Final Exam Practice and Evaluation

This is the project of final exam in Advanced Computing and Big Data course.

I printed the codes with the result and comments as PDF files to make it as book chapters.

This file contains the following:

Chapter 1: Data Exploration

In this chapter we explore the movielen lasted small dataset. This dataset is used throughout this repository to build collaborative filtering recommender systems.

Section 1.1: Requirement tools for the next codes

```
import os

if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/master/recsys.zip
    !unzip recsys.zip
```

```
Saving to: ‘recsys.zip’

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

2023-01-01 09:21:52 (307 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/
   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
```

```
from recsys.datasets import mlLatestSmall

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

```
ratings, movies = mlLatestSmall.load()
```

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

Section 1.2: Data visualisation

In this section we can see the data that we will analyze

```
ratings.head()
```

	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

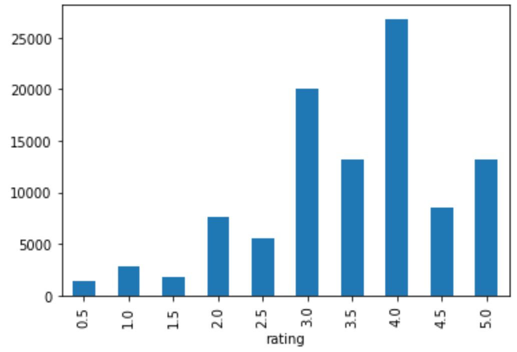
```
movies.head()
```

	itemid	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance

1.2.1. Histogram of ratings

```
ratings.groupby('rating').size().plot(kind='bar')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9959dbba00>
```

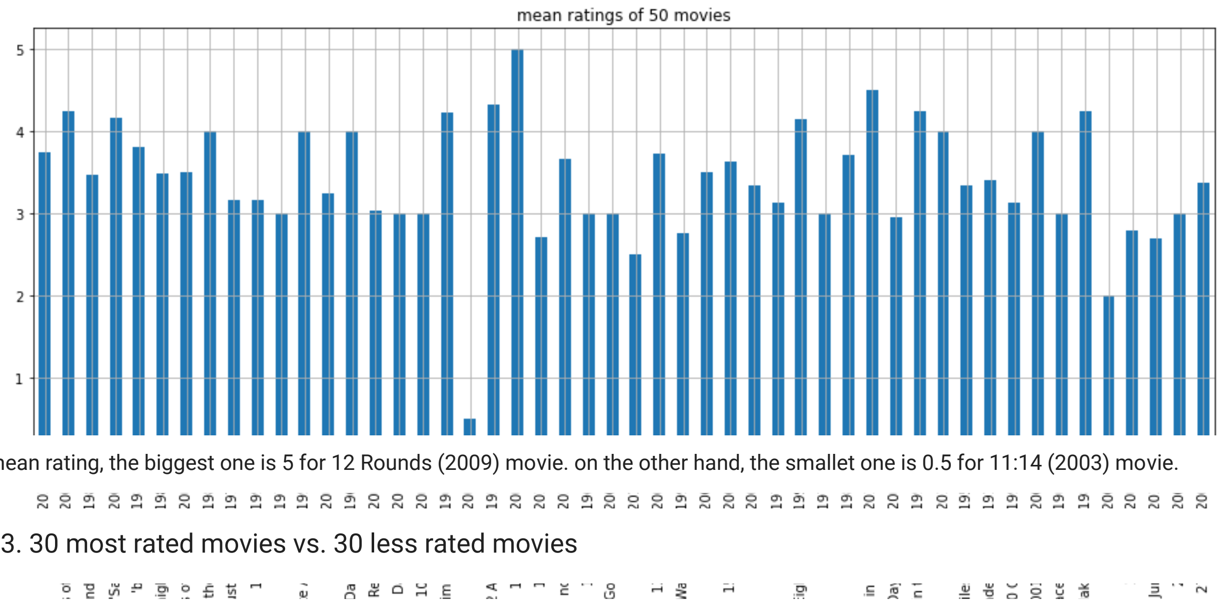


As we can see from the histogram above that the most number of ratings was for 4.0 and the least one for 0.5.

1.2.2. Average ratings of movies

