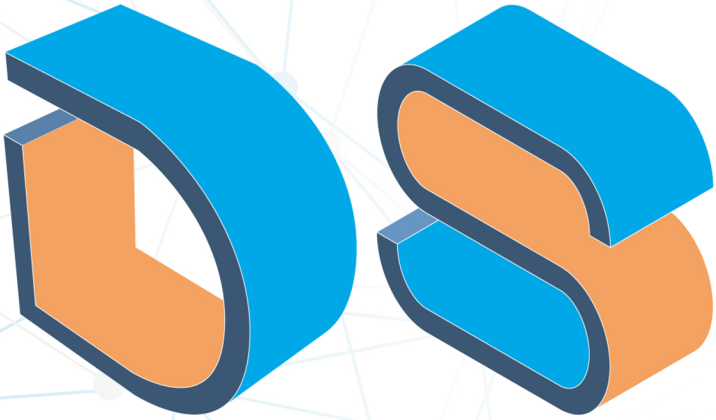


POLIMI
DATA SCIENTISTS

# Recommender Systems
## Course Notes

*Edited by:*
***Roberto Spatafora***
***Stiven Metaj***

# Credits

The following notes have been written by the Polimi Data Scientists student association by combining Prof. Cremonesi's lectures, content from `https://www.coursera.org/learn/basic-recommender-systems/home/welcome`, content from `https://www.coursera.org/learn/advanced-recommender-systems/home/welcome` and from the notes by Roberto Spatafora and Stiven Metaj.

They are meant as a support for the students following the course and they should not be considered as a replacement for the professor's lectures or the book suggested in the course bibliography.

It is strongly suggested to have a look on the Coursera content previously linked (powered by EIT Digital students) and also test the acquired capabilities there.

# Contents

# Chapter 1

# Introduction

A Recommender System (RS) is a system that provides recommendations to users. Its goal is to provide users with hints and suggestions about items that can meet their interest.
Some of the most common applications are streaming services and e-Commerce *(e.g., Netflix, Amazon)*

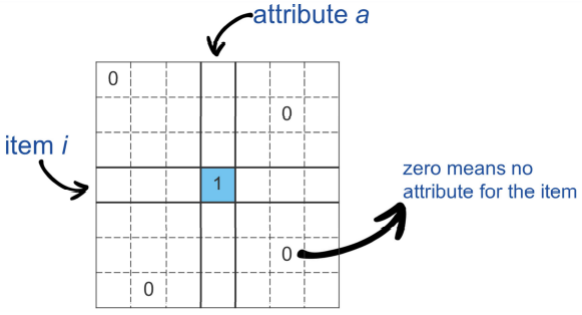
In order to provide recommendations to users a Recommender Systems needs data about *items, users and interaction between users and items.* For instance, if we recommend movies, genre, director and actors could be important data to describe that film; for what concerns users it could be important to take into account socio-demographics data about users, such as age and gender that can influence the recommendations *(e.g., It would be totally different to recommend movies to a 40 years-old man and movies to a 9 years-old girl)*; information about past user/item interactions can be used to understand their interests.

## 1.1 Taxonomy of Recommender Systems

Recommender algorithms can be first classified into two categories: *personalized* and *non-personalized* algorithms. The former approach is based on providing different recommendations to different users. The latter one provides the same recommendations to all the users. The course focuses the attention on personalized recommender algorithms; non-personalized algorithms are just used as baselines.
In order to build a personalized recommender algorithm there are two main techniques that can be used: Content-Based Filtering (CBF), Collaborative Filtering (CF), Context-Aware and Hybrid approaches.

## 1.2 Item Content Matrix

The Item Content Matrix (ICM) is a rectangular matrix which represents the items in the rows and their attributes in the columns. In its simplest form, the ICM contains binary values (0 or 1). If an item contains a specific attribute, the corresponding value in the matrix will be set to 1, 0 otherwise.



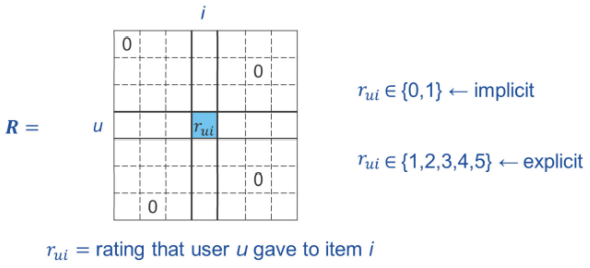

In a more realistic scenario, it can be useful to have positive real values in the matrix (instead of binary ones) in order to represent how much important an attribute is in characterizing an item.

## 1.3 User Rating Matrix

The User Rating Matrix (URM) is a rectangular matrix which contains the users in the rows and the items in the columns. Values in the URM are a mathematical representation of the ratings of past user/item interactions. Those values can be expressed in *implicit* or *explicit* representation.

In case of implicit ratings a value of 1 indicates that we can assume a positive interaction between a user and that item, on the other hand, a 0 value indicates that we have no information about the interaction of the user with the considered item. There are different ways to estimate the opinion of a user for an item, without asking for an opinion explicitly. Implicit ratings can be assumed from the user's behaviour
*(e.g., If a user buys an item on Amazon and do not send it back, we can assume the user likes that item)*
*(e.g., In the case of movies, we can assume that, if the user has stopped watching a movie after 20 minutes, the user did not like the movie)*

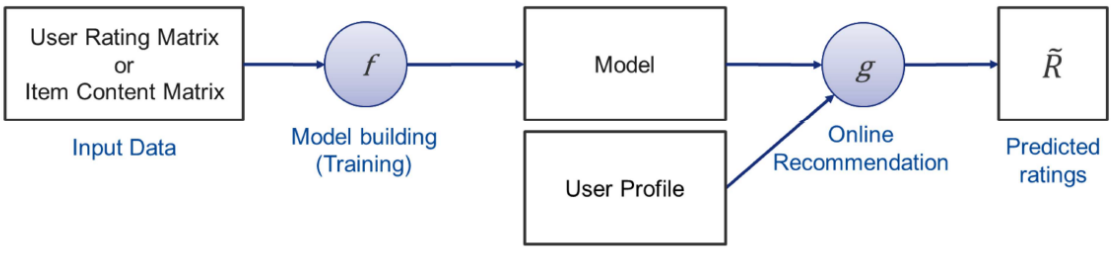
In case of explicit ratings, users use a range of numbers to rate items in the catalogue. A 0 value in the URM indicates the fact that we have no information about the user/item interaction. Therefore there are two possibilities: either the user has not interacted with that item or the user has not rated it. A way to understand user preferences is to ask him/her for an opinion. For this reason, it is important to decide how to organize the **rating scale**. A large rating scale would better reflect the opinion of the user. On the other hand, it requires more effort for the user to choose the correct rating. Thus, we have to expect fewer ratings.
Generally users tend to publish their rating only if they had a positive experience. This evidently creates a bias that affects the rating distribution.




Generally, a URM has a density $< 0.01\%$.
The goal of a recommender algorithm is to predict missing values in the URM.

## 1.4 High-level overview

A recommender algorithm is based on two mathematical functions, f *(model training)* and g *(recommendations)*.
Input to the $f$ function are data *(e.g., ICM or URM)* and the output is what we call the ***model***. The model is a representation of the user preferences.
The function $g$ takes as input the **model**, created from the previous function, and the **user profile**. The output is the estimation of some ratings.

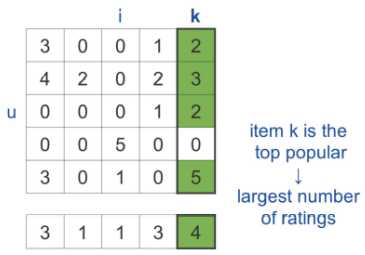


# Chapter 2

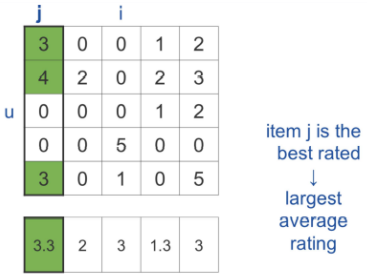
# Non-Personalized Recommenders

## 2.1 Non-Personalized Recommenders

Non-Personalized recommenders (NPR) provide the same recommendations to all the user.
An intuitive example of this category are Top-Popular recommenders. This type of recommenders is based on recommend the items that have been rated the greatest number of times. The popularity of an article only takes into account the number of users who have rated it, neglecting their actual opinion.
Another example of NPR is based on the Best Rated items. In order to compute the best rated items, we extract the average ratings per item from the URM and the recommended items will be the ones with highest average rating.




(a) Top-Popular recommender




(b) Best-Rated recommender

The average rating for item i is computed as:

$$b_i = \frac{\sum_u r_{ui}}{N_i}$$

where $r_{ui}$ are the ratings that the users have to that item (NON-zero ratings) and $N_i$ is the number of users who rated that item.
However the simple average rating formula does not take into account the support (subset of those elements not mapped to zero). Therefore, a simple yet effective option consists in modifying the formula by adding a constant term at the denominator called ***shrink term***.

$$b_i = \frac{\sum_u r_{ui}}{N_i + C}$$

In case of large support ($N_i \gg 0$) $C$ would be negligible.
In case of small support $C$ would have a consistent part in the formula.

**Shirnk Term**: it is important to understand that there is not a mathematical way to determine the best $C$ to use. So, the best way is to use a trial and error approach.

**Example**: a simple example that gives an idea of the effect of the shrink term using a $C = 1$

Average rating for item

$$b_i = \frac{5+4+3}{3} = 4 \qquad b_j = \frac{5}{1} = 5$$

Shrinked average rating for item (C=1)

$$b_i = \frac{5+4+3}{3+1} = 3 \qquad b_j = \frac{5}{1+1} = 2.5$$

| | | i | | |
|---|---|---|---|---|
| 5 | 0 | 0 | 1 | 2 |
| 4 | 2 | 0 | 2 | 3 |
| 0 | 0 | 0 | 1 | 2 |
| 0 | 0 | 5 | 0 | 0 |
| 3 | 0 | 0 | 0 | 5 |

j

As the picture points out, by simply introducing the constant term at the denominator, we obtain a much more reliable value.

## 2.2 Global Effects

Global Effect (GE) is a simple yet effective set of techniques that can be used when dealing with explicit ratings. The basic idea behind GE is that users have different ways of rate items, there are users that are more generous and others that are more strict in rating items. Moreover, there are some items that have larger ratings than other items. These differences from the average behavior are called the biases of items and users. GE uses these biases to make predictions.

There are 6 steps to compute the global effect:
**Step 1**: Average rating for all the items rated by all the users.

$$\mu = \frac{\sum_i \sum_u r_{ui}}{N}$$

Where $\mu$ indicates the average $r_{ui}$ indicates the explicit rating given by user $u$ to item $i$ and $N$ is the total number of NON-zero ratings.

**Step 2**: Normalize all the NON-zero ratings. This step simply consist in removing the $\mu$ computed at the previous step to all the NON-zero rating of the URM.

$$r'_{ui} = r_{ui} - \mu$$

The resulting URM will be composed of $> 0$, those elements above the average in the original URM, and $< 0$ values, that were below the average in the original URM.

**Step 3**: This step computes, for each item, the item bias.

$$b_i = \frac{\sum_u r'_{ui}}{N_i + C}$$

$N_i$ indicates the number of users who have rated item $i$.
Note that the use of the shrink term is not mandatory but it has the same benefit explained in the previous section.

**Step 4**: Recompute the rating for each NON-zero element by subtracting the bias of the considered item from normalized ratings (computed at Step 2)

$$r''_{ui} = r'_{ui} - b_i$$

**Step 5**: This step computes, for each user, the user bias. This bias is introduced because users have different way to rate items

$$b_u = \frac{\sum_i r''_{ui}}{N_u + C}$$

$N_u$ indicates the number of items rated by user $u$

**Step 6**: Global Effects final formula estimates how much user $u$ would rate item $i$ as the sum of the average rating for all the items rated by all the users (Step 1), the item bias (Step 3) and the user bias (Step 5)

$$\tilde{r_{ui}} = \mu + b_i + b_u$$

It is clear that the final formula depends on the user. Thus, the estimated ratings would be different for each user. However, even if estimated ratings are different for each user, the elements that will be recommended would be the same for each user. As a result of this, the list of recommendations would be the same for every user, even if the estimated rating for each item recommended in the list would be different among the users.
That explain the reason why despite GE produces different estimated ratings it is considered as a NON-personalized algorithm: the actual recommendations would be the same for every user.

A simple example to make it clearer: if we have two users $U_a$ and $U_b$ and we have just 3 items to recommend $I_a$, $I_b$ and $I_c$, by using GE as a recommender algorithm, we would recommend to both the users the same ordered list (for instance $I_c$, $I_a$ and $I_b$) even if the estimated ratings would have a difference equals to the difference between the two user's biases.
A numerical example for the possible ratings $r_{u_a i_c} = 4$, $r_{u_a i_a} = 3$ and $r_{u_a i_b} = 2$ while $r_{u_b i_c} = 3.5$, $r_{u_b i_a} = 2.5$ and $r_{u_b i_b} = 1.5$. As we can see the recommendations would be the same even if estimated ratings are different.

# Chapter 3

# Quality of a Recommender System

The quality evaluation part is the most difficult one in the whole process of building a recommender algorithm. The quality of a recommender algorithm depends on the **dataset**, **algorithm** and **user interface**.
There are two different categories of evaluation techniques: *Online* and *Offline*. Those two categories will be better explained in the following sections.

## 3.1 Quality indicators

In this section we will study some useful indicators that will allow us to analyse the quality of a recommender algorithm.

1. ***Relevance***: ability to recommend items that users like (user's point of view)

2. ***Coverage***: ability to recommend most of the items in a catalogue (provider's point of view)

3. ***Novelty***: ability to recommend something that is novel for the user (user's point of view)

4. ***Diversity***: ability to recommend items which are not too similar with each other (provider's point of view)
   *(e.g., If I know that a user loves pizza and I recommend him always to eat a pizza, I will never be wrong but it can annoy the user)*

5. ***Consistency***: ability to find the evolution of the user and at the same time not change too frequently recommendations in order to not disorient the user (provider's point of view)

6. ***Confidence***: ability to measure how much a system is sure about recommendations (provider's point of view)

7. ***Serendipity***: ability of surprising the user. It is the ability to recommend something that the users would never be able to find on their own

## 3.2 Online Evaluation Techniques

The first category of the evaluation techniques is composed by the online ones.
There are four different online evaluation techniques:

- ***Direct user feedback***: we can ask some real users to define their level of satisfaction about the recommender system *(e.g., For example through a survey)*
  This is a good choice but there are two problems:

  1. Only a very small number of users accept to do surveys/questionnaires
  2. The opinion expressed by the user cannot be reliable since can be affected by the request.

