Recommender Systems - notebook 1 - Traditional Approaches

In [ ]:
```
"""
(Practical tip) Table of contents can be compiled directly in jupyter noteb
I set an exception: if the package is in your installation you can import i
then import it.
"""
try:
    from jyquickhelper import add_notebook_menu
except:
    !pip install jyquickhelper
    from jyquickhelper import add_notebook_menu

"""
Output Table of contents to navigate easily in the notebook.
For interested readers, the package also includes Ipython magic commands to
wherever you are in the notebook to look for cells faster
"""
add_notebook_menu()
```

Out[ ]: run previous cell, wait for 2 seconds

Imports

In [ ]:
```
import ssl

ssl._create_default_https_context = ssl._create_unverified_context
```

In [ ]:
```
from tqdm import tqdm
import pandas as pd
import numpy as np
```

Dataset description

We use here the MovieLens 100K Dataset. It contain 100,000 ratings from 1000 users on 1700 movies.

- u.train / u.test part of the original u.data information
  - The full u data set, 100000 ratings by 943 users on 1682 items. Each user has rated at least 20 movies. Users and items are numbered consecutively from 1. The data is randomly ordered. This is a tab separated list of user id | item id | rating | timestamp. The time stamps are unix seconds since 1/1/1970 UTC
- u.info
  - The number of users, items, and ratings in the u data set.
- u.item
  - Information about the items (movies); this is a tab separated list of movie id | movie title | release date | video release date | IMDb URL | unknown | Action | Adventure | Animation | Children's | Comedy | Crime | Documentary | Drama | Fantasy | Film-

Processing math: 100%

Noir | Horror | Musical | Mystery | Romance | Sci-Fi | Thriller | War | Western | The last 19 fields are the genres, a 1 indicates the movie is of that genre, a 0 indicates it is not; movies can be in several genres at once. The movie ids are the ones used in the u.data data set.

- u.genre
  - A list of the genres.
- u.user
  - Demographic information about the users; this is a tab separated list of user id | age | gender | occupation | zip code The user ids are the ones used in the u.data data set.

In [ ]:
```python
path = "https://www.i3s.unice.fr/~riveill/dataset/dataset_movilens_100K/"
```

Before we build our model, it is important to understand the distinction between implicit and explicit feedback, and why modern recommender systems are built on implicit feedback.

- **Explicit Feedback:** in the context of recommender systems, explicit feedback are direct and quantitative data collected from users.
- **Implicit Feedback:** on the other hand, implicit feedback are collected indirectly from user interactions, and they act as a proxy for user preference.

The advantage of implicit feedback is that it is abundant. Recommender systems built using implicit feedback allow recommendations to be adapted in real time, with each click and interaction.

Today, online recommender systems are built using implicit feedback.

Data preprocessing

In [ ]:
```python
# Load data
np.random.seed(123)

ratings = pd.read_csv(
    path + "u.data",
    sep="\t",
    header=None,
    names=["userId", "movieId", "rating", "timestamp"],
)
ratings = ratings.sort_values(["timestamp"], ascending=True)
print("Nb ratings:", len(ratings))
ratings
```

```
Nb ratings: 100000
```

Processing math: 100%

Out[ ]:

	userId	movieId	rating	timestamp
214	259	255	4	874724710
83965	259	286	4	874724727
43027	259	298	4	874724754
21396	259	185	4	874724781
82655	259	173	4	874724843
...	...	...	...	...
46773	729	689	4	893286638
73008	729	313	3	893286638
46574	729	328	3	893286638
64312	729	748	4	893286638
79208	729	272	4	893286638

100000 rows × 4 columns

Data splitting

Separating the dataset between train and test in a random fashion would not be fair, as we could potentially use a user's recent evaluations for training and previous evaluations. This introduces a data leakage with an anticipation bias, and the performance of the trained model would not be generalizable to real world performance.