```
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")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9959caa1c0>
```

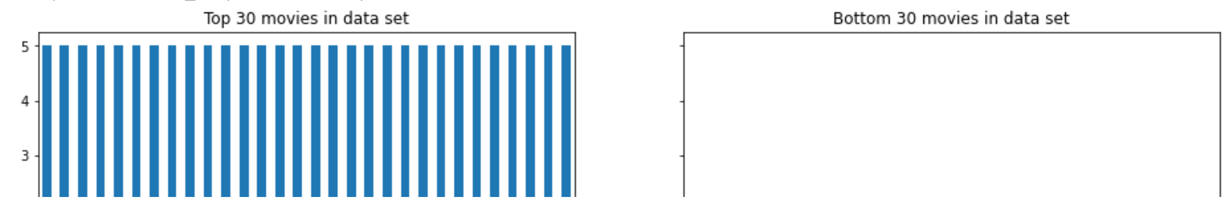


As mean rating, the biggest one is 5 for 12 Rounds (2009) movie. on the other hand, the smallet one is 0.5 for 11:14 (2003) movie.

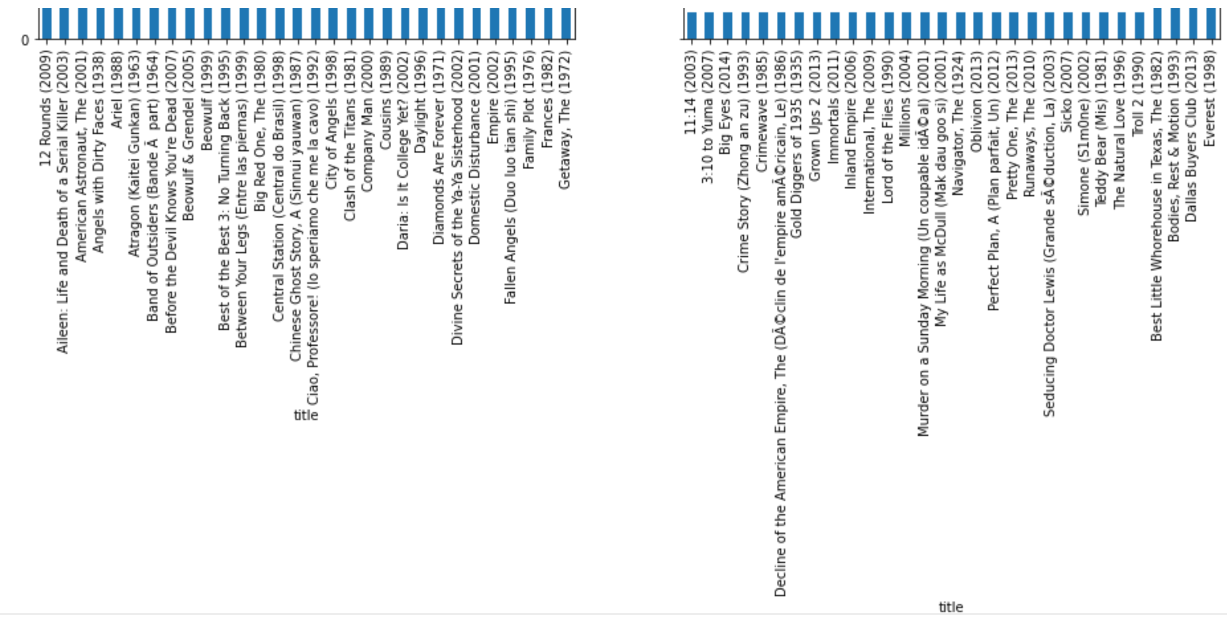
1.2.3. 30 most rated movies vs. 30 less rated movies

```
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")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9957e8e880>
```



we can see from the first plot the 30 most rated movies which all of their rating is 5, while the second plot shows the 30 least ratedd movies with rating less than or equat 1.



Chapter 2: Memory-based Collaborative Filtering

Section 2.1: Introduction

Chapter two talks about Memory-based Collaborative Filtering. There is two main algorithms to do it:

1. User-based (or user to user) Collaborative Filtering : implements user-based collaborative filtering.
2. Item-based (or item to item) Collaborative Filtering : implements item-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

we will explain each algorithm in the next two sections.

Section 2.2: User-based Collaborative Filtering

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

Algorithm: user-to-user collaborative filtering

1. Identify G_u, the set of k users similar to an active user u
2. Find candidate items
3. Rating prediction
4. Top-N recommendation

2.2.1. First we prepare the tool that we will use it.

Download tools

```
import os

if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/master/recsys.zip
    !unzip recsys.zip
```

```
--2023-01-01 11:17:14--  https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/master/recsys.zip
Resolving github.com (github.com)... 20.27.177.113
Connecting to github.com (github.com)|20.27.177.113|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/nzhinusoftcm/review-on-collaborative-filtering/master/recsys.zip [follow
--2023-01-01 11:17:14--  https://raw.githubusercontent.com/nzhinusoftcm/review-on-collaborative-filtering/master/rec
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
```

```
2023-01-01 11:17:16 (111 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/
```

Import requirements

```
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
```

Load MovieLen ratings

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

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

userids and itemids encoding

this create the encoder

```
ratings, uencoder, iencoder = ids_encoder(ratings)
```

Transform rating dataframe to matrix

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

```
R = ratings_matrix(ratings)
```

2.2.2. Algorithm Steps

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.

