users ratings ...
Initialize the similarity model ...
Compute nearest neighbors ...
User to user recommendation model created with success ...
==========================
Evaluate the model on 10000 test data ...

MAE : 0.8125945111976461
```

[5]:
```
0.8125945111976461
```

### 2.1.2 Evaluation with Cosine Similarity

[6]:
```
# Evaluate with cosine similarity
usertouser = UserToUser(ratings, movies, metric='cosine')
print("=========================")
usertouser.evaluate(x_test, y_test)
```

```
Normalize users ratings ...
Initialize the similarity model ...
Compute nearest neighbors ...
User to user recommendation model created with success ...
=========================
Evaluate the model on 10000 test data ...

MAE : 0.7505910931068639
```

[6]:
```
0.7505910931068639
```

## 2.2 Item-based CF

[7]:
```
# Load data
ratings, movies = ml100k.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings, labels_column='rating')

# Split dataset into train set and test set
```

```
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)
```

### 2.2.1 Evaluation with Euclidean Distance

[8]:
```
# Evaluation with Euclidean distance
itemtoitem = ItemToItem(ratings, movies, metric='euclidean')
print("==================")
itemtoitem.evaluate(x_test, y_test)
```

```
Normalize ratings ...
Create the similarity model ...
Compute nearest neighbors ...
Item to item recommendation model created with success ...
==================
Evaluate the model on 10000 test data ...

MAE : 0.8277111416143341
```

[8]:
```
0.8277111416143341
```

### 2.2.2 Evaluation with Cosine Similarity

[9]:
```
# Evaluation with cosine similarity
itemtoitem = ItemToItem(ratings, movies, metric='cosine')
print("==================")
itemtoitem.evaluate(x_test, y_test)
```

```
Normalize ratings ...
Create the similarity model ...
Compute nearest neighbors ...
Item to item recommendation model created with success ...
==================
Evaluate the model on 10000 test data ...

MAE : 0.507794195659005
```

[9]:
```
0.507794195659005
```

## 2.3 Matrix Factorization

[10]:
```
#Define the number of epoch for training process
epochs = 10
```

[11]:
```
# Load the ml100k dataset
ratings, movies = ml100k.load()

# Encode userid and itemid in ratings
```

```
ratings, uencoder, iencoder = ids_encoder(ratings)

m = ratings.userid.nunique()   # total number of users
n = ratings.itemid.nunique()   # total number of items

# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings)

# Split dataset into train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
 ↪labels=raw_labels)

# Create the model
mf = MF(m, n, k=10, alpha=0.01, lamb=1.5)

# Fit the model on the training set
history = mf.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
 ↪y_test))

# Evaluate the model with testing set
print("==================")
mf.evaluate(x_test, y_test)
```

```
Training Matrix Factorization Model ...
k=10     alpha=0.01      lambda=1.5
epoch 1/10 - loss : 2.734 - val_loss : 2.779
epoch 2/10 - loss : 1.764 - val_loss : 1.794
epoch 3/10 - loss : 1.592 - val_loss : 1.614
epoch 4/10 - loss : 1.538 - val_loss : 1.556
epoch 5/10 - loss : 1.515 - val_loss : 1.531
epoch 6/10 - loss : 1.503 - val_loss : 1.517
epoch 7/10 - loss : 1.496 - val_loss : 1.509
epoch 8/10 - loss : 1.491 - val_loss : 1.504
epoch 9/10 - loss : 1.488 - val_loss : 1.5
epoch 10/10 - loss : 1.486 - val_loss : 1.497
==================
validation error : 1.497
```

[11]: 1.4973507972141993

### 2.4 Non-negative Matrix Factorization

[12]:
```
from surprise import NMF
from surprise import Dataset
from surprise.model_selection import cross_validate

# Load the movielens-100k dataset (download it if needed).
```

```
data = Dataset.load_builtin('ml-100k')

# Use the NMF algorithm.
nmf = NMF(n_factors=10, n_epochs=10)

# Run 5-fold cross-validation and print results.
history = cross_validate(nmf, data, measures=['MAE'], cv=5, verbose=True)
```

```
Dataset ml-100k could not be found. Do you want to download it? [Y/n] Y
Trying to download dataset from
https://files.grouplens.org/datasets/movielens/ml-100k.zip...
Done! Dataset ml-100k has been saved to /root/.surprise_data/ml-100k
Evaluating MAE of algorithm NMF on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
MAE (testset)     0.9615  0.9501  0.9548  0.9582  0.9675  0.9584  0.0059
Fit time          0.55    0.45    0.49    0.45    0.50    0.49    0.04
Test time         0.18    0.29    0.14    0.23    0.15    0.20    0.06
```

### 2.5 Explainable Matrix Factorization

[13]:
```
# load data
ratings, movies = ml100k.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

users = sorted(ratings.userid.unique())
items = sorted(ratings.itemid.unique())

m = len(users)
n = len(items)

# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings)