Therefore, we need to slice the train and test based on the timestamp

In [ ]:

```
# Split dataset
train_ratings, test_ratings = np.split(ratings, [int(0.9 * len(ratings))])

max(train_ratings["timestamp"]) <= min(test_ratings["timestamp"])
```

Out[ ]:
```
True
```

In [ ]:

```
# drop columns that we no longer need
train_ratings = train_ratings[["userId", "movieId", "rating"]]
test_ratings = test_ratings[["userId", "movieId", "rating"]]

len(train_ratings), len(test_ratings)
```

Out[ ]:
```
(90000, 10000)
```

In [ ]:

```
# Get a list of all movie IDs
all_movieIds = ratings["movieId"].unique()
```

Build pivot table

In [ ]:

```
""" Pivot table for train set """
train_pivot = pd.pivot_table(
    data=train_ratings,
    values="rating",
```

Processing math: 100%

```
    index="userId",
    columns="movieId",
)
print("Nb users: ", train_pivot.shape[0])
print("Nb movies:", train_pivot.shape[1])
train_pivot
```

```
Nb users:  867
Nb movies: 1637
```

Out[ ]:

movieId	1	2	3	4	5	6	7	8	9	10	...	1673	1674	1675	167
userId															
1	5.0	3.0	4.0	3.0	3.0	5.0	4.0	1.0	5.0	3.0	...	NaN	NaN	NaN	Na
2	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.0	...	NaN	NaN	NaN	Na
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	Na
5	4.0	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	Na
6	4.0	NaN	NaN	NaN	NaN	NaN	2.0	4.0	4.0	NaN	...	NaN	NaN	NaN	Na
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	.
939	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	5.0	NaN	...	NaN	NaN	NaN	Na
940	NaN	NaN	NaN	2.0	NaN	NaN	4.0	5.0	3.0	NaN	...	NaN	NaN	NaN	Na
941	5.0	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	...	NaN	NaN	NaN	Na
942	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	Na
943	NaN	5.0	NaN	NaN	NaN	NaN	NaN	NaN	3.0	NaN	...	NaN	NaN	NaN	Na

867 rows × 1637 columns

In [ ]:

```
train_users = train_pivot.index
train_movies = train_pivot.columns
```

Collaborative filtering based on Users similarity

This approach uses scores that have been assigned by other users to calculate predictions.

In pivot table

- Rows are users, u, v
- Columns are items, i, j

pred(u, i) = $\sum_v sim(u, v) * r_{v,i} \sum_v sim(u, v)$

Wich similarity function:

- Euclidean distance [0, 1]: sim(a, b) = $11+\sqrt{\sum_i (r_{a,i}-r_{b,i})^2}$
- Pearson correlation [− 1, 1]: sim(a, b) = $\sum_i (r_{a,i}-r_a)(r_{b,i}-r_b) \sum_i (r_{a,i}-r_a)^2 \sum_i (r_{b,i}-r_b)^2$
- Cosine similarity [− 1, 1]: sim(a, b) = $a.b |a|.|b|$

Processing math: 100%

Which function should we use? The answer is that there is no fixed recipe; but there are some issues we can take into account when choosing the proper similarity function. On the one hand:

- Pearson correlation usually works better than Euclidean distance since it is based more on the ranking than on the values. So, two users who usually like more the same set of items, although their rating is on different scales, will come out as similar users with Pearson correlation but not with Euclidean distance.
- On the other hand, when dealing with binary/unary data, i.e., like versus not like or buy versus not buy, instead of scalar or real data like ratings, cosine distance is usually used.

Build predictor

In [ ]:

```
# Step 1: build the similarity matrix between users
correlation_matrix = train_pivot.transpose().corr("pearson")
correlation_matrix
```

Out[ ]:

userId	1	2	3	5	6	7	8	
userId								
1	1.000000	1.608412e-01	0.112780	0.420809	0.287159	1.237128e-01	0.692086	-0.10
2	0.160841	1.000000e+00	0.067420	0.327327	0.446269	4.807341e-01	0.585491	0.24
3	0.112780	6.741999e-02	1.000000	NaN	-0.109109	-5.037555e-17	0.291937	
5	0.420809	3.273268e-01	NaN	1.000000	0.241817	1.490373e-01	0.537400	0.57
6	0.287159	4.462695e-01	-0.109109	0.241817	1.000000	1.758687e-01	0.687745	0.13
...	...	...	...	...	...	...	...	
939	0.534390	-7.671236e-18	NaN	0.880705	0.206315	1.425665e-01	-0.333333	
940	0.263289	-1.168173e-02	-0.104678	0.027038	-0.024419	3.142734e-02	0.320487	0.17
941	0.205616	-6.201737e-02	1.000000	0.468521	0.399186	0.000000e+00	0.166667	1.00
942	-0.180784	8.596024e-02	-0.011792	0.318163	0.092349	4.548076e-01	0.201328	0.70
943	0.067549	4.797016e-01	NaN	0.346234	0.109833	3.534118e-01	0.040741	0.92