```
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
```

```
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:]
```

Now, 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

```
model = create_model(rating_matrix=R, metric='cosine') # we can also use the 'euclidian' distance
similarities, neighbors = nearest_neighbors(R, model)
```

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

```
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
    """
    user_neighbors = neighbors[userid]
    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=Fal
    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.

```
# 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]


np_ratings = mean_ratings.to_numpy()
```

Let us define function `predict` that predict rating between user u and

1. List item
2. List item

```
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.

```
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)


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)


user2userCF()
```

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

As we see here, the progress of rating predictions is so satisfying.

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.

```
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


user2userRecommendation(212)
```

	userid	itemid	predicted_rating	title
0	212	483	4.871495	Casablanca (1942)
1	212	357	4.764547	One Flew Over the Cuckoo's Nest (1975)
2	212	50	4.660002	Star Wars (1977)
3	212	98	4.613636	Silence of the Lambs, The (1991)
4	212	64	4.550733	Shawshank Redemption, The (1994)
5	212	194	4.522336	Sting, The (1973)
6	212	174	4.521300	Raiders of the Lost Ark (1981)
7	212	134	4.414819	Citizen Kane (1941)
8	212	187	4.344531	Godfather: Part II, The (1974)
9	212	196	4.303696	Dead Poets Society (1989)
10	212	523	4.281802	Cool Hand Luke (1967)
11	212	216	4.278246	When Harry Met Sally... (1989)
12	212	100	4.260087	Fargo (1996)
13	212	168	4.206139	Monty Python and the Holy Grail (1974)
14	212	435	4.122984	Butch Cassidy and the Sundance Kid (1969)
15	212	135	4.115228	2001: A Space Odyssey (1968)
16	212	83	4.106995	Much Ado About Nothing (1993)
17	212	69	4.086366	Forrest Gump (1994)
18	212	70	4.086328	Four Weddings and a Funeral (1994)
19	212	275	3.985037	Sense and Sensibility (1995)
20	212	153	3.981619	Fish Called Wanda, A (1988)
21	212	514	3.956640	Annie Hall (1977)

So, the rating predection for the user that have ID (212) is between [3, 4.9].

2.2.3. Evaluation with Mean Absolute Error (MAE)

First we will evaluate the model on test data, then we calculate the mean absolute error (**MAE**) to evaluate the performance of a predictive model

28 212 202 3.508617 Groundhog Day (1993)

```
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


evaluate(x_test, y_test)
```

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

MAE : 0.7505910931068639
0.7505910931068639
```

This means that on average, the model's predictions are off by 0.7505910931068639 units.

Section 2.3: Item-based Collaborative Filtering

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

Algorithm: item-to-item collaborative filtering

1. Find similarities for each of the items

2. Top N recommendation for a given user

   a- Finding candidate items

   b- Find similarity between each candidate item and the set I_u

   c- Rank candidate items according to their similarities to I_u

2.3.1. First we prepare the tool that we will use it.

Import useful requirements

```
import os

if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/master/recsys.zip
    !unzip recsys.zip
```

Import requirements

```
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
```

Load ratings

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

userids and itemids encoding

```
# create the encoder
ratings, uencoder, iencoder = ids_encoder(ratings)
```

Let's now implements the item-based collaborative filtering algorithm described above

2.3.2. Algorithm Steps

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

```
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
```

```
mean, norm_ratings = normalize()
np_ratings = norm_ratings.to_numpy()
norm_ratings.head()
```

	userid	itemid	rating	rating_mean	norm_rating
0	0	0	5	3.610294	1.389706

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

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

```
R = item_representation(norm_ratings)
```

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

```
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.

```
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:]
```

Ajusted 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. 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.

```
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.

uncomment the two lines of the following cell to compute the adjusted cosine between all items. As we have already run the next cell before, we will just load the precomputed similarities for further use.