- ***A/B testing***: the core idea is to divide users in two groups A and B and provide a different RS to each group ($RS_A$ to group A and $RS_B$ to group B). Then the provider can compare the behaviour of users who were totally unaware of the existence of two different systems.
  This is the most powerful technique.
  For example, if we consider a social network, the provider would monitor the period of time in which customers were kept online.

- ***Controlled experiments***: the provider select a group of potential customers and experiments with them, asking them for an opinion at the end. In this case users are aware of the experiment. This method has some problems since opinions cannot be reliable due to the fact that users are not motivated as real users for a real online system

- ***Crowdsourcing***: consists in asking people to answer an online questionnaire, for a compensation, expressing their opinion about the system. This technique can have a large number of "volunteers" but has some problems since these users are less reliable due to the fact that opinions can be affected by the compensation.

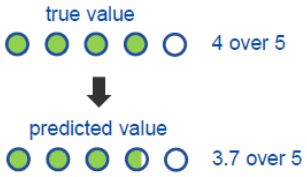
## 3.3 Offline Evaluation Techniques

There are four aspects that allow to offline estimate the quality of a recommender algorithm: **task**, **dataset**, **partitioning** and **metrics**.

### 3.3.1 Task

Firstly, we have to choose which task we want to evaluate. Two different methods that can be used:

1. **Rating prediction**: the goal is to predict a value that is as near as possible to the true value of the rating.




   As shown in the example, the 3.7 predicted value can be considered a good estimation since the real value is 4.

2. **Top-N recommendation**: the goal is to find N items which are relevant for the user. The typical approach is to first rank items from the most relevant to the less relevant.


   From the example reported in the above figure, 5 movies have been ranked according to the taste of a user, from the one that is estimated to be the most relevant for the user, to the less relevant.

### 3.3.2 Dataset

The dataset represents all the information we have available to make proper recommendations. In a recommender system scenario, our dataset is represented by a URM, whose values contains ratings users gave to items. As already mentioned in the URM section, usually, we know a very small percentage of all the possible ratings. The part that we know is called "ground truth" and it is made up of all the NON-zero ratings. The ground truth is divided in two main categories:

- ***Relevant***: contains all the positive opinions given by the users

- ***NON-relevant***: contains all the negative opinions given by the users.

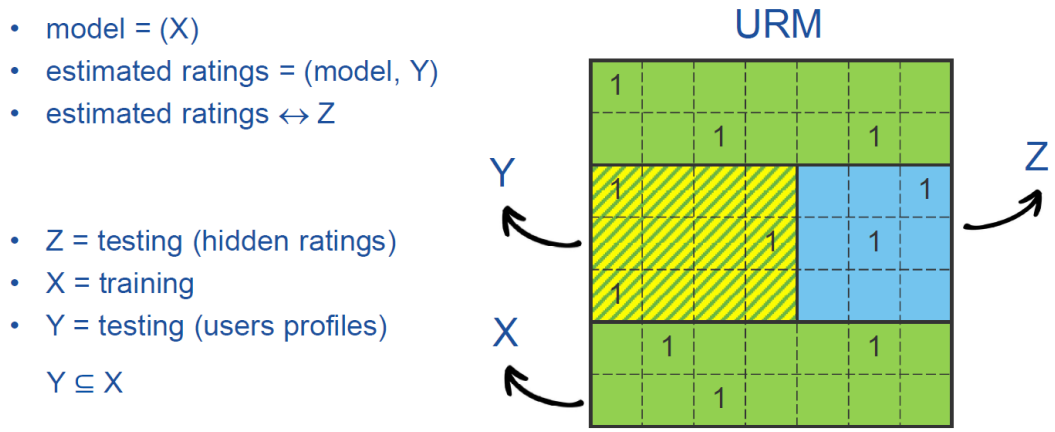
All the other data that are not contained in the ground truth are the ***unknown*** ratings.

### 3.3.3 Partitioning

#### Hold Out of Ratings

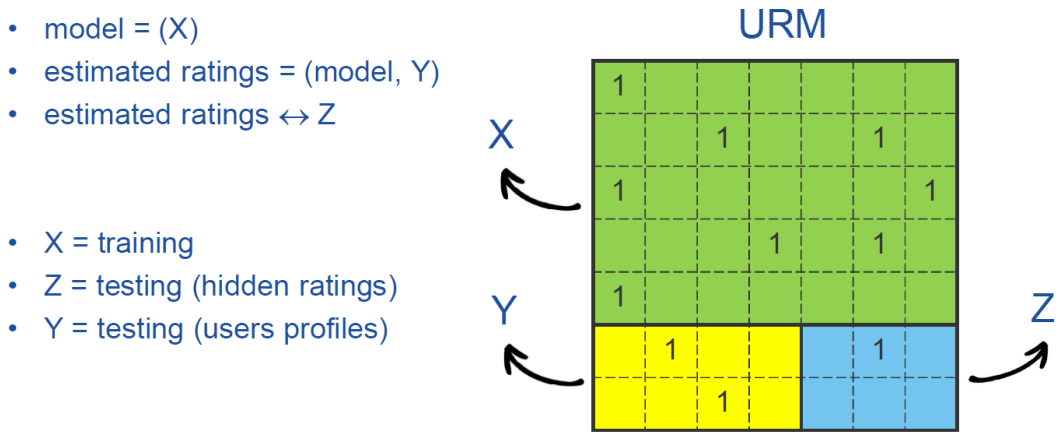
This technique is based on extracting some interactions (highlighted in blue in the image-example below) from the dataset and keep them away. The goal is to use them to evaluate the quality of the recommendations, once we have estimated the ratings.
This set (called Z in the example) is called *test set*. Then, the remaining ratings will be used as input for the function f and goes under the name of *training set* (X set in the example below).
Finally we select the user profiles for which we want to produce some recommendations (Y in the example below).

It is important to notice that, with hold-out of ratings, to produce recommendations, we use the same information to build the model and as user profiles to produce recommendation. Thus, a critical limitation of this approach is that we would not be able to provide recommendations for new users, not already present in the dataset.




#### Hold Out of Users

This is a technique that overcomes the problem of the Hold Out of Ratings. The key idea is to randomly remove some users from the dataset and use the remaining ones (highlighted in green in the image-example below) will be given as input to the f function, in order to be used for the model training.
Then, among the removed ones, we choose some interactions to build the test set (Z in the image below).
Finally, we keep the other interactions of those users whose ratings have been chosen to create the test set Z, in order to build the user profiles set Y.

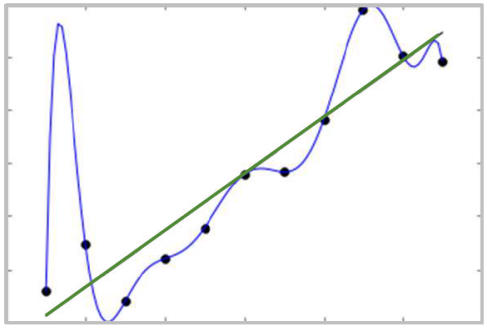


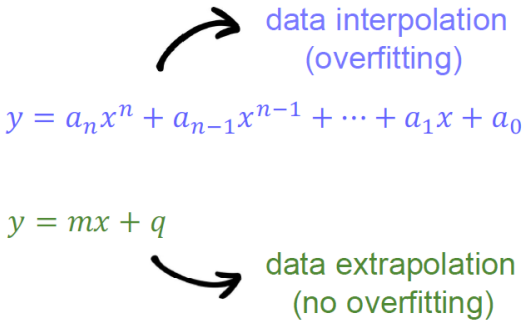

It is important to notice that, this type of approach has the set Y that is not a subset of the training set. Therefore, our model would work even for users that are not already in the dataset while building the model.

## 3.4 Overfitting

Overfitting is one of the main problems in the recommender system's field, and, more in general, in the data science world.

Suppose we have a certain number n of data, which are represented by the dots in this graph. They represent a linear dataset, with a little bit of noise. If we try to approximate through a n graded polynomial maybe we are able to perfectly connect all of them (data interpolation), but we have "listened" to much to the noise.
By doing this, we would lose the capability to generalize (data extrapolation).






## 3.5 Metrics

A recommender algorithm can be evaluated with three different kinds of quality metrics: ***error metrics***, ***classification metrics*** and ***ranking metrics***.

### 3.5.1 Error Metrics

The objective of Error Metrics is to estimate the difference between ratings assigned by algorithms and ratings assigned by users.
Error metrics are based on the difference between the true value and the estimated one. This is the error made by our algorithm.

**Mean Absolute Error**

Mean Absolute Error (MAE) computes, for each predicted rating, the difference between the actual rating and estimated rating. The overall average is the MAE.

$$MAE = \frac{\sum_{u,i \in T} |r_{ui} - \hat{r_{ui}}|}{N_T}$$

$\hat{r_{ui}}$: rating estimated by the algorithm
$r_{ui}$: true rating in the test set
$N_T$: number of interactions in the test set (NON-zero ratings)
$T$: test set

**Mean Squared Error**

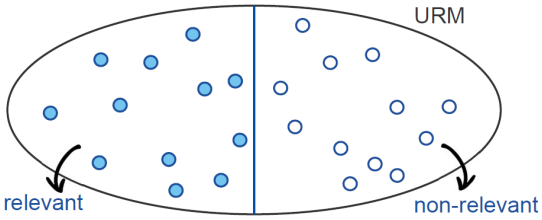
Mean Squared Error (MSE) consists in computing the average of the squared difference between each actual rating and the estimated rating.
To normalize, it is considered the root of the result.

This technique is much more popular since it is very difficult to minimize MAE while it is relatively easier to minimize the MSE.

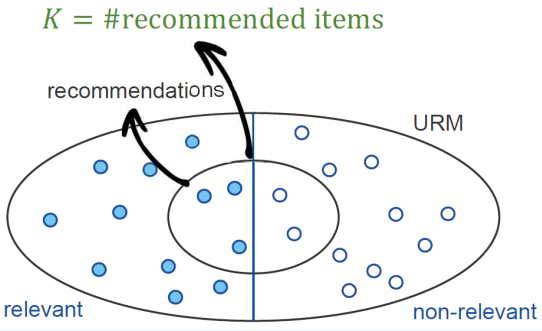
Our recommendations have to be overlapped with the ground truth, otherwise we cannot measure the error. This is equivalent to make an assumption: *Missing As Random*, **MAR**. That assumption means that we are assuming that the distribution of unknown ratings is identical to the distribution of the ground truth.
This assumption is far away from reality since the distribution of unknown ratings is almost opposed to the distribution of the ground truth. Indeed, we know that user are more likely to rate items that they liked with respect to those they did not like.

### 3.5.2 Classification metrics

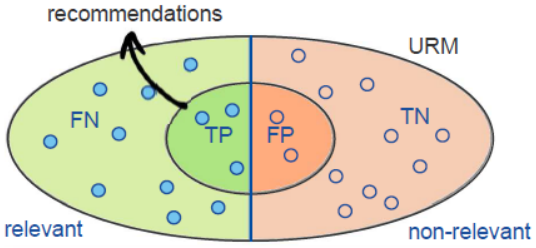
Firstly, we can divide our URM in two groups: *relevant* and *NON-relevant*.




Our algorithm may suggest a number $K$ of items, a group of which can be relevant and another part can be NON-relevant.




Therefore, our recommendations include both relevant and non-relevant items. There are some quantities to be defined in order to refer, to each possible group of items in the dataset, as accurately as possible.

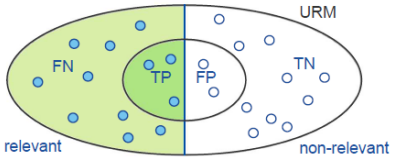



***True Positive (TP)***: items the algorithm recommends that are actually relevant for the user
***False Positive (FP)***: items the algorithm recommends that are NON-relevant for the user
***True Negative (TN)***: items the algorithm does not recommends that are effectively NON-relevant for the user
***False Negative (FN)***: items the algorithm does not recommends that are effectively NON-relevant for the user

### Recall

Recall is a measure of the percentage of positive instances correctly detected by the system among all the relevant items. It is defined as the number of relevant items recommended to the user, divided by the number of relevant items for the user.

$$\text{Recall}(K) = \frac{\#\text{relevant recommended items}}{\#\text{tested relevant items}} = \frac{\#\text{hits}}{\text{FN} + \text{TP}}$$

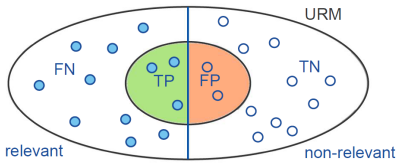



Recall is a measure that depends on the number of recommended items, K. Therefore, we will talk about recall at K. For instance, if out algorithm recommend all the items in the dataset, the recall will be 1. However, the quality depends also on the K items that are suggested.

### Precision

Precision is a measure of the ability of our recommender algorithm to suggest relevant items. It is defined as the number of relevant recommended items, divided by the number of all the recommended items.

$$\text{Precision}(K) = \frac{\#\text{relevant recommended items}}{\#\text{all recommended items}} = \frac{\#\text{hits}}{K}$$

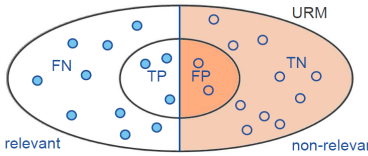



Precision depends on the number of recommended items (K) too, we will talk about precision at K.

### Fallout

Fallout is defined as the ration between all the NON-relevant recommended items, and all the NON-relevant recommended items. It it a measure of how bad our recommender algorithm is.

$$\text{Fallout}(K) = \frac{\#\text{non relevant recommended items}}{\#\text{all non relevant items}} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$




It is supposed to be as low as possible. Note that this measure can be computed only if the negative ratings are explicitly defined, since in an implicit dataset we should do assumption on zero values.

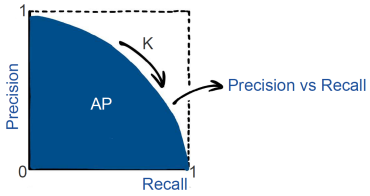
Considering our dataset, recommendations (using classification metrics) may be overlapped both with the ground truth and the unknown ratings. This would overcome the error metrics problem that works only for recommendations of items that are in the ground truth.
In this case we have to make an assumption on the unknown ratings since we do not know whether users did not like them and that is why they did not rate them or simply they did not interact with them. All these recommendations are considered as NON relevant. This assumption is called *Missing As Negative*, **MAN**. It is obviously a false assumption: it is not true that all the unknown are negative but it is more similar to the reality compared to the MAR.