867 rows × 867 columns

In [ ]:

```
# Step2 build rating function
# We want to calculate the rating that a user could have given for an item.

# Il est plus efficace de travailler avec numpy qu'avec pandas.
# On transforme donc la matrice pivot en numpy
pivot = train_pivot.to_numpy()
# idem pour la matrice de correlation
corr = correlation_matrix.to_numpy()
# Malheureusement, on doit utiliser 2 dictionnaires pour passer
nom de la colonne movieId dans son indice en numpy
```

```python
movie2column = {j: i for i, j in enumerate(train_pivot.columns)}
# Du nom de la ligne userId dans son indice en numpy
user2row = {j: i for i, j in enumerate(train_pivot.index)}


def predict(pivot, corr, userId, movieId):
    if movieId in movie2column.keys():
        movie = movie2column[movieId]
    else:
        return 2.5
    if userId in user2row.keys():
        user = user2row[userId]
    else:
        return 2.5

    # Normalement le rating est inconnu
    if np.isnan(pivot[user, movie]):
        num = 0
        den = 0
        for u in range(len(corr)):
            if not np.isnan(pivot[u, movie]) and not np.isnan(corr[user, u]
                # Si l'utilisateur u a déjà vu le film movie
                # et si les deux utilisateurs ont au moins vu un même film
                den += abs(corr[user, u])
                num += corr[user, u] * pivot[u, movie]
        if den != 0:
            return num / den
        else:
            return 2.5  # default value
    else:
        # le film a déjà été vu
        print(
            f"l'utilisateur {userId} a déjà vu le film {movieId}",
            f"et lui a donné la note de {pivot[user, movie]}",
        )
        return pivot[user, movie]


predict(pivot=pivot, corr=corr, userId=1, movieId=1)
predict(pivot=pivot, corr=corr, userId=3, movieId=28)
```

```
l'utilisateur 1 a déjà vu le film 1 et lui a donné la note de 5.0
```

Out[ ]:

```
1.8527972301545377
```

Predict

In [ ]:

```python
# Step 3 add the predicted rating to the test set

test_ratings["User based"] = [
    predict(pivot, corr, userId, movieId)
    for _, userId, movieId, _ in tqdm(
        test_ratings[["userId", "movieId", "rating"]].itertuples()
    )
]
test_ratings
```

```
10000it [00:06, 1578.53it/s]
```

Processing math: 100%

Out[ ]:

	userId	movieId	rating	User based
557	90	900	4	0.657995
6675	90	269	5	0.987827
562	90	289	3	0.572064
62660	90	270	4	1.077609
68756	90	268	4	1.202310
...	...	...	...	...
46773	729	689	4	2.500000
73008	729	313	3	2.500000
46574	729	328	3	2.500000
64312	729	748	4	2.500000
79208	729	272	4	2.500000

10000 rows × 4 columns

Evaluate the predictor

Now that we have trained our model, assigned a value to each pair (userId, movieId), we are ready to evaluate it.

Evaluation with classical metrics: RMSE

In traditional machine learning projects, we evaluate our models using measures such as accuracy (for classification problems) and RMSE (for regression problems). This is what we will do in the first instance.

In [ ]:

```
test_ratings["rating"]
```

Out[ ]:

```
557      4
6675     5
562      3
62660    4
68756    4
        ..
46773    4
73008    3
46574    3
64312    4
79208    4
Name: rating, Length: 10000, dtype: int64
```

In [ ]:

```
# Step 4 evaluate the resulte : with classical metrics
from sklearn.metrics import mean_absolute_error, mean_squared_error

print(
    "RMSE:",
    np.sqrt(mean_squared_error(test_ratings["rating"], test_ratings["User b
)
```

```
RMSE: 1.7798966446725806
```

Processing math: 100%

Hit Ratio @ K

However, a measure such as RMSE does not provide a satisfactory evaluation of recommender systems. To design a good metric for evaluating recommender systems, we need to first understand how modern recommender systems are used.

Amazon, Netflix and others uses a list of recommendations. The key here is that we don't need the user to interact with every single item in the list of recommendations. Instead, we just need the user to interact with at least one item on the list — as long as the user does that, the recommendations have worked.

To simulate this, let's run the following evaluation protocol to generate a list of top 10 recommended items for each user.