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

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 initialise 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.

```
# metric : choose among [cosine, euclidean, adjusted_cosine]

metric = '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)


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

Step 2. Top N recommendation for a given user

Top-N recommendations are made for example for a user u who has already rated a set of items I_u

2.a- 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()`

```
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


test_user = uencoder.transform([1])[0]
i_u, u_candidates = candidate_items(test_user)


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
```

2.b- Find similarity between each candidate item and the set I_u

```
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
```

2.c- Rank candidate items according to their similarities to I_u

```
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
```

2.3.3. Putting all together

Now that we defined all functions necessary to build our item to item top-N recommendation, let's define function `item2item_topN()` that makes top-N recommendations for a given user

```
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
```

```
topn_recommendation(test_user)
```

	itemid	similarity_with_Iu	title
0	1356	52.867173	Ed's Next Move (1996)
1	1189	50.362199	Prefontaine (1997)
2	1516	31.133267	Wedding Gift, The (1994)
3	1550	31.031738	Destiny Turns on the Radio (1995)
4	1554	27.364494	Safe Passage (1994)
5	1600	27.287712	Guantanamera (1994)
6	1223	26.631850	King of the Hill (1993)
7	1388	26.624397	Gabbeh (1996)
8	766	26.590175	Man of the Year (1995)
9	691	26.461802	Dark City (1998)
10	1378	25.787842	Rhyme & Reason (1997)
11	1664	25.327445	8 Heads in a Duffel Bag (1997)
12	1261	24.785660	Run of the Country, The (1995)
13	1123	24.524028	Last Time I Saw Paris, The (1954)
14	1538	24.492453	All Over Me (1997)
15	1485	24.345312	Colonel Chabert, Le (1994)
16	1450	24.262120	Golden Earrings (1947)
17	909	23.357301	Dangerous Beauty (1998)
18	359	22.973658	Assignment, The (1997)
19	1369	22.710078	Forbidden Christ, The (Cristo proibito, Il) (1...
20	1506	22.325504	Nelly & Monsieur Arnaud (1995)
21	1537	22.061914	Cosi (1996)

This dataframe represents the top 30 recommendation list for the test 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.

2.3.4. Top N recommendation with 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

```
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

```
def topn_prediction(userid):
    """
    :param userid : id of the active user
    :return topn : initial topN recommendations returned by the function item2item_topN
    :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 user 1 and compare the two list

```
topn, topn_predict = topn_prediction(userid=test_user)
```

```
topn_predict
```

	itemid	similarity_with_Iu	title	prediction
7	1388	26.624397	Gabbeh (1996)	4.666667
18	359	22.973658	Assignment, The (1997)	4.600000
4	1554	27.364494	Safe Passage (1994)	4.500000
14	1538	24.492453	All Over Me (1997)	4.500000
27	1448	20.846909	My Favorite Season (1993)	4.490052
29	1375	20.627152	Cement Garden, The (1993)	4.333333
26	1466	21.063269	Margaret's Museum (1995)	4.271915
2	1516	31.133267	Wedding Gift, The (1994)	4.000000
23	1467	21.861203	Saint of Fort Washington, The (1993)	4.000000
21	1537	22.061914	Cosi (1996)	4.000000
10	1378	25.787842	Rhyme & Reason (1997)	4.000000
19	1369	22.710078	Forbidden Christ, The (Cristo proibito, Il) (1...	4.000000
3	1550	31.031738	Destiny Turns on the Radio (1995)	3.777778

As we 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.

11	1664	25.327445	8 Heads in a Duffel Bag (1997)	3.610294

Evaluation with Mean Absolute Error

6	1223	26.631850	King of the Hill (1993)	3.610294

```
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

evaluate(x_test, y_test)
```

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

MAE : 0.672389703640273
0.672389703640273
```

This means that on average, the model's predictions are off by 0.672389703640273

Chapter 3: Dimensionality reduction

After we explained about memory-based collaborative filtering algorithms, this chapter applies some of dimensionality reduction models.

Here the explored models are :

1. Singular Value Decomposition (SVD) : implements dimensionality reduction with Singular Value Decomposition for collaborative filtering recommender systems
2. Matrix Factorization : builds and trains a Matrix Factorization based recommender system.
3. Non Negative Matrix Factorization: applying non negativity to the learnt factors of matrix factorization.
4. Explainable Matrix Factorization: add explainability to matrix factorization factors in order to improve recommendation performances.

Section 3.1: Singular Value Decomposition 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.

SVD algorithm

> 1. Factor the normalize rating matrix R_{norm} to obtain matrices P, Σ and Q
> 2. Reduce Σ to dimension k to obtain Σ_k
> 3. Compute the square-root of Σ_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.