# Split dataset into train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)

# Create the user to user model for similarity measure
usertouser = UserToUser(ratings, movies)

# Compute explainable score
W = explainable_score(usertouser, users, items)

print("==================")
```

```
# Create the model
emf = EMF(m, n, W, alpha=0.01, beta=0.4, lamb=0.01, k=10)
# Train the model with training data
history = emf.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
↪y_test))

print("===================")
#Evaluate model with testing data
emf.evaluate(x_test, y_test)
```

```
Normalize users ratings ...
Initialize the similarity model ...
Compute nearest neighbors ...
User to user recommendation model created with success ...
Compute explainable scores ...
===================
Training EMF
k=10     alpha=0.01      beta=0.4        lambda=0.01
epoch 1/10 - loss : 0.922 - val_loss : 1.036
epoch 2/10 - loss : 0.79 - val_loss : 0.873
epoch 3/10 - loss : 0.766 - val_loss : 0.837
epoch 4/10 - loss : 0.757 - val_loss : 0.822
epoch 5/10 - loss : 0.753 - val_loss : 0.814
epoch 6/10 - loss : 0.751 - val_loss : 0.808
epoch 7/10 - loss : 0.749 - val_loss : 0.805
epoch 8/10 - loss : 0.748 - val_loss : 0.802
epoch 9/10 - loss : 0.746 - val_loss : 0.799
epoch 10/10 - loss : 0.745 - val_loss : 0.797
===================
MAE : 0.797
```

[13]: 0.797347824723284

## 3 Results on MovieLens 1M (ML-1M)

### 3.1 User-based CF

[14]:
```
# load ml100k ratings
ratings, movies = ml1m.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

# get examples as tuples of userids and itemids and labels from normalize ratings
raw_examples, raw_labels = get_examples(ratings, labels_column='rating')

# Split dataset into train set and test set
```

```
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
↪labels=raw_labels)
```

```
Download data 100.1%
Successfully downloaded ml-1m.zip 5917549 bytes.
Unzipping the ml-1m.zip zip file ...
```

#### 3.1.1 Evaluation with Euclidean Distance

[15]:
```
# Create the user-based CF with 'euclidean' metric
usertouser = UserToUser(ratings, movies, k=20, metric='euclidean')

# Evaluate the user-based CF on the ml1m test data
print("==========================")
usertouser.evaluate(x_test, y_test)
```

```
Normalize users ratings ...
Initialize the similarity model ...
Compute nearest neighbors ...
User to user recommendation model created with success ...
==========================
Evaluate the model on 100021 test data ...

MAE : 0.8069332535426614
```

[15]:
```
0.8069332535426614
```

#### 3.1.2 Evaluation with Cosine Similarity

[16]:
```
# Create the user-based CF with 'cosine' metric
usertouser = UserToUser(ratings, movies, k=20, metric='cosine')

# Evaluate the user-based CF on the ml1m test data
print("==========================")
usertouser.evaluate(x_test, y_test)
```

```
Normalize users ratings ...
Initialize the similarity model ...
Compute nearest neighbors ...
User to user recommendation model created with success ...
==========================
Evaluate the model on 100021 test data ...

MAE : 0.732267005840993
```

[16]:
```
0.732267005840993
```

## 3.2 Item-based CF

### 3.2.1 Evaluation with Euclidean Distance

[17]:
```
itemtoitem = ItemToItem(ratings, movies, metric='euclidean')
print("==========================")
#Evaluate model with testing data
itemtoitem.evaluate(x_test, y_test)
```

```
Normalize ratings ...
Create the similarity model ...
Compute nearest neighbors ...
Item to item recommendation model created with success ...
==========================
Evaluate the model on 100021 test data ...

MAE : 0.82502173206615
```

[17]:
```
0.82502173206615
```

### 3.2.2 Evaluation with Cosine Similarity

[18]:
```
itemtoitem = ItemToItem(ratings, movies, metric='cosine')
print("==========================")
# Evaluate model with testing data
itemtoitem.evaluate(x_test, y_test)
```

```
Normalize ratings ...
Create the similarity model ...
Compute nearest neighbors ...
Item to item recommendation model created with success ...
==========================
Evaluate the model on 100021 test data ...

MAE : 0.42514728655396045
```

[18]:
```
0.42514728655396045
```

## 3.3 Matrix Factorization

[19]:
```
# Load the ml1m dataset
ratings, movies = ml1m.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

m = ratings.userid.nunique()   # total number of users
n = ratings.itemid.nunique()   # total number of items
```

```
# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings)

# Split dataset into train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
 ↪labels=raw_labels)

# Create the model
model = MF(m, n, k=10, alpha=0.01, lamb=1.5)

# Fit the model on the training set
history = model.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
 ↪y_test))

print("===================")
# Evaluate model with testing data
model.evaluate(x_test, y_test)
```