- For each user, randomly select 99 items that the user has not interacted with.
- Combine these 99 items with the test item (the actual item that the user last interacted with). We now have 100 items.
- Run the model on these 100 items, and rank them according to their predicted probabilities.
- Select the top 10 items from the list of 100 items. If the test item is present within the top 10 items, then we say that this is a hit.
- Repeat the process for all users. The Hit Ratio is then the average hits.

This evaluation protocol is known as **Hit Ratio @ K**, and it is commonly used to evaluate recommender systems.

TODO − Students

- Fill the gaps

In [ ]:

```
# Step 2 with hit ratio
def HRatio(test_ratings, predictor, K=10, predict_func=predict):
    # User-item pairs for testing
    test_user_item_set = set(
        list(set(zip(test_ratings["userId"], test_ratings["movieId"])))[:10
    )

    # Dict of all items that are interacted with by each user
    user_interacted_items = ratings.groupby("userId")["movieId"].apply(list

    hits = []
    for (u, i) in tqdm(test_user_item_set):
        interacted_items = user_interacted_items[u]
        not_interacted_items = set(all_movieIds) - set(interacted_items)
        selected_not_interacted = list(np.random.choice(list(not_interacted
        test_items = selected_not_interacted + [i]
        predicted_labels = predictor(
            pairs=[np.array([u] * 100), np.array(test_items)],
            predict_func=predict_func,  # added to be able to pass custom p
        ).reshape(-1)
        topK_items = [test_items[i] for i in np.argsort(predicted_labels)[-

        if i in topK_items:
            hits.append(1)
        else:
            hits.append(0)
    hr = np.average(hits)
```

Processing math: 100%

```
    print("The Hit Ratio @ {} is {:.2f}".format(K, hr))
    return hr
```

In [ ]:

```
def predictor(
    pairs,
    predict_func=predict,  # allows to pass custom predict functions
):
    pred = []
    for userId, movieId in zip(pairs[0], pairs[1]):
        pred += [predict_func(pivot, corr, userId, movieId)]
    return np.array(pred)


HR = dict()
hr = HRatio(
    test_ratings=test_ratings,
    predictor=predictor,
    K=25,
)
HR["User based"] = hr
```

```
100%|██████████| 1000/1000 [00:58<00:00, 17.22it/s]
The Hit Ratio @ 25 is 0.78
```

Improve the rating

Trick 1:

Since humans do not usually act the same as critics, i.e., some people usually rank movies higher or lower than others, this prediction function can be easily improved by taking into account the user mean as follows:

$$pred(u, i) = \bar{r}_u + \frac{\sum_v sim(u, v) * (r_{v,i} - \bar{r}_v)}{\sum_v sim(u, v)}$$

TODO – Students

- Modify the previous code in order to implement "Trick 1"

In [ ]:

```
def user_center(pivot):
    """
    Compute train_pivot user centered (uc), which removes the mean of every
    """
    user_mean = pivot.transpose().mean()
    return (pivot.transpose() - user_mean).transpose()


train_pivot_uc = user_center(
    pivot=train_pivot
)  # user centered version of `train_pivot`
correlation_matrix = train_pivot_uc.transpose().corr("pearson")


def predict_uc(pivot, corr, userId, movieId):
    if movieId in movie2column.keys():
        movie = movie2column[movieId]
    else:
        return 2.5
    if userId in user2row.keys():
```

```
        user = user2row[userId]
    else:
        return 2.5

    # Normalement le rating est inconnu
    if np.isnan(pivot[user, movie]):
        num = 0
        den = 0
        for u in range(len(corr)):
            if not np.isnan(pivot[u, movie]) and not np.isnan(corr[user, u]
                # Si l'utilisateur u a déjà vu le film movie
                # et si les deux utilisateurs ont au moins vu un même film
                den += abs(corr[user, u])

                # remove the mean of user rating
                num += corr[user, u] * (pivot[u, movie] - np.nanmean(pivot[
        if den != 0:
            return (num / den) + np.nanmean(pivot[user])  # add user mean
        else:
            return 2.5  # default value
    else:
        # le film a déjà été vu
        print(
            f"l'utilisateur {userId} a déjà vu le film {movieId}",
            f"et lui a donné la note de {pivot[user, movie]}",
        )
        return pivot[user, movie]


predict_uc(pivot=pivot, corr=corr, userId=1, movieId=1)
predict_uc(pivot=pivot, corr=corr, userId=3, movieId=28)
```

```
l'utilisateur 1 a déjà vu le film 1 et lui a donné la note de 5.0
```