Now let's implement the SVD collaborative filtering

3.1.1. First we prepare the tool that we will use it.

Download useful tools

```
import os

if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/master/recsys.zip
    !unzip recsys.zip
```

Import requirements

```
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
```

Loading movielen ratings

```
ratings, movies = mlLatestSmall.load()
```

Let's see how our rating matrix looks like

```
pd.crosstab(ratings.userid, ratings.itemid, ratings.rating, aggfunc=sum)
```

```
itemid    1    2    3    4    5    6    7    8    9   10  ...  193565  193567  193571  193573  193
userid
   1    4.0  NaN  4.0  NaN  NaN  4.0  NaN  NaN  NaN  NaN  ...     NaN     NaN     NaN     NaN    N
   2    NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  ...     NaN     NaN     NaN     NaN    N
   3    NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  ...     NaN     NaN     NaN     NaN    N
   4    NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  ...     NaN     NaN     NaN     NaN    N
   5    4.0  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  ...     NaN     NaN     NaN     NaN    N
  ...   ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...     ...     ...     ...     ...
  606   2.5  NaN  NaN  NaN  NaN  NaN  2.5  NaN  NaN  NaN  ...     NaN     NaN     NaN     NaN    N
  607   4.0  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  ...     NaN     NaN     NaN     NaN    N
  608   2.5  2.0  2.0  NaN  NaN  NaN  NaN  NaN  NaN  4.0  ...     NaN     NaN     NaN     NaN    N
  609   3.0  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  4.0  ...     NaN     NaN     NaN     NaN    N
  610   5.0  NaN  NaN  NaN  NaN  5.0  NaN  NaN  NaN  NaN  ...     NaN     NaN     NaN     NaN    N

610 rows × 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 subtrat from each rating the corresponding user mean to normalize the data. This helps to improve the accuracy of the model.

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


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

```
df
```

itemid	1	2	3	4	5	6	7	8	9	
userid										
1	-0.366379	-0.934561	-0.366379	-2.009236	-1.294951	-0.366379	-1.181194	-1.491379	-1.241379	-0.87
2	-0.027346	-0.516458	-0.688660	-1.591133	-0.876847	-0.002197	-0.763091	-1.073276	-0.823276	-0.45
3	1.485033	0.995921	0.823718	-0.078755	0.635531	1.510181	0.749288	0.439103	0.689103	1.06
4	0.365375	-0.123737	-0.295940	-1.198413	-0.484127	0.390523	-0.370370	-0.680556	-0.430556	-0.05
5	0.363636	-0.204545	-0.376748	-1.279221	-0.564935	0.309715	-0.451178	-0.761364	-0.511364	-0.14
...	...	...	...	...	...	...	...	...	...	
606	-1.157399	-0.225581	-0.397784	-1.300256	-0.585971	0.288679	-1.157399	-0.782399	-0.532399	-0.16
607	0.213904	-0.354278	-0.526481	-1.428953	-0.714668	0.159982	-0.600911	-0.911096	-0.661096	-0.28
608	-0.634176	-1.134176	-1.134176	-0.777033	-0.062747	0.811903	0.051009	-0.259176	-0.009176	0.86
609	-0.270270	0.161548	-0.010655	-0.913127	-0.198842	0.675808	-0.085085	-0.395270	-0.145270	0.72
610	1.311444	-0.256738	-0.428941	-1.331413	-0.617127	1.311444	-0.503371	-0.813556	-0.563556	-0.19

Ids encoding

Let's encode users and items ids such that their values range from 0 to 909 (for users) and from 0 to 9723 (for items)

```
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)
```

```
LabelEncoder()
```

3.1.2. 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 ratin 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.

```
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.

```
# create our svd model
svd = SVD(umean)

# fit our model with normalized ratings
svd.fit(R)
```

Rating prediction

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

```
ratings.head(10)
```

	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 sens. We will make predictions for user 1 on the 10 items listed above.

```
# 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
```