### 3.5.3 Ranking Metrics

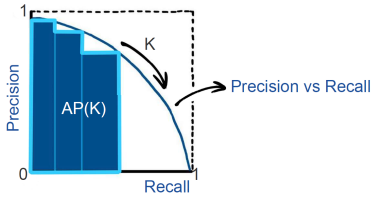
Ranking evaluation metrics try to measure how much a user likes an item, compared to the other items. In practice, rating metrics are used when an ordered list of recommendations is presented to the user. This metric tries to solve the problem of K of the Classification Metrics.
Generally speaking, when presenting to a user a list of items, it is advisable to put higher in the list the items that the user likes most. So, in principle, a ranking algorithm, given two items, x and y, should rank them, putting higher in the list the most liked one.

#### Average Precision

The Average Precision (AP) aims at finding the area under the Precision-Recall curve. In order to compute the AP ne need to divide the area under the blue curve in many different rectangles and approximate it. We build a rectangle for every value of K, that is a function of the number of recommended items K.




(a) Average Precision




(b) Average Precision at K

The delta recall at step k is computed as $\Delta R(k) = R(k) - R(k-1)$, knowing that $R(0) = 0$. As can be simply to deduce from the figure on the right above, $AP(K) = AP(k-1) + \Delta R(k) \cdot P(k)$. Summarizing the formulas we can describe the average precision as a function of K, as the sum for every k of the precision at k, multiplied by the delta recall. $AP(k) = \sum_k P(k) \cdot \Delta R(k)$

#### Mean Average Precision

The Mean Average Precision (MAP) is the sum of every average precision of every user, divided by the number of users, $N_u$. Mean average precision is the area under the precision-recall curve, from one, to the number of items that you recommend, calculated for every user (in the formula below it is indicated with the subscript u).

$$MAP(k) = \frac{\sum_u AP(k)_u}{N_u}$$

This is a powerful measure since it takes into account only the first K recommendations, not all the K values. It is important to highlight that this measure is computed for each user

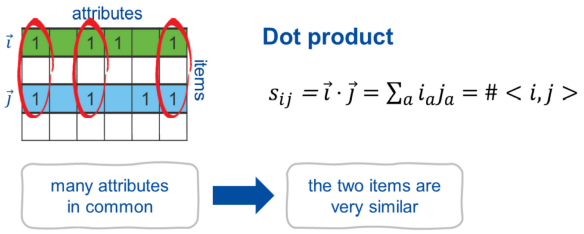
# Chapter 4

# Content Based Filtering

The basic idea behind Content Based Filtering (CBF) is to compare items based on their attributes. Based on their attribute, we want to understand how similar are two items. Depending on the shared attributes, we can do associations among items. The main assumption, at the foundation of content-based methods, is that a user that expressed a preference for an item, will probably like similar items.
*Example: We know that user X has seen the movie Star Wars. We also know that the movies "Star Wars" and "Indiana Jones" both have the actor, Harrison Ford, as a common attribute. We can assume that these two movies are similar with a certain degree, because they share a guest star. Given this, we can presume that the user X, having watched the movie "Star Wars", will probably like the movie "Indiana Jones" too. Following what we discovered, we can recommend the movie "Indiana Jones" to the user X.*
In order to represent the attributes of a pool of items, we use the ICM *(Ref. section 1.3)*

## 4.1 Cosine Similarity

In order to find a measure of similarity between two items i and j, based on the content of the ICM, we can look at the items i and j, as two vectors. Each vector has a number of elements, which is equal to the total number of attributes, available in the ICM. Vectors are binary vectors. In other words, values in the vectors can be either 0, or 1. The value 1 means that the item has such an attribute, while the value 0 means that the item does not have that attribute.
The simplest approach can be used may be to count how many attributes are in common between the two items. If two items have many attributes in common, we can assume that they are very similar.



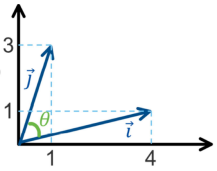

In a more formal way, the number of common elements between two binary vectors can be computed by using the dot product, where, the result of the similarity is the product of the two items' vector.

In same cases, the similarity can be improved by normalizing the dot product. We can take the dot product between the two vectors and normalize it with the length of the two vectors.

$$s_{ij} = \frac{\sum_a i_a \cdot j_a}{\sqrt{\sum_a i_a^2 \cdot \sum_a j_a^2}} = \frac{\vec{i} \cdot \vec{i}}{\left\|\vec{i}\right\|_2 \cdot \left\|\vec{j}\right\|_2} = \frac{\#<i,j>}{\sqrt{\#<i> \cdot \#<j>}}$$

The similarity computed this way is the cosine of the two vectors of attributes. In a graphical way, we can see that the angle of the cosine similarity is the angle included between the two vectors, that are the two items.

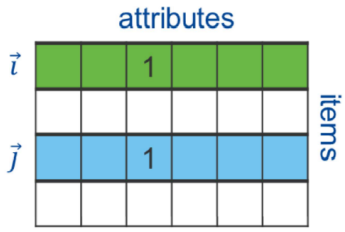
$$s_{ij} = \frac{\sum_a i_a \cdot j_a}{\sqrt{\sum_a {i_a}^2 \cdot \sum_a {j_a}^2}} = \frac{\vec{i} \cdot \vec{j}}{|\vec{i}|_2 \cdot |\vec{j}|_2} = cos\theta$$



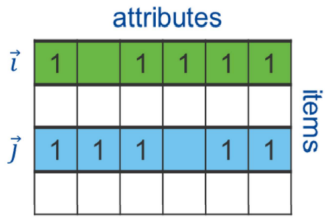

The more two items are similar, the more the cosine will be large.

### Support & Shrink Term

The support of a vector is the number of NON-zero elements in the item vector.




(a) Small support




(b) Large support

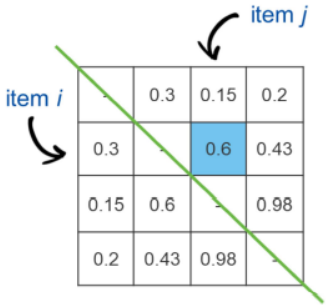
To have a clearer idea we can compute the similarities between i and j in both the examples reported above as explained in the previous paragraph. In the left example we would have $s_{ij} = \frac{1}{\sqrt{1 \cdot 1}} = 1$. Instead, in the second example $s_{ij} = \frac{4}{\sqrt{5 \cdot 5}} = 0.8$. As we can see from this simple example, we need to take into account something else since we cannot trust the formula of this simple approach.
In order to reduce similarity, so that we can take into account only most similar with large support, we introduce a new variable in the cosine similarity. This new variable is a constant term at the denominator, and it is called ***Shrink Term***.

$$s_{ij} = \frac{\vec{i} \cdot \vec{i}}{\left\|\vec{i}\right\|_2 \cdot \left\|\vec{j}\right\|_2 + C}$$

Coming back to the previous matrices, and choosing $C = 3$ as Shrink Term, we can notice that now the items into the second matrix are more similar than the one contained in the first one. This happens because, if the denominator of the cosine similarity is large, the shrink term will have a smaller effect, with respect to the situation in which the denominator is small. In the left example, now we have $s_{ij} = \frac{1}{\sqrt{1 \cdot 1} + 3} = 0.25$. Instead, in the second example $s_{ij} = \frac{4}{\sqrt{5 \cdot 5} + 3} = 0.5$. The computation is now, way more reliable than before.

### Similarity Matrix

The similarities computed across all pairs of items constitute the similarity matrix. An element $s_{ij}$ in the matrix describes how much item i is similar to item j. The similarity matrix, computed with the dot product or the shrinked cosine, is a symmetric matrix. In other words, the similarity between item i and item j is equal to the similarity between item j and item i: $s_{ij} = s_{ji}$




Be careful, this is just one of the many possibilities to compute item-item similarities. There are other possible definitions of similarities, which are not necessarily symmetric.

## 4.2 Estimated Ratings

Once that you have the similarity matrix, you can estimate $\tilde{r_{ui}}$, the rating of user u on item i, by relying on the past ratings from the same user on other items user u had already interacted with.

$$\tilde{r_{ui}} = \frac{\sum_j r_{ui} \cdot s_{ji}}{\sum_j s_{ji}}$$

The rating that user u will give to an item i is equal to the summation of the user's past rating on items j, multiplied by the similarity between the item j and i, and is all divided by the normalization.
Note that the normalization is useful if we want to estimate the ratings as accurately as possible. In case we want to rank items, by removing the denominator we can decrease the complexity of the formula and obtain better recommendations.

## 4.3 Matrix Notation

The matrix notation allows to write the equations of our recommender models in a more compact way. Considering each user u, it is possible to rewrite the formula $\tilde{r_{ui}} = \sum_j r_{ui} \cdot s_{ji}$ as:

$$\vec{\tilde{r_u}} = \vec{r_u} \cdot S$$

where $\vec{r_u}$ indicates the user profile and $S$ is the entire item-item similarity matrix.
It is possible to further generalize the formula considering all the users:

$$\tilde{R} = R \cdot S$$

where $R$ is the entire URM.

## 4.4 K-Nearest Neighbours

K-Nearest Neighbours (KNN) is a technique that is used to simplify and get the most out of the use of a similarity matrix. The biggest problem of the similarity matrix is that it is very dense. This causes the matrix to be heavy in a computational sense. Having a big data structure is expensive, in both terms of memory and time. Moreover, another problem of the similarity matrix is that the majority of its values are close to zero and therefore are useless for the recommendations.

A solution to these two problems is the K-Nearest Neighbours technique. This method consists in keeping only the K most similar items for each item. As K, we intend an integer number.
**Note:** a possible solution could be to keep only similarities above a certain threshold. This approach would be totally **wrong** since each row (item) can have different similarity values with other items.

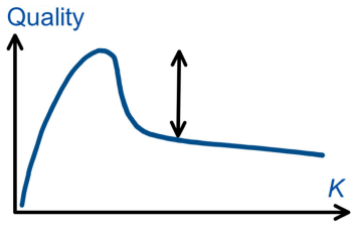
By setting all the values in the similarity matrix's rows to zero, except for the ones in KNN of the considered item, the matrix would not be symmetric anymore.

The KNN approach, modifies the estimated rating formula in order to not take into account the noise provided by all the items that are not the most similar for the considered item.

$$\tilde{r_{ui}} = \frac{\sum_{j \in KNN(i)} r_{uj} \cdot s_{ji}}{\sum_{j \in KNN(i)} s_{ji}}$$

| items \ items | | | | |
|---|---|---|---|---|
| | - | 0.3 | 0 | 0.2 |
| | 0 | - | 0.6 | 0.43 |
| | 0 | 0.6 | - | 0.98 |
| | 0 | 0.43 | 0.98 | - |

The quality of the recommender algorithm depends on the value of K. If K is too small, the model does not have enough data to make reliable estimations. On the other hand, if the value of K is taken too big, the data will contain too much noise.




## 4.5 Improving the Item Content Matrix

It is possible to further improve the quality of the ICM. One main improvement can be done by introducing *Non-Binary attributes*. This introduction allows to specify different weights to each type of attribute.
However, giving weights which are valid is difficult because it is a subjective opinion.

### 4.5.1 TF-IDF

The main principle of TF-IDF technique is to balance the weights of the attributes of the ICM depending on the frequency of appearance in the items.
This technique is used to automatically adjust the weights of attributes in the ICM.

The TF-IDF is given by the product of two terms, the Term Frequency, and the Inverse Document Frequency.

#### Term Frequency

Term Frequency (TF) is the ratio between the number of appearances of attribute a in item i and the number of attributes of item i.

$$TF_{a,i} = \frac{N_{a,i}}{N_i}$$

#### Inverse Document Frequency

Inverse Document Frequency (IDF) aims at solving the problem of little values of TF, for rare attributes. The IDF is the logarithm of the ration between the total number of items, and the number of items in which the attribute a is present.

$$IDF_a = log \frac{N_{ITEMS}}{N_a}$$

IDF measures the quantity of information we can derive from an attribute. This brings to identify an importance hierarchy.

*The objective of IDF is to penalize the attributes that are presents in so many items since it means that they are not relevant (almost obvious to have them) and favor the attributes that are more rare.*

# Chapter 5

# Collaborative Filtering

The basic idea of Collaborative Filtering (CF) methods is that the ratings a user has not given explicitly to the items can be inferred, because the observed ratings are often highly correlated across various users and items. CF approach is based on the user/item interactions. CF techniques do not require any item attribute to work. They rely on the opinions of a community of users. The main input to a CF algorithm is the URM. There are two different categories of collaborative filtering techniques: ***User-based*** and ***Item-based*** techniques.

## 5.1 User Based

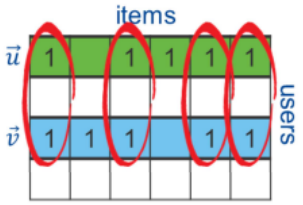
Let's assume we want to make recommendations to a user. The basic idea behind user-based collaborative filtering is to search for other users with a similar taste, and to recommend the items these users like most. From a more practical perspective we know that users can be similar and therefore we can recommend something that was interesting for a certain user u to a similar user v.
The idea is to look at the ratings of the users. We can compare users' taste based on their ratings.
The general idea is that if two users gave similar ratings to several items, we can assume that these two users have the same opinion on these items, and if they have the same opinions on a lot of items, we can assume that these two users are very similar.

### 5.1.1 Implicit Ratings

In order to find a measure of similarity between two users u and v, based on the content of the URM, we can look at the users u and v, as two vectors. Each vector has a number of elements, which is equal to the total number of items, available in the URM. Vectors are binary vectors. In other words, values in the vectors can be either 0, or 1. The value 1 means that the user has interacted with that item, while the value 0 means that the user has never interacted with that item.
The simplest approach can be used may be to count how many interactions are in common between the two users. If two users have many attributes in common, we can assume that they are very similar.

$$s_{uv} = \# < u, v >= \sum_i r_{ui} \cdot r_{vi}$$

In a more formal way, the number of common elements between two binary vectors can be computed by using the dot product, where, the result of the similarity is the product of the two users' vectors.
The example below shows a practical computation of the user-user similarity.




$$s_{uv} = 4$$

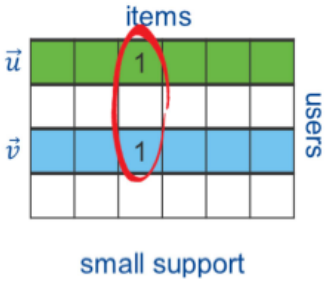
As we have already seen, it is a common practice to have these values normalized. One simple way of doing so is with the ***cosine similarity***.

$$s_{uv} = \frac{\sum_i r_{ui} \cdot r_{vi}}{\sqrt{\sum_i r_{ui}^2 \cdot \sum_i r_{vi}^2}}$$

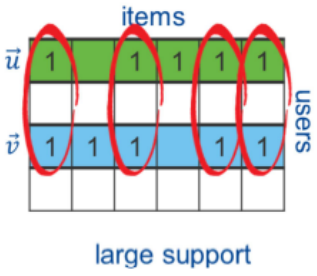
**Note**: this formula works *only* with implicit ratings.