Out[ ]:
```
2.9400162942557957
```

In [ ]:
```
# Step 3 add the predicted rating to the test set

test_ratings["User based_uc"] = [
    predict_uc(pivot, corr, userId, movieId)
    for _, userId, movieId, _ in tqdm(
        test_ratings[["userId", "movieId", "rating"]].itertuples()
    )
]
test_ratings
```

```
10000it [00:32, 308.22it/s]
```

Processing math: 100%

Out[ ]:

	userId	movieId	rating	User based	User based_uc
557	90	900	4	0.657995	3.758684
6675	90	269	5	0.987827	3.845841
562	90	289	3	0.572064	3.687360
62660	90	270	4	1.077609	3.422702
68756	90	268	4	1.202310	3.836968
...	...	...	...	...	...
46773	729	689	4	2.500000	2.500000
73008	729	313	3	2.500000	2.500000
46574	729	328	3	2.500000	2.500000
64312	729	748	4	2.500000	2.500000
79208	729	272	4	2.500000	2.500000

10000 rows × 5 columns

In [ ]:

```
# Step 4 evaluate the resulte : with classical metrics

print(
    "RMSE:",
    np.sqrt(mean_squared_error(test_ratings["rating"], test_ratings["User b
)
```

```
RMSE: 1.4531244671937105
```

In [ ]:

```
hr = HRatio(
    test_ratings=test_ratings,
    predictor=predictor,
    predict_func=predict_uc,
    K=25,
)
HR["User based_uc"] = hr
```

```
100%|██████████| 1000/1000 [02:32<00:00,  6.56it/s]
The Hit Ratio @ 25 is 0.80
```

Trick 2:

If two users have very few items in common, let us imagine that there is only one, and the rating is the same, the user similarity will be really high; however, the confidence is really small. It's possible to add a ponderation coefficient.

$\text{newsim}(a,b) = \text{sim}(a,b) * \frac{\min(N, |P_{a,b}|)}{N}$

where $|P_{a,b}|$ is the number of common items shared by user a and user b. The coefficient is < 1 if the number of common movies is $< N$ and 1 otherwise.

In [ ]:

```
# Count the number of common items shared by user 1 and user 2.
a = user2row[1]
b = user2row[2]

thresh_common_nb = 10  # N
```

Processing math: 100%

```
common = pivot[a] + pivot[b]  # sum of np arrays propagates nan
common = np.count_nonzero(~np.isnan(common))
enough_common_coeff = np.min([thresh_common_nb, common]) / thresh_common_nb
enough_common_coeff
```

Out[ ]:
```
1.0
```

TODO − Students

- Modify the previous code in order to implement "Trick 2"

In [ ]:
```
def predict_thresh(
    pivot,
    corr,
    userId,
    movieId,
    thresh_common_nb=10,  # N
):
    if movieId in movie2column.keys():
        movie = movie2column[movieId]
    else:
        return 2.5
    if userId in user2row.keys():
        user = user2row[userId]
    else:
        return 2.5

    # Normalement le rating est inconnu
    if np.isnan(pivot[user, movie]):
        num = 0
        den = 0
        for u in range(len(corr)):
            if not np.isnan(pivot[u, movie]) and not np.isnan(corr[user, u]
                # Si l'utilisateur u a déjà vu le film movie
                # et si les deux utilisateurs ont au moins vu un même film
                common = pivot[user] + pivot[u]  # sum of np arrays propaga
                common = np.count_nonzero(~np.isnan(common))
                enough_common_coeff = np.min([thresh_common_nb, common]) /

                num += enough_common_coeff * corr[user, u] * pivot[u, movie
                den += abs(enough_common_coeff * corr[user, u])

        if den != 0:
            return num / den

        else:
            return 2.5  # default value
    else:
        # le film a déjà été vu
        print(
            f"l'utilisateur {userId} a déjà vu le film {movieId}",
            f"et lui a donné la note de {pivot[user, movie]}",
        )
        return pivot[user, movie]


predict_thresh(pivot=pivot, corr=corr, userId=1, movieId=1)
predict_thresh(pivot=pivot, corr=corr, userId=3, movieId=28)
```

```
l'utilisateur 1 a déjà vu le film 1 et lui a donné la note de 5.0
207668795514
```