The prediction error is less than 0.00001

Make recommendations

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

```
userid = 1

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

##
# Now let's exclud 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
```

	itemid	predictions	title	genres
0	148	5.0	Awfully Big Adventure, An (1995)	Drama
1	6086	5.0	I, the Jury (1982)	Crime|Drama|Thriller
2	136445	5.0	George Carlin: Back in Town (1996)	Comedy
3	6201	5.0	Lady Jane (1986)	Drama|Romance
4	2075	5.0	Mephisto (1981)	Drama|War
5	6192	5.0	Open Hearts (Elsker dig for evigt) (2002)	Romance
6	117531	5.0	Watermark (2014)	Documentary
7	158398	5.0	World of Glory (1991)	Comedy
8	6021	5.0	American Friend, The (Amerikanische Freund, De...	Crime|Drama|Mystery|Thriller
9	136556	5.0	Kung Fu Panda: Secrets of the Masters (2011)	Animation|Children
10	136447	5.0	George Carlin: You Are All Diseased (1999)	Comedy
11	136503	5.0	Tom and Jerry: Shiver Me Whiskers (2006)	Animation|Children|Comedy
12	134095	5.0	My Love (2006)	Animation|Drama
13	3851	5.0	I'm the One That I Want (2000)	Comedy
14	136469	5.0	Larry David: Curb Your Enthusiasm (1999)	Comedy

The first 30 items have an equivalent rating prediction for the user 1

Section 3.2: Matrix Factorization

19 158027 5.0 SORI: Voice from the Heart (2016) Drama|Sci-Fi

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.

22 118894 5.0 Scooby-Doo! Abracadabra-Doo (2010) Animation|Children|Mystery

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.

25 3940 5.0 Slumber Party Massacre III (1990) Horror

Matrix Factorization : algorithm

1. Initialize P and Q with random values
2. 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:
  - $p_u \leftarrow p_u + \alpha \cdot (e_{u,i} \cdot q_i - \lambda \cdot p_u)$
  - $q_i \leftarrow q_i + \alpha \cdot (e_{u,i} \cdot p_u - \lambda \cdot q_i)$

3. Repeat step 2 until the optimal parameters are reached.

3.2.1. First we prepare the tool that we will use it.

Download useful files

```
import os
if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/master/recsys.zip
    !unzip recsys.zip
```

Import requirements

```
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
```

3.2.2. Model definition

```
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
    epochs = 10
```

3.2.3. MovieLens 100k

Evaluation on raw ratings

```
# load the ml100k dataset
ratings, movies = ml100k.load()

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 normalize ratings
raw_examples, raw_labels = get_examples(ratings)

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


# 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
```

```
MF.evaluate(x_test, y_test)
```

```
validation error : 1.497
1.4973507972141993
```

If the validation error is 1.497, it means that on average, the model's predictions are off by 1.497 units.

Evaluation on normalized ratings

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

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 substracting 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 normalize ratings
raw_examples, raw_labels = get_examples(ratings, labels_column=normalized_column_name)

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


# 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
```

```
MF.evaluate(x_test, y_test)
```

```
    validation error : 0.828
    0.8276982643684648
```

If the validation error is 0.828, it means that on average, the model's predictions are off by 0.828 units.

3.2.4. MovieLens 1M

Evaluation on raw data

```
[24] # load the ml1m dataset
     ratings, movies = ml1m.load()

     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 normalize ratings
     raw_examples, raw_labels = get_examples(ratings)

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

```
[25] # 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
```

```
[26] MF.evaluate(x_test, y_test)
```

```
validation error : 1.482
1.4820034560467208
```

If the validation error is 1.482, it means that on average, the model's predictions are off by 1.482 units.

Evaluation on normalized ratings

```
# 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
```

```
MF.evaluate(x_test, y_test)
```

```
validation error : 0.828
0.8276982643684648
```

If the validation error is 0.825, it means that on average, the model's predictions are off by 0.825 units.

3.2.5. 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 1 on item 1 for which the truth rating $r = 5.0$

```
ratings.userid = uencoder.inverse_transform(ratings.userid.to_list())
ratings.itemid = uencoder.inverse_transform(ratings.itemid.to_list())
ratings.head(5)
```

	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

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

```
4.188679163563357
```

So, our rating prediction for user 1 on item 1 for which the truth rating $r = 5.0$ is 4.2

Section 3.3: 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 $R = PQ^\top$.

One limitation of Matrix Factorization

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

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.

Objective function

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

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

The goal is to minimize the cost function J by optimizing parameters P and Q, with λ_P and λ_Q the regularizer parameters.

3.3.1. First we prepare the tool that we will use it.

Install and import useful packages

```
import os

if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/master/recsys.zip
    !unzip recsys.zip


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
```

Load and preprocess rating

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

# prepare data
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 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)
```

3.3.2. Non-negative Matrix Factorization Model

```
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
        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
```

3.3.3. Train the NMF model with ML-100K dataset

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).

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

# create and train the model
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
```

```
nmf.evaluate(x_test, y_test)
```

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

If the validation error is 0.917, it means that on average, the model's predictions are off by 0.917 units.