### Support & Shrink Term

The support of a vector is the number of NON-zero elements in the item vector.




(a) Small support




(b) Large support

To have a clearer idea we can compute the similarities between users u and v in both the examples reported above as explained in the previous paragraph. In the left example we would have $s_{uv} = \frac{1}{\sqrt{1 \cdot 1}} = 1$. Instead, in the second example $s_{uv} = \frac{4}{\sqrt{5 \cdot 5}} = 0.8$. As we can see from this simple example, we need to take into account something else since we cannot trust the formula of this simple approach.
In order to reduce similarity, so that we can take into account only most similar with large support, we introduce a new variable in the cosine similarity. This new variable is a constant term at the denominator, and it is called ***Shrink Term***.

$$s_{uv} = \frac{\vec{i} \cdot \vec{i}}{\left\|\vec{i}\right\|_2 \cdot \left\|\vec{j}\right\|_2 + C}$$

Coming back to the previous matrices, and choosing $C = 3$ as Shrink Term, we can notice that now the users into the second matrix are more similar than the one contained in the first one. This happens because, if the denominator of the cosine similarity is large, the shrink term will have a smaller effect, with respect to the situation in which the denominator is small. In the left example, now we have $s_{uv} = \frac{1}{\sqrt{1 \cdot 1} + 3} = 0.25$. Instead, in the second example $s_{uv} = \frac{4}{\sqrt{5 \cdot 5} + 3} = 0.5$. The computation is now, way more reliable than before.

The similarities computed across all pairs of users constitute the similarity matrix. An element $s_{uv}$ in the matrix describes how much user u is similar to user v. The similarity matrix, computed with the dot product or the shrinked cosine, is a symmetric matrix. In other words, the similarity between user u and user v is equal to the similarity between user v and user u: $s_{uv} = s_{vu}$

| user u \ user v | | | | |
|---|---|---|---|---|
| | - | 0.3 | 0.15 | 0.2 |
| | 0.3 | - | 0.6 | 0.43 |
| | 0.15 | 0.6 | - | 0.98 |
| | 0.2 | 0.43 | 0.98 | - |

Be careful, this is just one of the many possibilities to compute user-user similarities. There are other possible definitions of similarities, which are not necessarily symmetric.

#### K-Nearest Neighbours

As we have already seen for the item-item similarity matrix, the biggest problem of the similarity matrix is that it is very dense. This causes the matrix to be heavy in a computational sense. Having a big data structure is expensive, in both terms of memory and time. Moreover, another problem of the similarity matrix is that the majority of its values are close to zero and therefore are useless for the recommendations.

Once again, a solution to these two problems is the K-Nearest Neighbours technique. This method consists in keeping only the K most similar users for each user. As K, we intend an integer number.
By setting all the values in the similarity matrix's rows to zero, except for the ones in KNN of the considered user, the matrix would not be symmetric anymore.

The KNN approach, modifies the estimated rating formula in order to not take into account the noise provided by all the items that are not the most similar for the considered item.

$$\tilde{r_{ui}} = \frac{\sum_{v \in KNN(u)} r_{vi} \cdot s_{vu}}{\sum_{v \in KNN(u)} s_{vu}}$$

The quality of the recommender algorithm depends on the value of K. If K is too small, the model does not have enough data to make reliable estimations. On the other hand, if the value of K is taken too big, the data will contain too much noise.

### 5.1.2 Explicit Ratings

In order to compute the similarity with explicit ratings, we have to take into account the fact that different users have different rating metrics. Thus, we need to consider the different ways users tend to express their opinion.

To do so, we have to take into account the user bias: the fact that some users, on average, give larger ratings than other users, so they express their opinions in different ways and the item bias: the fact that some items tend to attract larger ratings than others.

#### Pearson Correlation Coefficient Shrinked

In order to consider all the factors explained above, one trick is to normalize the ratings before the multiplication of the cosine formula and to take into account the support we can shrink the formula. Another trick is to remove the average of the ratings of user u and the average of the ratings of user v.

So, before doing the multiplication, we somehow normalize the ratings of each user by removing the user average. We do this normalization both at the numerator and the denominator.

$$s_{uv} = \frac{\sum_i (r_{ui} - \bar{r_u}) \cdot (r_{vi} - \bar{r_v})}{\sqrt{\sum_i (r_{ui} - \bar{r_u})^2 \cdot \sum_i (r_{vi} - \bar{r_v})^2 + C}}$$

where $\bar{r_u} = \frac{\sum_i r_{ui}}{N_u + C}$ and $\bar{r_v} = \frac{\sum_i r_{vi}}{N_v + C}$.

This formula allows to take into account the effect of the user bias. The fact the some users, on average, give larger ratings than other users, so they express their opinions in different ways.

**NOTE**: summation at the numerator must be computed *only* on **co-rated** items.

#### Estimated Ratings

To estimate how much a user would like a certain item i:

$$\tilde{r}_{ui} = \bar{r_u} + \frac{\sum_{v \in KNN(u)} (r_{vi} - \bar{r_v}) \cdot s_{vu}}{\sum_{v \in KNN(u)} s_{vu}}$$

It is important to consider that we are measuring the similarity between delta ratings. Delta ratings are how much the user will like an item more or less than his or her average.

## 5.2 Item Based

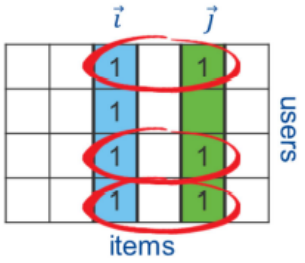
As we have seen so far, the basic idea behind Collaborative Filtering (CF) is to recommend items to users based on the items they previously interacted with.
In an Item-based approach, we want to measure the similarity between items.
It is extremely important to have a clear distinction in mind of the difference between item-item similarity in a CBF approach and a CF one. While in a CBF approach the similarity between items is computed based on common attributes, in a CF approach two items are similar if they have users that have interacted in common.

### 5.2.1 Implicit Ratings

In order to find a measure of similarity between two items i and j, based on the content of the URM, we can look at the items i and j, as two vectors. Each vector has a number of elements, which is equal to the total number of users, available in the URM. Vectors are binary vectors. In other words, values in the vectors can be either 0, or 1. The value 1 means that the user has interacted with that item, while the value 0 means that the user has never interacted with that item.
The simplest approach can be used may be to count how many users the items have in common, and we can assume that the similarity s i j, between item i, and item j, is equal to the number of common interactions between items i and j.

$$s_{ij} = \# < i, j >= \sum_u r_{ui} \cdot r_{uj}$$

In a more formal way, the number of common elements between two binary vectors can be computed by using the dot product, where, the result of the similarity is the product of the two items' vectors.
The example below shows a practical computation of the item-item similarity.




As we have already seen, it is a common practice to have these values normalized. One simple way of doing so is with the ***cosine similarity***.

$$s_{uv} = \frac{\sum_i r_{ui} \cdot r_{vi}}{\sqrt{\sum_i r_{ui}^2 \cdot \sum_i r_{vi}^2}}$$

**Note**: this formula works *only* with implicit ratings.

Moreover, in order to further improve the quality of our similarities we should shrink the form. At that point the reader should be familiar with the concept of Shrink Term, otherwise a reading of the previous chapters is suggested. As discussed in the previous sections, the matrix containing all the similarity values goes under the name of similarity matrix. In an Item-based CF we deal with an item-item similarity matrix.

#### Estimated Ratings

To estimate how much a user would like a certain item i:

$$\tilde{r}_{ui} = \frac{\sum_{j \in KNN(i)} r_{uj} \cdot s_{ji}}{\sum_{j \in KNN(i)} s_{ji}}$$

Note that this formula is completely equal to the CBF one.

### 5.2.2 Explicit Ratings

In order to compute the similarity with explicit ratings, we have to take into account the fact that different users have different rating metrics. Thus, we need to consider the different ways users tend to express their opinion.

To do so, we have to take into account the user bias: the fact that some users, on average, give larger ratings than other users, so they express their opinions in different ways and the item bias: the fact that some items tend to attract larger ratings than others.

#### Adjusted Cosine

In order to consider all the factors explained above, we use the Adjusted Cosine formula.

$$s_{ij} = \frac{\sum_u (r_{ui} - \bar{r_u}) \cdot (r_{uj} - \bar{r_u})}{\sqrt{\sum_u (r_{ui} - \bar{r_u})^2 \cdot \sum_u (r_{uj} - \bar{r_u})^2 + C}}$$

where $\bar{r_u} = \frac{\sum_i r_{ui}}{N_u + C}$.

This formula takes into account user biases, a shrink term and a normalization at the denominator.

# Chapter 6

# Memory Based and Model Based techniques

When building a RS, the first step is to create a model which describes users' behaviour. Once that you have the model, you have to compute the estimated ratings through the g function. *(Ref. section 1.4)*. The g function takes as input both the model and the profiles of the users you want to provide recommendations to.

If function g is such that is able to provide recommendations *only* to users whose profiles were already present when building the model then we deal with **memory based approach**.
On the other hand, if function g is such that is able to provide recommendations *also* to users whose profiles were not already present when building the model then we deal with **model based approach**.

## 6.1 Memory Based techniques

Memory Based techniques are those techniques in which ratings can be provided only to users whose profiles were already present when building the model. They use the ratings of the URM directly in predictions.

For new users, with this approach, we need to add the user to the URM, and recompute the model.

## 6.2 Model Based techniques

Model Based techniques are those techniques in which ratings can be provided also to users whose profiles were not already present when building the model. This technique allows to overcome the memory-based limitation since it can provide recommendations even to new users.

For new users, with this approach, we do not need to add the user to the URM.

#### Results of addition of user in Collaborative Filtering

In a User-based model the addition of a new user to the URM implies a completely different model (based on a user-user similarity matrix), which needs to be recomputed every time we add a user. This is clearly a **model based** approach.

In an Item-based model the addition of a new user to the URM does not affect too much the model (based on an item-item similarity matrix). This is an example of **memory based** approach.

# Chapter 7

# Association Rules

Association Rules (AR) from a RS point of view is a probabilistic approach to recommend an item to a user that has previously interacted with a set of items. We can estimate the probability that user u likes item i, conditioned by the fact that the user has previously interacted with other items.

$$P(i|j) \approx \frac{\#< i, j >}{\#< j >}$$

We may use this probability as a way to compute the similarity between two items, i and j.

We can, as usual, add the shrink term to further improve the quality.

$$\mathcal{P}(i|j) \approx \frac{\#<i,j>}{\#<j> + C} = s_{ij}$$

**Note**: the similarity matrix in this case would not be symmetric since $P(i|j) \neq P(j|i)$.

# Chapter 8

# Machine Learning approach - SLIM

## 8.1 Optimization problem

In this section, we will focus on Item-Based Collaborative Filtering with a machine learning approach. As we have seen in a previous chapter, Item Based Techniques try to estimate the rating a user would give to an item, based on the similarity with other items the user rated.

$$\tilde{r}_{ui}(S) = \sum_j r_{uj} \cdot s_{ji}$$

Obviously, it is import to compute the similarty matrix $S$ where an element $s_{ij}$ describes how much item i is similar to item j.
Therefore, we can say that the estimated ratings are a function of the similarity matrix.

One possible way to build an item-based collaborative filtering model could be to try all possible values for matrix S, such that, we are able to provide the best possible recommendations. In this case we will obviously commit an error, a gap between estimated and real ratings. This error goes under the name of ***Loss Function*** $E(S)$.
The loss function depends on the matrix S since the estimated ratings depend on S.

In a machine learning model the similarity values in S are the parameters of the model.
From a mathematical point of view, we want to minimize the error (the loss function is our gap between true and estimated values) and instead of exploring all the possible (infinite) values, we want to explore for the optimal $S$ in a "smart way".

This smart approach is based on choosing one of the metrics *(seen at section 3.5)* and adjusting the $S$ relying on the results given by the metrics.
*(e.g., Suppose we start by filling S only with 5 ratings, at the first step we would obtain an error due to overestimation. Therefore, at the second step, starting from the measured error, we can modify the values in S in order to reduce the previous error. A possible result from the previous step can be to reduce the 5 ratings of a certain factor. As a result, we will obviously reduce the error for overestimation and then we will repeat the process until we reach the best S possible.)*

## 8.2 Sparse Linear Method (SLIM)

SLIM is an Item-Based Collaborative Filtering approach based on Machine Learning. This approach in an extension of the Item-Based Collaborative Filtering that computes the similarity matrix with ML.
SLIM focuses on minimizing the Mean Square Error (MSE) - used as loss function - since it is differentiable and easier than other metrics to be used.

The loss function is the sum of the square of the difference between the actual ratings of a user and the estimated ones.

$$E(S) = \sum_{u,i \in R^+} (r_{ui} - \tilde{r}_{ui})^2$$

where, as we have already seen: $\tilde{r}_{ui} = \sum_j r_{uj} \cdot s_{ji}$.

In a matrix notation $E(S)$ would be equivalent to

$$E(S) = \|R - RS\|_2$$

Using a 2-Norm, this corresponds to compute the error over all the non-zero ratings in the URM.

Our objective therefore is to minimize the MSE, by choosing the best similarity matrix. This means finding the elements of the matrix S, such that we have the minimum over all possible matrices S of this 2-Norm, so the minimum of the 2-Norm of the matrix R, minus the matrix R dot S.

$$S^* = \min_S \|R - RS\| = \min_S E(S)$$

Clearly, the best solution would be the Identity matrix $S = I$, that is, all the elements in the diagonal are equal to 1 and the rest is 0. Then we get an error of zero, the lowest possible.
However this solution generates overfitting, as it is able to perfectly reduce the error to zero, but is useless in terms of its ability to recommend items that a user never interacted with in the past. In other words, we say that this solution is not able to generalize.