Processing math: 100%

In [ ]:

```
for thresh_common_nb in [10, 15, 20, 30, 50]:
    # Step 3 add the predicted rating to the test set

    test_ratings[f"User based_thresh_{thresh_common_nb}"] = [
        predict_thresh(pivot, corr, userId, movieId, thresh_common_nb)
        for _, userId, movieId, _ in tqdm(
            test_ratings[["userId", "movieId", "rating"]].itertuples()
        )
    ]
    # test_ratings

    # Step 4 evaluate the resulte : with classical metrics

    print(
        f"RMSE for N={thresh_common_nb}:",
        np.sqrt(
            mean_squared_error(
                y_true=test_ratings["rating"],
                y_pred=test_ratings[f"User based_thresh_{thresh_common_nb}'
            )
        ),
    )
```

```
10000it [00:18, 547.80it/s]
RMSE for N=10: 1.727123456965418
10000it [00:17, 582.31it/s]
RMSE for N=15: 1.710466778197879
10000it [00:17, 584.81it/s]
RMSE for N=20: 1.701306646530708
10000it [00:16, 593.53it/s]
RMSE for N=30: 1.6905802601250242
10000it [00:17, 575.09it/s]
RMSE for N=50: 1.6794334332143162
```

In [ ]:

```
hr = HRatio(
    test_ratings=test_ratings,
    predictor=predictor,
    predict_func=predict_thresh,
    K=25,
)
HR["User based_thresh"] = hr
```

```
100%|██████████| 1000/1000 [01:37<00:00, 10.23it/s]
The Hit Ratio @ 25 is 0.78
```

In [ ]:

```
test_ratings
```

Processing math: 100%

Out[ ]:

	userId	movieId	rating	User based	User based_uc	User based_thresh_10	User based_thresh_15	based_
557	90	900	4	0.657995	3.758684	1.120209	1.204467	
6675	90	269	5	0.987827	3.845841	1.570607	1.585766	
562	90	289	3	0.572064	3.687360	1.100158	1.112312	
62660	90	270	4	1.077609	3.422702	1.241466	1.279309	
68756	90	268	4	1.202310	3.836968	1.587394	1.607269	
...	...	...	...	...	...	...	...	
46773	729	689	4	2.500000	2.500000	2.500000	2.500000	
73008	729	313	3	2.500000	2.500000	2.500000	2.500000	
46574	729	328	3	2.500000	2.500000	2.500000	2.500000	
64312	729	748	4	2.500000	2.500000	2.500000	2.500000	
79208	729	272	4	2.500000	2.500000	2.500000	2.500000	

10000 rows × 10 columns

In [ ]:

```
HR
```

Out[ ]:

```
{'User based': 0.78, 'User based_uc': 0.803, 'User based_thresh': 0.782}
```

To go further

1. Do the same, but with correlation between items. It's Collaborative filtering based on Items similarity. It's also possible to use the 2 previous trick
2. Use Matrix factorization: decompose R in P, Q at rank k (i.e. if R is a m.n matrix, P is a m.k matrix and Q is a n.k matrix) the reconstruct R with P and Q (i.e. $\hat{R} = PQ^T$)
3. Use Matrix decomposition: do an truncated SVD decomposition in order to obtain U, S and V, build $\hat{R} = USV^T$

TODO – Students

- Choose, implement and evaluate one of the above strategies.

Collaborative filtering based on items similarity

In [ ]:

```
""" Pivot table for train set """
train_pivot = pd.pivot_table(
    data=train_ratings,
    values="rating",
    index="movieId", # used to be userId
    columns="userId", # used to be movieId
) # the changes cause the axes to be reversed
print("Nb movies:", train_pivot.shape[0])
print("Nb users: ", train_pivot.shape[1])
train_pivot
```

Processing math: 100%

```
Nb movies: 1637
Nb users:  867
```

Out[ ]:

userId	1	2	3	5	6	7	8	9	10	12	...	934	935	936	937
movieId															
1	5.0	4.0	NaN	4.0	4.0	NaN	NaN	NaN	4.0	NaN	...	2.0	3.0	4.0	NaN
2	3.0	NaN	NaN	3.0	NaN	NaN	NaN	NaN	NaN	NaN	...	4.0	NaN	NaN	NaN
3	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	4.0	NaN
4	3.0	NaN	NaN	NaN	NaN	5.0	NaN	NaN	4.0	5.0	...	5.0	NaN	NaN	NaN
5	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1678	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN
1679	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN
1680	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN
1681	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN
1682	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN

1637 rows × 867 columns

In [ ]:

```python
# Step 1: build the similarity matrix between users

# no need to remove the transpose since we exchanged movieId and userId
# when making the pivot table
correlation_matrix = train_pivot.transpose().corr("pearson")
correlation_matrix
```

Out[ ]:

movieId	1	2	3	4	5	6	7	8	
movieId									
1	1.000000	0.198057	0.172936	0.128676	0.378934	0.529401	0.153225	0.272667	(
2	0.198057	1.000000	0.172189	0.187792	0.335075	-0.158114	0.140478	0.306391	-(
3	0.172936	0.172189	1.000000	-0.134625	0.177084	0.806226	0.017779	-0.182750	-(
4	0.128676	0.187792	-0.134625	1.000000	-0.190204	0.066625	0.186239	0.252612	(
5	0.378934	0.335075	0.177084	-0.190204	1.000000	1.000000	0.127930	0.233920	-(
...	...	...	...	...	...	...	...	...	
1678	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
1679	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
1680	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
1681	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
1682	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

1637 rows × 1637 columns

In [ ]:

Processing math: 100%

```python
# Step2 build rating function
# We want to calculate the rating that a user could have given for an item.
```

```
# Il est plus efficace de travailler avec numpy qu'avec pandas.
# On transforme donc la matrice pivot en numpy
pivot = train_pivot.to_numpy()
# idem pour la matrice de correlation
corr = correlation_matrix.to_numpy()
# Malheureusement, on doit utiliser 2 dictionnaires pour passer
# Du nom de la colonne movieId dans son indice en numpy
movie2column = {j: i for i, j in enumerate(train_pivot.columns)}
# Du nom de la ligne userId dans son indice en numpy
user2row = {j: i for i, j in enumerate(train_pivot.index)}


# the names of movieId and userId should be reversed
def predict(pivot, corr, userId, movieId):
    if movieId in movie2column.keys():
        movie = movie2column[movieId]
    else:
        return 2.5
    if userId in user2row.keys():
        user = user2row[userId]
    else:
        return 2.5

    # Normalement le rating est inconnu
    if np.isnan(pivot[user, movie]):
        num = 0
        den = 0
        for u in range(len(corr)):
            if not np.isnan(pivot[u, movie]) and not np.isnan(corr[user, u]
                # Si l'utilisateur u a déjà vu le film movie
                # et si les deux utilisateurs ont au moins vu un même film
                den += abs(corr[user, u])
                num += corr[user, u] * pivot[u, movie]
        if den != 0:
            return num / den
        else:
            return 2.5  # default value
    else:
        # le film a déjà été vu
        print(
            f"le film {userId} a déjà été vu par l'utilisateur {movieId}",
            f"et a reçu la note de {pivot[user, movie]}",
        )
        return pivot[user, movie]


predict(pivot=pivot, corr=corr, userId=1, movieId=1)
predict(pivot=pivot, corr=corr, userId=3, movieId=28)
```

```
le film 1 a déjà été vu par l'utilisateur 1 et a reçu la note de 5.0
```

Out[ ]:

```
2.577331638036494
```

In [ ]:

```
# Step 3 add the predicted rating to the test set

test_ratings["items_based"] = [
    predict(pivot, corr, userId, movieId)
    for _, userId, movieId, _ in tqdm(
        test_ratings[["movieId", "userId", "rating"]].itertuples() # invert
    )
]
_ratings
```

Processing math: 100%

```
10000it [00:11, 833.96it/s]
```

Out[ ]:

	userId	movieId	rating	User based	User based_uc	User based_thresh_10	User based_thresh_15	based_
557	90	900	4	0.657995	3.758684	1.120209	1.204467	
6675	90	269	5	0.987827	3.845841	1.570607	1.585766	
562	90	289	3	0.572064	3.687360	1.100158	1.112312	
62660	90	270	4	1.077609	3.422702	1.241466	1.279309	
68756	90	268	4	1.202310	3.836968	1.587394	1.607269	
...	...	...	...	...	...	...	...	
46773	729	689	4	2.500000	2.500000	2.500000	2.500000	
73008	729	313	3	2.500000	2.500000	2.500000	2.500000	
46574	729	328	3	2.500000	2.500000	2.500000	2.500000	
64312	729	748	4	2.500000	2.500000	2.500000	2.500000	
79208	729	272	4	2.500000	2.500000	2.500000	2.500000	

10000 rows × 11 columns

In [ ]:

```
# compute RMSE
print(
    "RMSE:",
    np.sqrt(mean_squared_error(test_ratings["rating"], test_ratings["items_
)
```

```
RMSE: 1.7554528828991547
```

Matrix factorization

Same approach as in the slides.