3.3.4. Evaluation of NMF with Scikit-suprise

We can use the scikt-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.

```
!pip install scikit-surprise
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting scikit-surprise
  Downloading scikit-surprise-1.1.3.tar.gz (771 kB)
     |████████████████████████████████| 771 kB 5.2 MB/s
Requirement already satisfied: joblib>=1.0.0 in /usr/local/lib/python3.8/dist-packages (from scikit-surprise) (1.2.0)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.8/dist-packages (from scikit-surprise) (1.21.6)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.8/dist-packages (from scikit-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=2626504 sha256=230c
  Stored in directory: /root/.cache/pip/wheels/af/db/86/2c18183a80ba05da35bf0fb7417aac5cddbd93bcb1b92fd3ea
Successfully built scikit-surprise
Installing collected packages: scikit-surprise
Successfully installed scikit-surprise-1.1.3
```

```
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)
```

```
Evaluating MAE of algorithm NMF on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
MAE (testset)     0.9584  0.9640  0.9456  0.9566  0.9481  0.9545  0.0068
Fit time          0.50    0.48    0.49    0.49    0.50    0.49    0.01
Test time         0.34    0.24    0.23    0.15    0.23    0.24    0.06
```

As result, the mean MAE on the test set in Folder 5 is **MAE = 0.9481** which is equivalent to the result we have obtained on *ml-100k* with our own implementation **mae = 0.9165**

ML-1M

```
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)     0.9553  0.9546  0.9598  0.9582  0.9546  0.9565  0.0021  
Fit time          3.88    4.13    4.40    4.26    4.36    4.21    0.19    
Test time         2.61    2.39    2.84    2.13    2.73    2.54    0.25    
```

The mean MAE on a 5-fold cross-validation is **MAE = 0.9546**

Section 3.4: Explainable Matrix Factorization (EMF)

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

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:

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

here, $\frac{\beta}{2}(||P_u||^2 + ||Q_i||^2)$ is the L_2 regularization term weighted by the coefficient β, and λ 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.

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

3.4.1. First we prepare the tool that we will use it.

```
import os

if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/master/recsys.zip
    !unzip recsys.zip


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
```

3.4.2. Compute Explainable Scores

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

```
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
```

3.4.3. Explainable Matrix Factorization Model

```
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]
```

```
epochs = 10
```

3.4.4. Model Evaluation

MovieLens 100K

Evaluation on raw data

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

# encode users and items ids
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 normalize ratings
raw_examples, raw_labels = get_examples(ratings)

# train test split
(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)
```

```
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%
```

```
# initialize the model
EMF = ExplainableMatrixFactorization(m, n, W, alpha=0.01, beta=0.4, lamb=0.01, k=10)

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
```

```
EMF.evaluate(x_test, y_test)
```

```
validation error : 0.797
```

If the validation error is 0.797, it means that on average, the model's predictions are off by 0.797 units.

Evaluation on normalized data

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

# encode users and items ids
ratings, uencoder, iencoder = ids_encoder(ratings)

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

m = len(users)
n = len(items)

# normalize ratings by substracting 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 normalize ratings
raw_examples, raw_labels = get_examples(ratings, labels_column=normalized_column_name)

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


# initialize the model
EMF = ExplainableMatrixFactorization(m, n, W, alpha=0.022, beta=0.65, lamb=0.01, k=10)

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
```

MovieLens 1M

Evaluation on raw data

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

# encode users and items ids
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 normalize ratings
raw_examples, raw_labels = get_examples(ratings)

# train test split
(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)
```

```
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%
```

```
# initialize the model
EMF = ExplainableMatrixFactorization(m, n, W, alpha=0.01, beta=0.4, lamb=0.01, k=10)

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
```

Evaluation on normalized data

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

# encode users and items ids
ratings, uencoder, iencoder = ids_encoder(ratings)

# normalize ratings by substracting 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 normalize ratings
raw_examples, raw_labels = get_examples(ratings, labels_column=normalized_column_name)
```

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

# initialize the model
EMF = ExplainableMatrixFactorization(m, n, W, alpha=0.023, beta=0.59, lamb=0.01, k=10)

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
```

Ratings prediction

```
# 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
```

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

22	2905	4.224974	Sanjuro (1962)	Action|Adventure

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.

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

Chapter 4: Performances comparison

This chapter presents an overall performance comparaison of all the models listed before:

1. User-based CF
2. Item-based CF
3. Singular Value Decomposition (SVD)
4. Matrix Factorization (MF)
5. Non-negative Matrix Factorization (NMF)
6. Explainable Matrix Factorization (EMF)

Section 4.1: First we prepare all the requirements tool