```
Training Matrix Factorization Model ...
k=10     alpha=0.01      lambda=1.5
epoch 1/10 - loss : 1.713 - val_loss : 1.718
epoch 2/10 - loss : 1.523 - val_loss : 1.526
epoch 3/10 - loss : 1.496 - val_loss : 1.498
epoch 4/10 - loss : 1.489 - val_loss : 1.489
epoch 5/10 - loss : 1.485 - val_loss : 1.486
epoch 6/10 - loss : 1.484 - val_loss : 1.484
epoch 7/10 - loss : 1.483 - val_loss : 1.483
epoch 8/10 - loss : 1.483 - val_loss : 1.483
epoch 9/10 - loss : 1.482 - val_loss : 1.482
epoch 10/10 - loss : 1.482 - val_loss : 1.482
===================
validation error : 1.482
```

[19]:
```
1.4820034560467208
```

### 3.4 Non-negative Matrix Factorization

[20]:
```
from surprise import NMF
from surprise import Dataset
from surprise.model_selection import cross_validate

# Load the movielens-100k dataset (download it if needed).
data = Dataset.load_builtin('ml-1m')

# Use the NMF algorithm.
nmf = NMF(n_factors=10, n_epochs=10)
```

```
# Run 5-fold cross-validation and print results.
history = cross_validate(nmf, data, measures=['MAE'], cv=5, verbose=True)
```

```
Dataset ml-1m could not be found. Do you want to download it? [Y/n] Y
Trying to download dataset from
https://files.grouplens.org/datasets/movielens/ml-1m.zip...
Done! Dataset ml-1m has been saved to /root/.surprise_data/ml-1m
Evaluating MAE of algorithm NMF on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
MAE (testset)     0.9435  0.9456  0.9527  0.9546  0.9524  0.9498  0.0044
Fit time          5.29    6.11    6.15    5.92    5.61    5.82    0.32
Test time         2.15    3.63    3.68    2.57    4.07    3.22    0.73
```

### 3.5 Explainable Matrix Factorization

[21]:
```
# Load data
ratings, movies = ml1m.load()

# Encode userid and itemid in ratings
ratings, uencoder, iencoder = ids_encoder(ratings)

users = sorted(ratings.userid.unique())
items = sorted(ratings.itemid.unique())

m = len(users)
n = len(items)

# Get examples as tuples of userids and itemids and labels from raw ratings
raw_examples, raw_labels = get_examples(ratings)

# Split dataset into train set and test set
(x_train, x_test), (y_train, y_test) = train_test_split(examples=raw_examples,␣
 ↪labels=raw_labels)

# Create the user to user model for similarity measure
usertouser = UserToUser(ratings, movies)

# Compute explainable score
W = explainable_score(usertouser, users, items)

# Construct the model
emf = EMF(m, n, W, alpha=0.01, beta=0.4, lamb=0.01, k=10)
# Train the model with training set
history = emf.fit(x_train, y_train, epochs=epochs, validation_data=(x_test,␣
 ↪y_test))
```

```python
print("===================")
#Evaluate model with testing data
emf.evaluate(x_test, y_test)
```

```
Normalize users ratings ...
Initialize the similarity model ...
Compute nearest neighbors ...
User to user recommendation model created with success ...
Compute explainable scores ...
Training EMF
k=10     alpha=0.01      beta=0.4        lambda=0.01
epoch 1/10 - loss : 0.782 - val_loss : 0.807
epoch 2/10 - loss : 0.762 - val_loss : 0.781
epoch 3/10 - loss : 0.76 - val_loss : 0.775
epoch 4/10 - loss : 0.758 - val_loss : 0.771
epoch 5/10 - loss : 0.757 - val_loss : 0.769
epoch 6/10 - loss : 0.756 - val_loss : 0.767
epoch 7/10 - loss : 0.754 - val_loss : 0.764
epoch 8/10 - loss : 0.752 - val_loss : 0.762
epoch 9/10 - loss : 0.751 - val_loss : 0.761
epoch 10/10 - loss : 0.75 - val_loss : 0.76
===================
MAE : 0.76
```

```
[21]: 0.7596115374525224
```

# 4 Summary

MAE comparison between User-based and Item-based CF

| Metric | Dataset | User-based | Item-based |
|---|---|---|---|
| Euclidean | ML-100k | 0.81 | 0.83 |
| Euclidean | ML-1M | 0.81 | 0.82 |
| Cosine | ML-100k | 0.75 | 0.51 |
| Cosine | ML-1M | 0.73 | 0.42 |

MAE comparison between MF, NMF and EMF

| Preprocessing | Dataset | MF | NMF | EMF |
|---|---|---|---|---|
| Raw data | ML-100k | 1.497 | 0.951 | 0.797 |
| Raw data | ML-1M | 1.482 | 0.9567 | 0.76 |
| Normalized data | ML-100k | 0.828 | — | 0.783 |
| Normalized data | ML-1M | 0.825 | — | 0.758 |

# 5 Author

Carmel WENGA, PhD student at Université de la Polynésie Française, Applied Machine Learning Research Engineer, ShoppingList, NzhinuSoft.