**How to avoid overfitting**

In order to avoid overfitting, we need to do two things:

1. We need to force the elements on the diagonal of S to be equal to zero
2. We need to add some regularization terms

Regularization terms help in avoid overfitting. Without them, you would be able to perfectly fit the model, but that model would lack in generalization.
By adding a regularization term we are actually modifying the loss function $E(S)$:

$$E(S) = \|R - RS\|_2 + \underbrace{\lambda \|S\|_2}_{\text{Regularization term}}$$

The goal of regularization term is to find a value of $S$ that tries to minimize the error between true and estimated ratings as much as possible *and* at the same time to **keep the matrix $S$ as sparse as possible**. The modifies error metric is called Ridge Regression, where only the 2-Norm is used.
It is important to properly tune the hyperparameter $\lambda$ with a validation set.
Let us consider the extreme cases for lambda

- $\lambda = 0$ implies $E(S) = \|R - RS\|_2$. The result is equal to not have regularization term
- $\lambda \to \infty$ implies $E(S) = \lambda \|S\|_2$ ans S tends to be as close as possible to zero. In this scenario we are keeping S as sparse as possible but the error would be maximum.

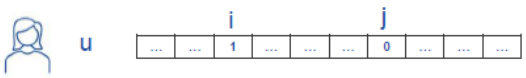
We can further modify the loss function, adding more than a single term for regularization. **Elastic Net** is another approach that consider both the 1-norm and the 2-norm adding two regularization terms:

$$E(S) = \|R - RS\|_2 + \underbrace{\lambda_A \|S\|_2 + \lambda_B \|S\|_1}_{\text{Regularization term}}$$

## 8.3 Bayesian Probabilistic Ranking

So far, we have focused our attention on optimizing the mean square error. Now, we will focus our attention on the **pairwise ranking**. For simplicity of discussion, we will focus on implicit ratings. Assume that we have a user u, and two items, i and j. Having 1, it means that user u have interacted with the item. So, we can assume that item i is relevant. Instead, having 0, it means that user u have not interacted with the item. So, we can assume that item j is not relevant.

From here on, we use index i to identify relevant items, that are the items the user has interacted with and index j to identify non-relevant items, that are the items the user did not interacted with.




Our goal is to estimate the rating of user u for items i and j, in such a way that the estimated rating for item i is greater than the estimated rating for item j. In this way, the estimated pairwise ranking will be identical to the true pairwise ranking. Suppose we are able to compute the probability of estimating rating u on i and j: our objective function is to maximize this probability, so that, this probability should be as close as possible to 1.

$$\max P(\tilde{r}_{ui} > \tilde{r}_{uj}|u)$$

We can define this probability as a function of the difference ($x_{u_{ij}} = \tilde{r}_{ui} - \tilde{r}_{uj}$) of the two estimated ratings: $\sigma(x_{u_{ij}}) = \frac{1}{1+e^{-x_{u_{ij}}}}$. We can approximate the probability with the sigma function. Function sigma is such that, when $x_{uj}$ is very large, the estimated probability is close to one. On the contrary, when the $x_{uj}$ is very small, the estimated probability is very low.

It is possible to demonstrate that the BPR algorithm optimizes the AUC (Area Under the Curve), which is the area of the Fallout-Recall graph and it must be as large as possible.

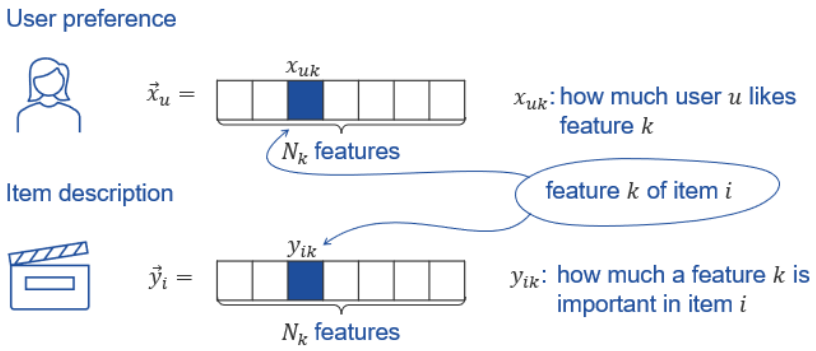
# Chapter 9

# Matrix Factorization

## 9.1 Matrix Factorization

Matrix Factorization (MF) is a machine learning approach based on Dimensionality Reduction.
This approach assumes that each user has preferences about different attributes and each item is characterized by the relevancy of each attribute used to describe it.

Bringing this idea to Matrix factorization, from now on, attributes will be called *"features"*, or *"latent factors"*, since we are in a different field. Be careful that latent factors are just a mathematical abstraction. These features has nothing to do with real item's attribute. We can say that the user preferences for each feature, are stored in vector $\vec{x}_u$. while the item features, are described by vector $\vec{y}_i$. The length of both these vectors, is $N_k$, that is the number of features used to described each item.



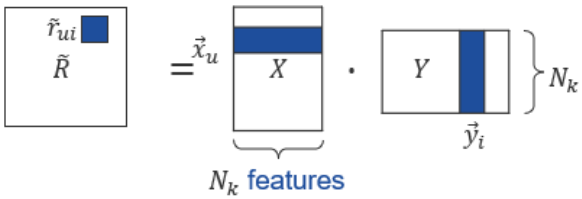

Each element $x_{uk}$ of vector $\vec{x}_u$, describes how much user u, likes feature k. Similarly, each element $y_{ik}$ of vector $\vec{y}_i$, describes how much feature k is important for item i.

With Matrix Factorization, we estimate the rating for user u on item i ($\tilde{r}_{ui}$), as the sum, for all the features, of how much user u likes feature k, times, how much feature k is important for item i

$$\tilde{r}_{ui} = \sum_k x_{uk} \cdot y_{ik} = \vec{x}_u \cdot \vec{y}_i$$

If we expand this computation for all the users and all the items, it is easy to see that matrix R, can be written as the product between matrix X (the users-features matrix), and matrix Y (the features-items matrix).

$$\tilde{R} = X \cdot Y$$




Differently from content-based approaches, with Matrix Factorization we usually do not assume that we already have information about the preferences of the users for a feature, or about the importance of a feature for an item. Instead, we rely on machine learning, in order to fill the matrices X and Y.
We call these ideal matrices $X^*$ and $Y^*$. And we estimate their values, by minimizing the same error between the true rating (given by the user) and the estimated rating.

$$X^*, Y^* = \min_{X,Y} \left\| R - \tilde{R} \right\|_2$$

The simplest approach, is to use the square 2-norm, minimizing the Mean Square Error, between true, and estimated ratings. This means that we use the missing as random assumption (MAR): the opinion for the items that user has not rated, is equivalent to the average of the opinions on the rated items.

As we can see, both matrices X and Y are rectangular matrices. Matrix X, has a number of rows equal to the number of users $N_u$, and number of columns equal to the number of features $N_k$. Similarly, matrix Y, has a number of columns equal to the number of items $N_i$, and number of rows equal to the number of features $N_k$.

$N_k$ is a hyperparameter that needs to be tuned, in order to have a good model. If we have a User Rating Matrix that is very sparse, and we choose a large number of latent factors, we risk to overfit the model to the few ratings available.
In this scenario, our recommender system will have an extremely poor performance in new cases.
Instead, if we have a dense User Rating Matrix, and we choose a small number of latent factors, we reduce the personalization capabilities of the system. In the extreme case, if we choose 1 as number of latent factors, our system will recommend the most popular items.

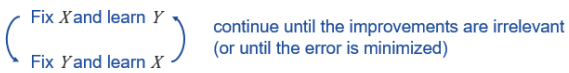
As it happens for other techniques, in Matrix Factorization too, we must be careful to prevent the risk of overfitting the model.
The solution, is to add regularization terms for the matrices X and Y in the error function. The goal of the regularization term, is to push matrices X and Y to be sparse. The parameters $\lambda_1$ and $\lambda_2$, control how much the regularization is important, compared to the rating error.

$$X^*, Y^* = \min_{X,Y} \left\| R - \tilde{R} \right\|_2 + \underbrace{\lambda_1 \|X\| + \lambda_2 \|Y\|}_{\text{Regularization term}}$$

## 9.2 Alternating Least Squares

Alternating Least Squares (ALS) is a technique to easily solve the Matrix Factorization problem.
Since we cannot solve analytically the optimization, to find $X^*$ and $Y^*$, we need a way to learn those matrices. From now on we won't consider the regularization term for simplicity.

ALS is the solution. It relies upon the idea of considering just one block at the time of the building blocks.
At the beginning, the matrices X and Y are filled with random values.

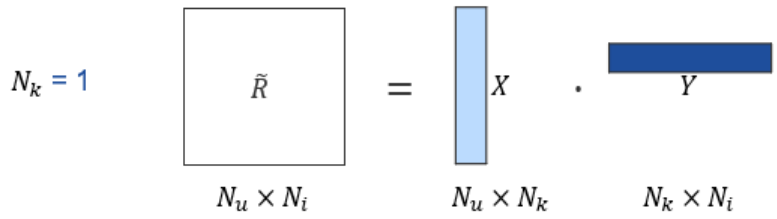



We then focus on fixing one of the two variables and solving the optimization problem by learning the other one; then change fixed and learning variables continuing this process until improvements are

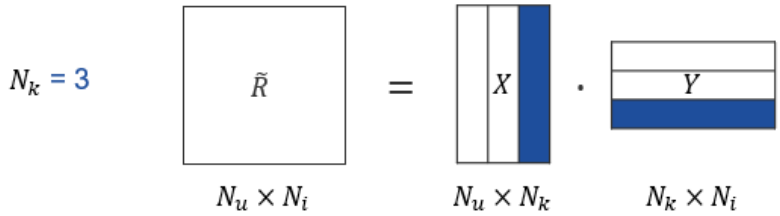
irrelevant.
The optimal solution from a mathematical point of view can be obtained after an infinite number of iterations, in reality we reach a result close to the optimal solution after 4 or 5 iterations.

## 9.3 Funk SVD

Funk SVD is a Matrix Factorization algorithm that relies on the ALS.
Here, the factors are not computed all at once. We start with the simplest model, so the number of latent factor $N_k$, is equal to 1.
Therefore, at this step the matrices X and Y are respectively a column vector and a row vector. We factorize the URM as the scalar product between these two vectors. With $N_k$ equals to 1, we apply ALS, to find the optimal values for X and Y. So first, we fix X, and learn Y. Then we fix Y, and learn X. And we iterate, till matrices X and Y do not change too much anymore.




When the error is minimized for $N_k$ equal to 1, we add an additional feature, now $N_k$ is equal to 2. The additional feature corresponds to the second column for X and the second row for Y. We already found the optimal values for the precedent factor, so we fix the first column of X and the first row of Y and ***apply ALS only to the column and row representing the additional feature.***




An example for k=3

The operation of increasing $N_k$, and tuning the matrices, is repeated until we reach the desired number of latent factors or the error is minimized to the desired value.
We learn the factorization in an incremental way. This trick is interesting for three reasons:

1. It is easy to implement
2. It consumes a small quantity of computational resources since at each iteration of $N_k$ you store only one vector for X and Y
3. It helps to avoid overfitting however it's not enough, so we need to add the regularization terms

## 9.4 SVD++

SVD++ is an extension of the Funk SVD model that is suitable only for explicit ratings. Thus, it adds to the Funk SVD the Global Effects that are added to the Funk SVD formula to normalize the estimated ratings.

$$\tilde{r}_{ui} = \mu + b_u + b_i + \sum_k x_{uk} \cdot y_{ki}$$

As seen in the second chapter, global effects are the sum of the mean rating $\mu$, plus, the user bias $b_u$, plus, the item bias $b_i$.

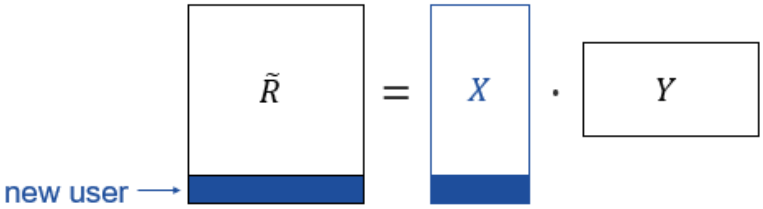
Because of the global effects, it is not possible anymore to use ALS, to solve the optimization problem.

$$\mu^*, b_u^*, b_i^*, X^*.Y^* = \min \left\| R - \tilde{R} \right\|_2 + \lambda_1 \|X\| + \lambda_2 \|Y\|$$

Therefore, SVD++, is typically solved by using an optimization technique called Stochastic Gradient Descent.

**Disadvantages of SVD++ (and Funk SVD)**

Even though SVD++, is an improved version of Funk SVD, both these techniques have the same problem: these approaches are memory-based, as the model need to be retrained every time a new user is added to the system, which means that we have to compute again the matrices X and Y, and eventually, the global effects.




## 9.5 Asymmetric SVD

Asymmetric SVD is a method to avoid recomputing the estimated ratings from scratch, every time a new user is added. Thus, it is a model based approach.
From now on, we will not consider the Global Effects in our formula for simplicity but keep always in mind that it works also with GE.

The problem Asymmetric SVD solves, is based on the fact that SVD++ is not model-based but it is a memory-based technique. Therefore, for every new user, we must compute the vector $x_u$, which stores the user preferences for the latent factors.
Assuming we know how much a latent feature k is present in an item j, we can approximate how much the user u will like feature k by looking at the opinion user u has of other items j.
Therefore, a practical solution to this problem, is to approximate the value of $x_{uk}$, as the sum of $r_{uj}$, times, $z_{jk}$, for every j.

$$x_{uk} = \sum_j r_{uj} \cdot z_{jk}$$

This way, we approximate how much user u likes feature k, as the sum, for all the items , of the user rating for that item j, times, the importance of feature k in describing that item.

We can also see this approximation as the product of two vectors. Vector $\vec{r}_u$, describes how much user u likes all the items, while vector $\vec{z}_k$, describes how much a feature k is present in all the items.


$$x_{uk} = \sum_j r_{uj} \cdot z_{jk} = \vec{r}_u \cdot \vec{z}_k$$

This technique is powerful, since we create a model by adding a vector, that approximates the user preference for a feature k. This way, when new users are added to the system, you can start giving them relevant recommendations, just by asking for a few of their favorite items.