Below is a simple algorithm for factoring a matrix.

In [ ]:

```
# Matrix factorization from scratch
def matrix_factorization(R, K, steps=10, alpha=0.005):
    """
    R: rating matrix
    K: latent features
    steps: iterations
    alpha: learning rate
    beta: regularization parameter"""

    # N: num of User
    N = R.shape[0]
    # M: num of Movie
    M = R.shape[1]

    # P: |U| * K (User features matrix)
    P = np.random.rand(N, K)
    # Q: |D| * K (Item features matrix)
    Q = np.random.rand(M, K).T

    for step in tqdm(range(steps)):
        for i in range(N):
```

Processing math: 100%

```
            for j in range(M):
                if not np.isnan(R[i][j]):
                    # calculate error
                    eij = R[i][j] - np.dot(P[i, :], Q[:, j])

                    for k in range(K):
                        # calculate gradient with a and beta parameter
                        tmp = P[i][k] + alpha * (2 * eij * Q[k][j])
                        Q[k][j] = Q[k][j] + alpha * (2 * eij * P[i][k])
                        tmp = P[i][k]

    return P, Q.T
```

In [ ]:

```
# We try first on a toy example
# R: rating matrix
import math

R = [
    [5, 3, math.nan, 1],
    [4, math.nan, math.nan, 1],
    [1, 1, math.nan, 5],
    [1, math.nan, math.nan, 4],
    [0, 1, 5, 4],
    [2, 1, 3, math.nan],
]

R = np.array(R)
# Num of Features
K = 3

nP, nQ = matrix_factorization(R, K, steps=10)

nR = np.dot(nP, nQ.T)
nR
```

Out[ ]:

```
100%|██████████| 10/10 [00:00<00:00, 1784.28it/s]
array([[2.12758474, 1.84433709, 1.3047341 , 1.79528997],
       [1.62941137, 1.48932725, 1.00469   , 1.17875413],
       [1.41674085, 1.23374316, 0.87721332, 1.39813117],
       [1.06211232, 1.13392569, 0.70126885, 1.29508273],
       [1.27068877, 1.20998626, 0.79444241, 0.99844176],
       [1.83008845, 1.52090772, 1.09230563, 1.02466009]])
```

In [ ]:

```
""" TRY to predict with matrix factorization """
```

Out[ ]:

```
' TRY to predict with matrix factorization '
```

In [ ]:

```
""" Evaluate the result """
```

Out[ ]:

```
' Evaluate the result '
```

Decomposition using latent factor.

We use SVD decomposition

In [ ]:

```
pivot = train_pivot.fillna(0).to_numpy()
n_components = min(train_pivot.shape) - 1
```

Processing math: 100%

In [ ]:
```
from scipy.sparse.linalg import svds

k = 50
assert k < max_components

u, s, v_T = svds(pivot, k=k)
nR = u.dot(np.diag(s).dot(v_T))  # output of TruncatedSVD
```

In [ ]:
```
s
```

Out[ ]:
```
array([ 57.25005481,  57.57480375,  57.75314656,  57.99880033,
        58.13254489,  58.42083719,  58.8588979 ,  59.07946095,
        59.47557093,  59.65841912,  59.81901285,  60.58211626,
        61.21112094,  61.2998157 ,  62.04973244,  62.38679731,
        62.57380979,  62.99998426,  63.59984122,  64.39513078,
        64.71710402,  65.03539222,  65.38927487,  65.61214988,
        66.80944062,  67.53227707,  69.31276991,  69.78436118,
        70.0279118 ,  71.31421275,  72.47720353,  73.71497721,
        74.67513991,  76.60176717,  78.43382523,  79.64080918,
        82.23187885,  85.53988698,  88.66338897,  90.01310144,
        96.65966513, 101.72448643, 116.74772225, 120.12556462,
       138.24673889, 145.94464114, 152.41581902, 203.78726783,
       230.63108475, 603.76784628])
```

In [ ]:
```
""" TRY to predict with SVD decomposition """
```

Out[ ]:
```
' TRY to predict with SVD decomposition '
```

In [ ]:
```
""" Evaluate the result """
```

Out[ ]:
```
' Evaluate the result '
```

Processing math: 100%