```
import os

if not (os.path.exists("recsys.zip") or os.path.exists("recsys")):
    !wget https://github.com/nzhinusoftcm/review-on-collaborative-filtering/raw/master/recsys.zip
    !unzip recsys.zip
```

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

2023-01-01 17:26:22 (129 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/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
```

```
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
```

Section 4.2: Results on MovieLens 100k

4.2.1. User-based CF

```
# 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)

# evaluate with Euclidean distance
usertouser = UserToUser(ratings, movies, metric='euclidean')
print("==========================")
usertouser.evaluate(x_test, y_test)
```

```
Download data 100.2%
Successfully downloaded ml-100k.zip 4924029 bytes.
Unzipping the ml-100k.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 10000 test data ...

MAE : 0.8125945111976461
0.8125945111976461
```

```
# 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
0.7505910931068639
```

4.2.2. Item-based CF

```
# 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)

# 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
0.507794195659005
```

Evaluation with Euclidean distance

```
# 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
0.8277111416143341
```

4.2.3. Matrix Factorization

```
epochs = 10
```

```
# load the ml100k dataset
ratings, movies = ml100k.load()

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 normalize ratings
raw_examples, raw_labels = get_examples(ratings)

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

# create and train 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))
```

```
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
```

4.2.4. Non-negative Matrix Factorization

```
!pip install scikit-surprise
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting scikit-surprise
  Downloading scikit-surprise-1.1.3.tar.gz (771 kB)
     |████████████████████████████████| 771 kB 4.8 MB/s
Requirement already satisfied: joblib>=1.0.0 in /usr/local/lib/python3.8/dist-packages (from scikit-surprise) (1.2.0)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.8/dist-packages (from scikit-surprise) (1.21.6)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.8/dist-packages (from scikit-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=2626490 sha256=deb62416ec1cef38dcba2b8387927783e4f1d37cf3bbcbed9f55aa217ff0da72
  Stored in directory: /root/.cache/pip/wheels/af/db/86/2c18183a80ba05da35bf0fb7417aac5cddbd93bcb1b92fd3ea
Successfully built scikit-surprise
Installing collected packages: scikit-surprise
Successfully installed scikit-surprise-1.1.3
```

```
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          0.95    0.53    0.52    0.54    0.52    0.61    0.17
Test time         0.38    0.30    0.15    0.23    0.14    0.24    0.09
```

4.2.4. Explainable Matrix Factorization

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

# encode users and items ids
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 normalize ratings
raw_examples, raw_labels = get_examples(ratings)

# train test split
(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("===================")
# initialize the model
emf = EMF(m, n, W, alpha=0.01, beta=0.4, lamb=0.01, k=10)

history = emf.fit(x_train, y_train, epochs=epochs, validation_data=(x_test, y_test))

print("===================")
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
0.797347824723284
```

Section 4.3: Results on MovieLens 1M (ML-1M)

4.3.1. User-based CF

```
# 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)

# metric : cosine

# 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
0.732267005840993
```

```
# metric : euclidean

# create the user-based CF
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
0.8069332535426614
```

4.3.2. Item-based CF

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 100021 test data ...

MAE : 0.42514728655396045
0.42514728655396045
```

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 100021 test data ...

MAE : 0.82502173206615
0.82502173206615
```

4.3.3. Matrix Factorization

```
# load the ml1m dataset
ratings, movies = ml1m.load()

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 normalize ratings
raw_examples, raw_labels = get_examples(ratings)

# train test split
(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("==================")
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
    1.4820034560467208
```

4.3.4. Non-negative Matrix Factorization

```
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          5.06    5.69    5.76    5.87    5.86    5.65    0.30
    Test time         1.86    3.17    3.44    2.61    3.42    2.90    0.60
```

4.3.5. Explainable Matrix Factorization

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

# encode users and items ids
```

```
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 normalize ratings
raw_examples, raw_labels = get_examples(ratings)

# train test split
(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)

# initialize the model
emf = EMF(m, n, W, alpha=0.01, beta=0.4, lamb=0.01, k=10)

history = emf.fit(x_train, y_train, epochs=epochs, validation_data=(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
```

Section 4.4: 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

Preprocessing	Dataset	MF	NMF	EMF
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

In general, a lower mean absolute error (MAE) is better than a higher MAE, because it means that the predictions of the model are closer to the true values.