From a matrix perspective, matrix X can be decomposed as the product of two matrices: R and Z.

$$X = R \cdot Z$$

Matrix R, stores user ratings with respect to the items, while matrix Z, stores the importance of features in those items


The definition of $\tilde{r}_{ui}$, can be re-written with the **decomposition** we have just described.

$$\tilde{r}_{ui} = \sum_k (\sum_j r_{uj} \cdot z_{jk}) \cdot y_{ki}$$

Now, R tilde, is defined as the product of 3 matrices: R, Z and Y. The product of Z and Y, produces an asymmetric similarity matrix, thus the name of the technique.

Both the matrix Z and Y are obtained through machine learning approach. Being the result of their product a similarity matrix, we can see a similarity between Asymmetric SVD and SLIM. However, Asymmetric SVD presents some advantages over SLIM when the number of latent factors $N_k$ is much smaller than the number of items $N_i$.

In asymmetric SVD, $\tilde{r}_{ui}$, is therefore defined as the sum, for all the latent features, of the sum for all the items of the recorded user preference for item j, times, the latent feature preference factor, times, the importance of latent feature k in item i.

**Note**: remember that it is possible to add the GE in explicit ratings cases.

## 9.6 Singular Value Decomposition

Singular Value Decomposition (SVD) is an approach from the linear algebra to decompose matrices, which has inspired all the techniques that we have seen so far. SVD is a method of decomposing a matrix as the product of three matrices $U$, $\Sigma$ and $V^T$.
This method states that each rectangular matrix of m times n dimension can be decomposed as the product of three matrices which have some particular properties.

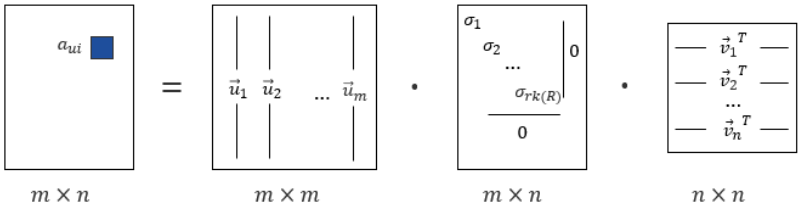
$$A = U \cdot \Sigma \cdot V^T$$

The first matrix of this decomposition is called $U$, and it is defined as left singular vectors. $U$ is the matrix of the eigenvectors of the square symmetric matrix $A \cdot A^T$ and therefore it has a dimension of $mxm$.
It is an orthogonal matrix, it means that $U^T \cdot U$ is equal to the identity matrix, and that $U \cdot U^T$ is equal to the identity matrix as well.
Its columns, are hierarchically arranged by their ability to describe the variance in columns of R, which means that the most important columns representing the matrix R, are positioned on the left.

The second matrix of this decomposition is called $V^T$, and it is defined as right singular vectors. $V$ is the matrix of the eigenvectors of the square, symmetric matrix $A^T \cdot A$ and therefore it has a dimension of $nxn$. $V^T$ is simply the transposition of V. It is an orthogonal matrix, it means that $V^T \cdot V$ is equal to the identity matrix, and that $V \cdot V^T$ is equal to the identity matrix as well. Its rows are hierarchically arranged by their ability to describe the variance in rows of R, which means that the most important rows representing the matrix R, are positioned at the top.

The third matrix of SVD formula is $\Sigma$. It is a diagonal matrix of the singular values *(remember that singular values are computed as the square root of the eigenvalues)* of the square symmetric matrix $A \cdot A^T$

or equivalently $A^T \cdot A$ since a matrix and its transpose have the same eigenvalues. Its dimension has to be $mxn$. The singular values are non-negative, and they are hierarchically sorted from the highest to the lowest.

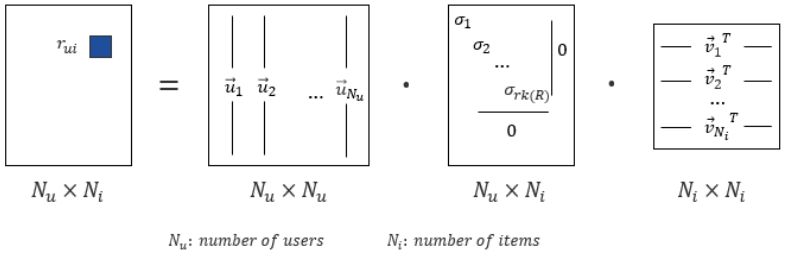



Since at each singular value in $\Sigma$ is associated one and only one eigenvector of $U$ and $V$, also columns of $U$ are hierarchically arranged alongside the correspondent singular value of $\Sigma$. On the left most side of $U$ we have the eigenvectors associated to the highest eigenvalues of $\Sigma$, and they are sorted following the order of the singular values in $\Sigma$.
A similar reasoning can be done on $V^T$. Its rows are hierarchically arranged alongside the correspondent singular value of $\Sigma$. On the top side of $V^T$ we have the eigenvectors associated to the highest eigenvalues of $\Sigma$, and they are sorted following the order of the singular values in $\Sigma$.

## 9.7 Pure SVD

In a recommender systems scenario, the SVD application would be based on the URM decomposition.

$$R = U \cdot \Sigma \cdot V^T$$

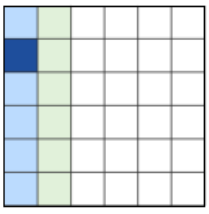



### 9.7.1 Interpretation of the matrices

**Matrix U**

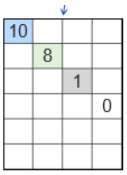
Matrix U expresses the preferences of users as regard features.
Rows correspond to users, while columns correspond to latent features.

It is important to note that, respect to the dimensionality explained before, we are now decomposing a URM of number of users times number of items dimension. Therefore, matrix U, would have a $N_u$x$N_u$ dimension.
So, the first column corresponds to feature 1, the second to feature 2, and so on. Each cell, defines how much a user is interested in a certain feature.



### Matrix $\Sigma$

Matrix $\Sigma$ expresses the importance of the features. The first singular value corresponds to the most important feature, that is feature 1, the second singular value corresponds to the second most important feature, and so on for all the features.

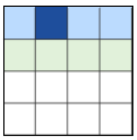



$N_u \times N_i$

### Matrix $V^T$

$V^T$ can be interpreted as a matrix that expresses how much is present a feature in items.
Columns correspond to items, while rows correspond to latent features.
Even in this case it is important to note that $V^T$ would have a $N_i$x$N_i$ dimension

It is important to note that, respect to the dimensionality explained before, we are now decomposing a URM of number of users times number of items dimension. Therefore, matrix $V^T$, would have a $N_i$x$N_i$ dimension.



$N_i \times N_i$

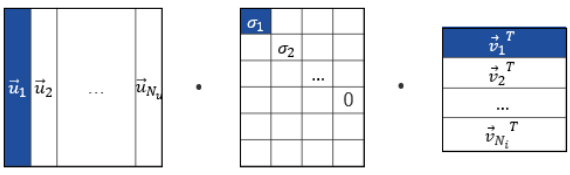
Now, let's make some additional considerations about the decomposition of the URM. As it is demonstrated from the linear algebra, if we apply the singular value decomposition to our URM, we will generate three matrices whose product will be exactly the original URM. This is clearly not what we are looking for, since it would imply the extreme case of overfitting: estimated URM equals to the original one.
This reasoning brings us to say that linear SVD needs some adjustments to fit in a recommender algorithm. In addition, it is very unlikely that the number of feature is either equals to the number of users or the number of items.

We can build our model using a step-by-step approach, in order to have an approximation of the original URM as a result. The approach starts from considering the highest singular value of sigma to the singular value for which the approximation of the original URM is good enough.

### SVD computation

When computing the SVD decomposition, first, we compute the part of the model for the most important feature. The first column of U is multiplied by the first singular value, that is multiplied by the first row of $V^T$.

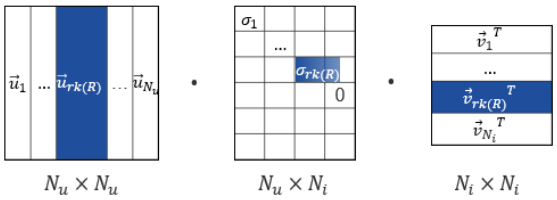
$$R = U \cdot \Sigma \cdot V^T = \sigma_1 \vec{u}_1 \vec{v}_1^T + ...$$




$N_u \times N_u$ $N_u \times N_i$ $N_i \times N_i$

Then, the algorithm works on the second most important feature summing the result to the previous one, and so on. For this reason, the decomposition of a matrix can be written as the sum of $\sigma_k$ times $u_k$ times $v_k^T$, for all the non-zero singular values.

$$R = U \cdot \Sigma \cdot V^T = \sum_{k=1}^{rk(R)} \sigma_k \cdot \vec{u}_k \cdot \vec{v}_k^T$$




We say this because there can only be a certain number of linearly independent columns fixed by the rank of R.
Therefore, all the other singular values are equal to zero, and do not add to the calculation.

$$R = U \cdot \Sigma \cdot V^T = \sum_{k=1}^{rk(R)} \sigma_k \cdot \vec{u}_k \cdot \vec{v}_k^T + \sum_{i=rk(R)+1}^{min(N_u,N_i)} 0 \cdot \vec{u}_i \cdot \vec{v}_i^T$$

Truncated SVD is the only way through which SVD can be exploited in a recommender algorithm. In order to not overfit our model, as we would do by considering all the singular values' contribute, the idea is to truncate the computation at early stages, when the approximation of the original URM is good enough.

If we consider separately the contributes from each feature, we have a sum of rank-1 matrices, ordered by decreasing importance: the first one, has the most important singular value, while the last has the least important one.
Each matrix is a rank 1 matrix, which means that there are only one column and one row that are linearly independent.

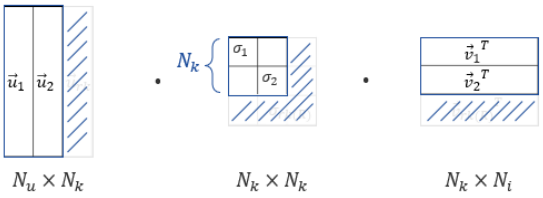
We call best rank-1 approximation, the product of $\sigma_1$, $u_1$, and $v_1^T$. It is the best approximation of rank-1 for R, among all the possible approximations of rank 1.
If we also consider the contribution given by the second rank-1 matrix, and we add it to the first one we will have the best approximation of rank-2 for R.
This can be repeated for all the other matrices: the sum corresponds to a progressive approximation of R.
It is important to remember that the order of the rank 1 matrices is established by sigma values, in decreasing order.

Truncation at $rank_k$, is defined as the approximation of matrix R, calculated as the sum of the first K rank-1 matrices.

We can note that, at each point of our computation we can find a scenario in which the addition of other singular values will not add a significant contribute in the computation of the estimated URM. *It would mean that we already kept most of the variance of the dataset.*

This way, R, can be approximated as the product of 3 matrices: $\tilde{U}$, $\tilde{\Sigma}$, and $\tilde{V^T}$, which are the truncated versions of the SVD decomposition matrices, corresponding to the largest $N_k$ singular values. $\tilde{U}$, is a portion of the first $N_k$ columns of U, while $\tilde{V^T}$, is composed of the first $N_k$ rows of V transposed.

$$R = U \cdot \Sigma \cdot V^T \approx \tilde{U} \cdot \tilde{\Sigma} \cdot \tilde{V^T}$$




Applying truncation to SVD, brings an additional consideration concerning matrices $\tilde{U}$ and $\tilde{V}$.

While $\tilde{U^T} \cdot \tilde{U}$ is equal to $\tilde{V^T} \cdot \tilde{V}$ that are the identity matrix $I$, it is not true if we switch the position of $\tilde{U}$ and $\tilde{U^T}$, or $\tilde{V}$ and $\tilde{V^T}$. The result of these products, is no longer an identity matrix.

With just few manipulation, considering the properties that characterize $\tilde{U}$ and $\tilde{V}$, we can make truncated SVD a model-based approach. Starting from our memory-based formula

$$R \approx \tilde{U} \cdot \tilde{\Sigma} \cdot \tilde{V^T}$$

by multiplying both the left side and the right part of the equation on the right by $\tilde{V}$

$$R \cdot \tilde{V} \approx \tilde{U} \cdot \tilde{\Sigma} \cdot \underbrace{\tilde{V^T} \cdot \tilde{V}}_{=\mathrm{I}}$$

we obtain on the second member of the equation $\tilde{V^T}$, times, v tilde, which is equal to the identity matrix. Now, by multiplying both the left side and the right part of the equation on the right by $\tilde{V^T}$

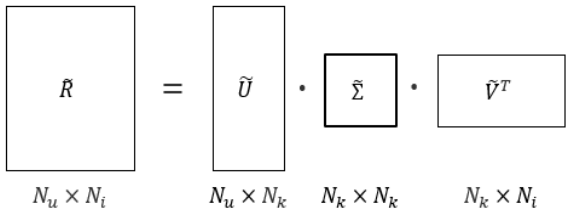
$$R \cdot \underbrace{\tilde{V} \cdot \tilde{V^T}}_{\neq I} \approx \underbrace{\tilde{U} \cdot \tilde{\Sigma} \cdot \tilde{V^T}}_{=\tilde{R}}$$

We can invert the two members

$$\tilde{R} \approx R \cdot \underbrace{\tilde{V} \cdot \tilde{V^T}}_{\approx S_{II}}$$

On the first member of the equation our estimated URM, $\tilde{R}$. At the second member there is the product between $\tilde{V}$, and $\tilde{V^T}$ which is no longer equals to the identity matrix and the resulting product would have a $N_i$ times $N_i$ dimension *(same structure of an item-item similarity matrix)*.

Truncated SVD, when used for recommendations, is called Pure SVD.




**Remember** that that SVD works as a recommender algorithm only if it is truncated.
If we compute, the full, non truncated SVD of the user rating matrix, when we try to estimate a missing rating we obtain zero, as the exact SVD correspond exactly to the original URM.

# Chapter 10

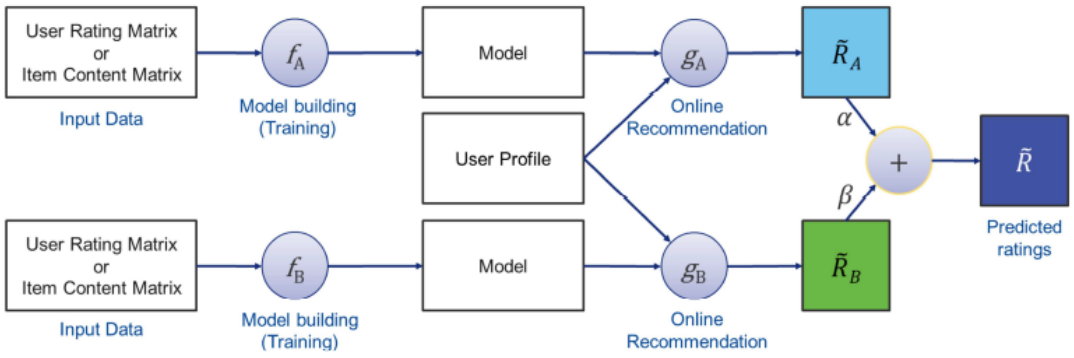
# Hybrid Recommender Systems

The idea behind Hybrid recommender systems is to combine different algorithms, so that, the resulting hybrid algorithm can take advantage of the strengths of each component algorithm.
To better understand this chapter, it is important to have a clear idea of the structure of a generic algorithm *(ref. section 1.4)*.

There are 5 different families of hybrid recommender systems: **Linear Combination**, **List Combination**, **Pipelining**, **Merging Models**, **Co-Training**.

## 10.1 Linear Combination

The most natural way to combine the recommendations of two algorithms, A and B, consists in adding their predicted ratings together by means of a linear combination.




This approach uses linear combination to combine the recommendations of two algorithms, A and B, for a specific user. Firstly, we estimate the ratings for that user by using algorithm A. Then, we compute the estimated ratings using Algorithm B. Finally, we compute the weighted sum between the two estimated ratings.
Each algorithm is used as it is, without any modification. In fact, only their output is used to provide a combined prediction. This construction allows an easy combination of the algorithms and their parallel execution.

**Note** that Linear Combination approach allows to combine results starting from different input data *(For instance, ICM for algorithm A, URM for algorithm B)*

$$\tilde{R} = \alpha \cdot \tilde{R}_A + \beta \cdot \tilde{R}_B$$

Weights, alpha and beta, are hyperparameters that must be carefully tuned in order to have an effective hybrid system.

**PRO**: The main advantage of linear combination is that it is very straightforward to implement.

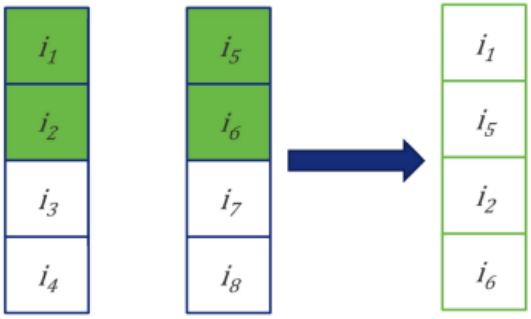
**CONS**: The problem is that the optimal values of the weights are different for each dataset, and must be carefully tuned. Also, if the estimated ratings are on different scales, it will be more difficult to obtain good results. In this case it is particularly important to weight the different components properly.

## 10.2 List Combination

List Combination approach is based on the combination of the recommendation lists. Assuming to have two recommendation lists, each produced by a different algorithm. What we can do, is to take only the highest ranked items from each list, and merge them. The result would be a new Top-N list that can be used for recommendations.

**How to merge two (or more) lists**

One simple yet effective approach is to use ***round-robin***: items are selected starting from the top of each list. We first pick the highest ranked item from list A, as the first element of our hybrid recommendation. Then, moving to list B, we pick the highest ranked item in list B, as the next element of our hybrid recommendation list. Then, turning back to list A, we choose the highest ranked item that has not been selected yet, and so on for the rest of the final list.




The example shows two lists of recommendations, obtained by algorithms A and B, in the context of a Top-4 recommendation. We first pick the highest ranked item from list A, as the first element of our hybrid recommendation. In this example we select item $i_1$. Then, following the process described above, we select item $i_5$ from list B. Then, we go back to list A, choosing $i_2$ and finally $i_6$ from list B. Now the Top-4 hybrid list is complete.

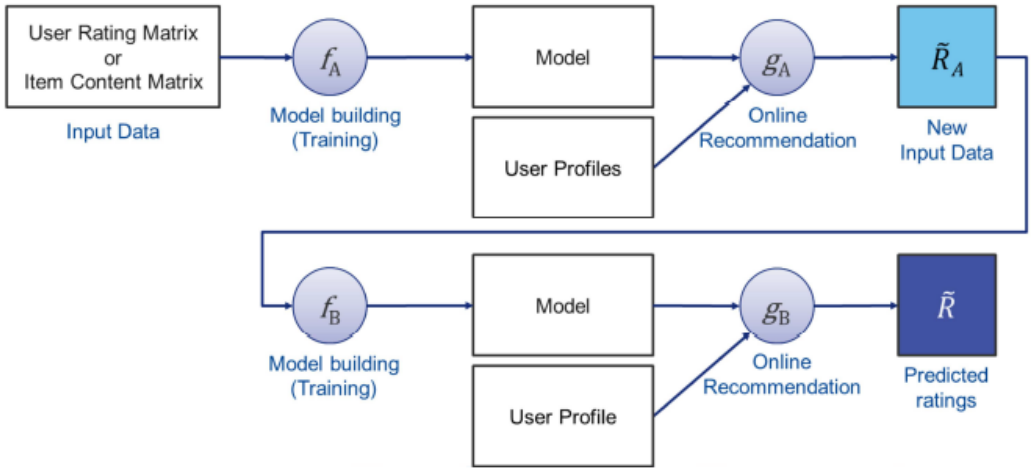
**Note** that there could be some cases which are a little more complex to manage for example when the two (or more) initial lists share some items. In this case the approach should be followed is the same as the one described above. In case of already chosen elements the algorithm continues without (re)selecting it. It will end when the final list is completed.

**PRO**: The list combination method has the advantage that it does not require comparable rating scales between its component algorithms. This makes it useful also with algorithms that produce very different ratings.

**CONS**: It is difficult to weight the relative importance of each list.

## 10.3 Pipelining

The third family of hybridization techniques is pipelining and consists in chaining two or more algorithms, so that, the output ratings of one algorithm is fed as input to the next algorithm in the pipeline. Considering a generic recommender algorithm A, this can take as input either a URM, or an ICM. After building the model, we are able to compute some estimated ratings. In the pipelining approach, the ratings estimated with algorithm A can be used to enrich an existing user rating matrix. This enriched user rating matrix becomes the input for a second algorithm B.

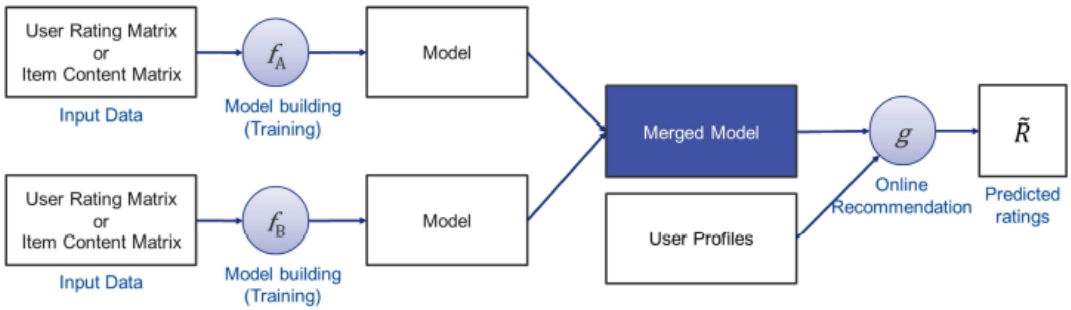



**Note** that except from the first algorithm, all the following ones take as input an enriched URM.

**PRO**: It increases the density of the URM
**CONS**: It is really expensive from a memory point of view and since the first algorithm enriches the URM in input, the following ones may receive noise as input that produce a low quality in the final estimated ratings.

## 10.4 Merging Models

This approach we can use to build a hybrid algorithm by merging two or more models.
***This approach only works if the models have the same structure***.




The model of algorithm A is combined with the model of algorithm B, to obtain a new model that is used for recommendation. The two similarity matrices, S a and S b, are combined by means of a linear combination to obtain the hybrid similarity matrix S. Notice that alpha is a hyperparameter, that must be tuned using one of the numerous techniques available.

## 10.5 Co-Training

Co-Training is a family of Recommender Systems where it is possible to train two (or more) models simoultaneously. It works in a way such that the training of model A influences the model B training too.
This family is similar to the Merging Models family, the difference being that with Co-Training the merged model is obtained directly during the training phase, and algorithms A and B must be trained together.

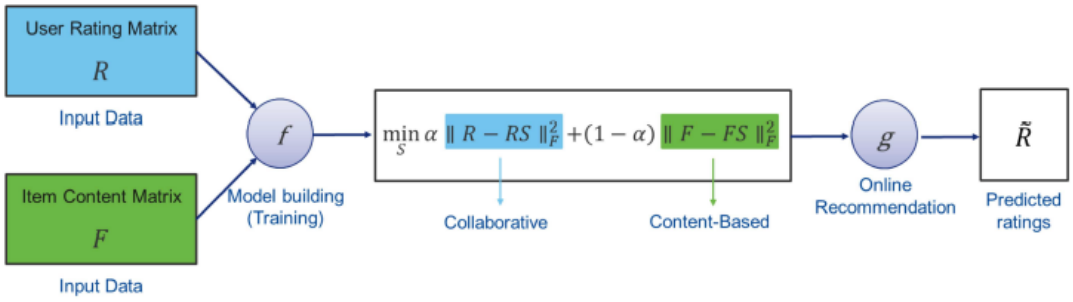
### 10.5.1 S-SLIM

S-SLIM is an example of Co-Training. S-SLIM combines CF and CBF approaches.
To better understand this section, it is important to have a clear idea of SLIM *(ref. chapter 8)*.

On top of SLIM, S-SLIM takes into consideration also the ICM matrix F. The basic idea is that if we have a theoretically perfect similarity matrix, we can predict a missing attribute for an item by looking at the attributes of similar items. Therefore, we can use machine learning to build a content-based similarity matrix, by minimizing some error between the true ICM and the predicted ICM.

#### Combine the two approaches

The optimization formula for S-SLIM includes both the URM and the ICM. In this way, the matrix to be learnt, S, must fit both the URM and the ICM.



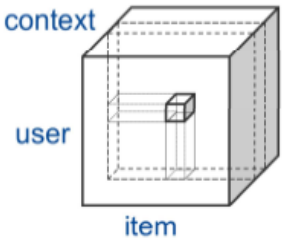

The optimization problem is hence composed of two parts: a collaborative part that represents the basic SLIM algorithm, and a content-based part, where the ICM is taken into account. **Notice** the use of a hyperparameter $\alpha$. This coefficient takes values between 0 and 1.

The choice of the hyperparameter $\alpha$ determines how much importance we are giving to the collaborative part in disfavour of the content-based part, and vice versa.

# Chapter 11

# Context Aware Recommender Systems

Context aware recommender systems include, as an additional information, the context in which a user interacted with an item.
Examples of context can be the hour of day in which a user watched a movie, or the outside weather when a user went to a restaurant.
The easiest way to build a context-aware recommender system is by using tensor factorization. A tensor is a matrix with 3 or more dimensions.Tensor factorization extend matrix factorization by adding a third dimension (or more dimensions) to the user rating matrix.
The first dimension represents the user, the second dimension represents the item, the third, new dimension describes the context.



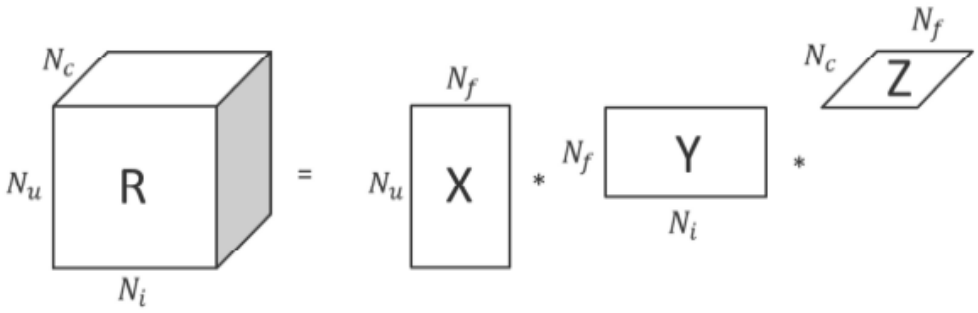

Tensor factorization extends matrix factorization to the contextual dimension. With the tensor factorization, we multiply X and Y by a third matrix, Z, that represents the contextual factors.
Mathematically, we can estimate the rating for user u on item i in context c, as the sum of the products of the corresponding three factors.

$$\tilde{r}_{uic} = \sum_{f} x_{uf} \cdot y_{if} \cdot z_{cf}$$

where f indicates the latent factors and c represents the context. In matrix format, the approximate rating tensor is computed as the product of matrices X, Y and Z.

$$\tilde{R} = X \cdot Y \cdot Z$$

**Note** that this is a tensor product. It is suggested to Google how to solve it in case of doubts for dimensions.




In order to find the tensor factorization, as we did for matrix factorization, we can minimize a loss function computed as the MSE, or two-norm, between the true ratings and the estimated ratings.

$$\min_{X,Y,Z} \left\| R - \tilde{R} \right\|_2 + \lambda_1 \|X\| + \lambda_2 \|Y\| + \lambda_3 \|Z\|$$

Remembers that the regularization terms are required to keep matrices X, Y and Z as sparse as possible to avoid overfitting and to increase the ability of the model to generalize.

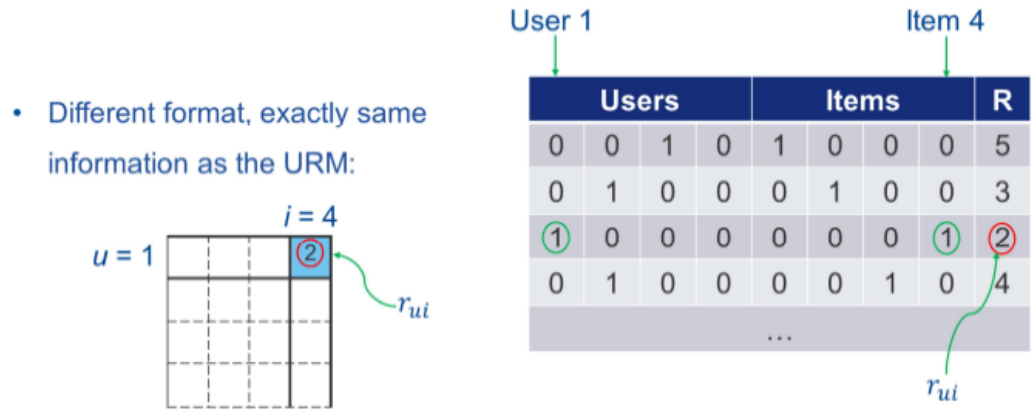
# Chapter 12

# Factorization Machines

Factorization Machines (FM) is a technique of Collaborative Filtering that can be easily extended with side information. It is an extension of a linear model that is designed to capture interactions between features within high dimensional sparse datasets economically.

FMs represent the input data differently than other Collaborative Filtering approaches seen so far. In a FM approach, data is described through a table. This table would have as many rows as many interactions in the URM. The columns in the table are divided into groups, except for the last column, which contains the actual rating of the user for items.
In a Recommender Systems scenario it represent what you want to predict, therefore it is called ***target***.
In a FM approach, all other columns of the table are referred to as ***features***.
Features will be in number equal to the sum of the number of users, the number of items and the number of all the additional information (if present).

In order to uniquely represent users and items, it is used One-Hot Encoding (OHE). This method allows to assign a column to each user and item, by setting the proper column to one and all the others to zero. Therefore, the base features will be binary vectors of user and items indicators, such that each training sample has exactly two non-zero entries *(simple case without side information)* corresponding to the given user/item combination.
For instance, the first column could represent the first user in the URM, the second column could be the second user and so on. By setting the first column to one and others to zero, you specify the first user.




The reported example clearly shows the presence of two different groups of columns (Users and Items) and the rating column expresses the rating for each user/item interaction. The third row represents that the first user gave a 2 rating to the fourth item. In the URM representation, this would correspond to the first row and fourth column having a value of two *(as shown on the left)*.

## 12.1 Estimated Ratings

Based on a Machine Learning approach, the model is trained and used to estimate unknown rating refers to the following structure:

$$\tilde{r}^{(k)} = \omega_0 + \sum_{i=1}^{n} \omega_i \cdot x_i^{(k)} + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \omega_{i,j} \cdot x_i^{(k)} \cdot x_j^{(k)}$$

In the reported model, index k is used to represent the row, while the subscripts i and j refer to columns. The element can be either 0 or 1 according to the OHE. The number n represents the sum of the number of users, the number of items and the number of additional information (if present).

It is important to note that the omega values are not referred to a specific interaction, but they are all referred to the whole table. Therefore, the learning algorithm has to find a way to find these values in a way to reproduce the ratings for every interaction as accurately as possible.

By looking at each of the three components of the model:

1. $\omega_0$ is a constant component that, in a ML approach, represents the *global bias*. It consists of the average of all the ratings.
2. $\sum_{i=1}^{n} \omega_i \cdot x_i^{(k)}$ has the goal to learn, for each column, the corresponding bias.
   Computing it for each column, would result in computing user and item's biases.
   Through the addition of this component, it means that the estimated rating is composed by three NON-zero terms: $\mu$, $b_i$, $b_u$. The sum of these three terms is equivalent to the Global Effect.
3. $\sum_{i=1}^{n} \sum_{j=i+1}^{n} \omega_{i,j} \cdot x_i^{(k)} \cdot x_j^{(k)}$ further improves the rating estimation by considering the relations between features.
   It is important to note that the second sum starts at $j = i + 1$ in order to find *two distinct features* that are both present in that interaction.
   Thus, the corresponding $\omega_{i,j}$ would be effective only for those two values of i and j for modelling the rating.

The overall estimated rating is the sum of the global effect and a *local effect* given by the last component. We can represent the Factorization Machines model, by using a vector notation:

$$\tilde{r}^{(k)} = \omega_0 + \boldsymbol{\omega} \cdot \boldsymbol{x}^{(k)} + \boldsymbol{x}'^{(k)} \cdot W \cdot \boldsymbol{x}^{(k)}$$

where $\mathbf{x}$ is the vector of features, $\boldsymbol{\omega}$ is the vector of unknown biases and $\boldsymbol{W}$ is the matrix of the unknown coefficients.

In this model, we need to learn $\omega_0$, $\boldsymbol{\omega}$, and $\mathbf{W}$ matrix.
The overall number of parameters to be learned is $\#parameters = 1 + n + n^2$.

### W Problem

This model has a relevant problem: $\mathbf{W}$ matrix is huge ($nxn$) and very sparse. We need to reduce $\mathbf{W}$ dimension with factorization. The goal of $\mathbf{W}$ factorization is to approximate $\mathbf{W}$ as the product of two rectangular matrices $\mathbf{A}$ ($nxk$) and $\mathbf{B}$ ($kxn$). $k$ is the number of latent factors and note that $k \ll n$.

The $\mathbf{W}$ factorization brings to approximate $\omega_{i,j}$ as the dot product between the row i of matrix A and the column j of matrix B.

$$\omega_{i,j} = \mathbf{a}_i \cdot \mathbf{b}_j = \sum_{h=1}^{k} a_{i,h} \cdot bh, j$$

By replacing the $\omega_{i,j}$ terms in the estimated rating formula, we obtain

$$\tilde{r}^{(k)} = \omega_0 + \sum_{i=1}^{n} \omega_i \cdot x_i^{(k)} + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{h=1}^{k} a_{i,h} \cdot b_{h,j} \cdot x_i^{(k)} \cdot x_j^{(k)}$$

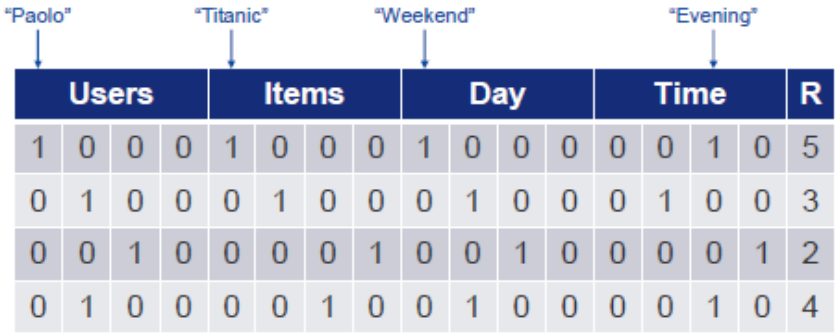
In the end, instead of matrix W with n by n parameters, we have two matrices (A and B), each having $nxk$ parameters. So, the overall number of parameters is now reduced to $\#parameters = 1 + n + 2nk$.

## 12.2 Model extensions

Factorization Machines are better than classical Collaborative Filtering, because FM are very flexible. We can optimize Factorization Machines for various applications, only by adding a different kind of information to the input table, without changing the model itself.

### 12.2.1 Side Information

We can very easily extend the model by adding side information. In order to extend the model, we simply add more columns, containing other types of information, and we do not need to change the equation of the model. Side information can be context, or attributes on items, or attributes on users, and we do not have to change anything in the final formula. Thus, it is important to note that the formula does not depend on the column. Hence, the formula used to learn the parameters does not need to be changed. Suppose that we want to add the context as side information. The following example shows the addition of the date of the week and the time in which users watched and rated a movie.




In the reported example, user "Paolo" watched "Titanic" on "Weekend", in the "Evening". A value of 1 is in the proper columns that represent user "Paolo", item "Titanic", "weekend", and "evening". At the end, in the last column for the ratings, there is the rating that "Paolo" gave to the movie, that is 5.

### 12.2.2 Multiple users

FMs approach is really powerful. So far, we have seen approaches that take into account a single-user/item interaction. FM allows to take into account the possibility that more people are together involved in an interaction. This is really easy to model in a FM approach since it just need to put a value of 1 in each of the user columns involved.

## 12.3 Model problems

### 12.3.1 Importance of the attributes

One of the problem that can be encountered is that there are some attributes that are more important than others on the same item.

Therefore, a possible solution is to put **real-values** in the table to express how much a feature is important.

### 12.3.2 Imbalance problem

This problem happens, because a dataset with implicit rating would only have positive values.
This means we have no information on the items that users does not like. In other words, dealing with implicit ratings, the algorithm that classifies items into the two categories (liked and disliked items), does not have any information on the disliked items.

Since there is no information on negative interactions, the algorithm would learn to rate every interaction with value of one.

**Solution**: to solve this problem, we need to create a balance between positive and negative samples. By adding equal number of negative interactions, in which users did not like an item, we provide the algorithm with balanced data, to learn the preference of the users.

| Users | | | | Items | | | | Context | | | | Content | | | | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| ... | | | | | | | | | | | | | | | | |

Usually, it is convenient to balance the number of ratings for each user. So, if user X rated three items, the balance problem would consists in putting three non relevant items for user X.

To get **information about the negative ratings** of the users we can simply choose randomly some of the missing ratings, and insert them into the table.
The problem with this method is that for the missing ratings, we do not know whether the user saw the item and did not like it or, otherwise, whether the user has never had any kind of interaction with the item, so that the item could potentially be even a good recommendation.
Actually, we have no way of knowing this information, but we are here assuming Missing ratings As Negative (MAN).
The logic behind using this assumption, is that, if we choose an item randomly, there is a high probability that the user won't like it. This is because in an actual dataset, the number of items is generally very large, and a user only likes a small subset of those items.

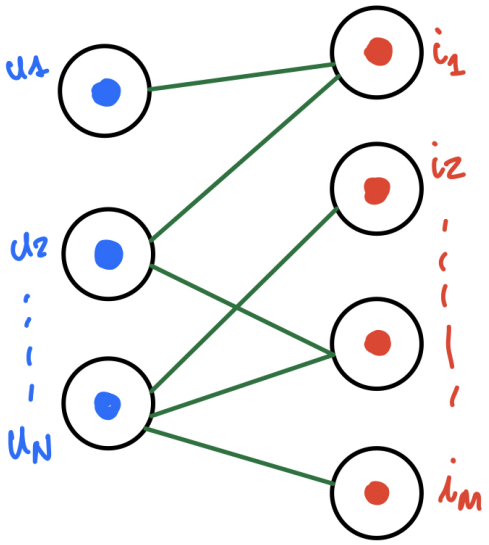
# Chapter 13

# Graph-Based Recommender Systems

The idea behind Graph-Based Recommender Systems is that we can decide the input data by using a graph. Through this approach we can estimate how much a feature is important for a user.

The idea is that if we arrive to an item only after a certain amount of jumps, it is unlikely that a user would like that item.
The average number of times you pass through a node is used to measure how much that node is important.

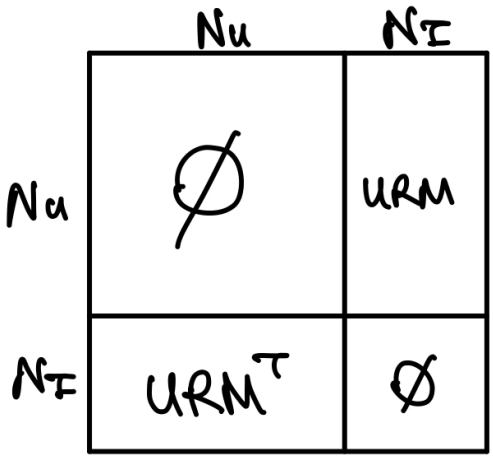
We can represent users, items and their interaction with a graph.




Links can **only** be from users to items.
If we want to learn what does an user like, we can moved to a linked item (that the user likes for sure since the two nodes are connected) and them we can jump back to a random user.
This means that this other user liked that item too. Therefore, by exploring what are the items the new user explored, we can find items that our first user is more likely to like.

The graph based approach allows to easily add further information by simply adding nodes categories to the graph *(for instance, a group of nodes for the context)*.
**Note** that this approach works even with explicit ratings. It will use weighted links.

## 13.1 Incidence Matrix




From now on, we consider user and items simply as nodes. We need to estimate the probability to jump into node j when we are at node i. Computing this probability we are doing two strong assumption:

1. The probability is equally distributed among the different links

2. The probability to pass from a node to another one does not change overtime *(Steady State Assumption)*

$$P_{ij} = \frac{G_{ij}}{\sum_k G_{ik}}$$

where $G_{ij}$ would be 0 or 1 and indicates the presence of a link between the nodes i and j; and the denominator indicates the number of links *from* node i.
Our goal is to compute the probability to pass through a specific node. Indicating with $\pi_i^{(k)}$ the probability of being at node i after k jumps, we can compute it as

$$\pi_i^{(k)} = \sum_j \pi_j^{(k-1)} \cdot P_{ji}$$

We will consider the probability as a ***measure of relevance***: if it is more likely that we pass from a node respect to another one, that node would be more relevant than the other one.

Note that at each stage $k$: $\sum_i \pi_i = 1$.
After an infinte number of jumps, the difference between $k$ and $k+1$ disappears and, the probability $\pi^{(\infty+1)} = \pi^{(\infty)}$. Therefore, considering the vectors of probabilities (vectors containing the probabilities of being at each of the nodes in the graph) we would have a linear set of equation $\vec{\pi} = P \cdot \vec{\pi}$ and a solution would be $\vec{\pi} = \vec{\emptyset}$ that is obviously meaningless for our purposes. Therefore, we can remove an equation from the set and substitute it with the fundamental property we described above: $\sum_i \pi_i = 1$

We want to make recommendations for a user. If we do not consider the starting node, the recommendations would be equivalent to a top-popular recommender algorithm. Therefore, we can add to our model a certain probability to ***restart***.

$$\pi = \gamma \pi P + (1-\gamma)\pi_0$$

where $\pi_0$ is the vector of restarting probabilities and $\gamma$ is a parameter that indicates at each stage the probability to restart the model.

## 13.2 $P_3^\alpha$

Turning back to the differentiation between user and item nodes, starting from a user, after performing 3 steps, we will always be in an item node (every odd number would bring to the same node category).

Our goal is to compute the probability of being on a certain node after 3 steps. If we perform only three jumps the model is equivalent to an Item-Based Collaborative Filtering Recommender algorithm.

The probability to jump from an item $i$ to user $v$ is equal to the ratio between the presence of the link between the presence of the link between the two nodes and the number of ratings involving item $i$ (elevated at the power of $\alpha$ that allows to normalize the probabilities)

$$P_{iv} = \frac{r_{iv}}{N_i^\alpha}$$

The probability to jump from a user $v$ to an item $j$ is equal to the ratio between the presence of the link between the two nodes and the number of items rated by that user (elevated at the power of $\alpha$ that allows to normalize the probabilities).

$$P_{vj} = \frac{r_{vj}}{N_v^\alpha}$$

We can therefore compute the ***similarity*** between the two items as

$$s_{ij} = \sum_v P_{iv} \cdot P_{vj}$$

that is clearly equivalent to an Item-Based CF approach.