

Introduction to Recommendation Systems

Till recently, people generally tended to buy products recommended to them by their friends or the people they trust. This used to be the primary method of purchase when there was any doubt about the product. But with the advent of the digital age, that circle has expanded to include online sites that utilize some sort of recommendation engine.

A recommendation engine filters the data using different algorithms and recommends the most relevant items to users. It first captures the past behaviour of a customer and based on that, recommends products which the users might be likely to buy.

If a completely new user visits an e-commerce site, that site will not have any past history of that user. So how does the site go about recommending products to the user in such a scenario? One possible solution could be to recommend the best selling products, i.e. the products which are high in demand. Another possible solution could be to recommend the products which would bring the maximum profit to the business.

If we can recommend a few items to a customer based on their needs and interests, it will create a positive impact on the user experience and lead to frequent visits. Hence, businesses nowadays are building smart and intelligent recommendation engines by studying the past behavior of their users.

Challenges when building recommendations systems

When we try to recommend items to users, we face a few fundamental challenges:

- **Data Sparsity:** There are lots of products to recommend to many users and it is unlikely that a user will ever try out a large fraction of products. Instead, probably a few items are demanded by many users, but many only by a few.
- **Cold Start:** We need to be able to give recommendations to users about which we only have scarce data (if at all).
- **Accurate, but also diverse predictions:** We want to give useful recommendations in the sense that they match the user's preferences, but also that the recommendation contains some novelty for the user.
- **Evaluation:** Evaluation is difficult and might differ from algorithm to algorithm.

- Scalability: We need to be able to give recommendations on the spot even though there might be millions of users and items which we have to analyze carefully.
- User interface: Users want to know why they get particular recommendations.
- Vulnerability to attacks: We do not want our recommendation system to be abused for promoting or inhibiting particular items.
- Temporal resolution: Tastes and preferences do not remain the same over time.

To face all this, regularly we'll talk about users and items. In most cases, we are going to predict a certain rating for each possible pair of user and item.

If the user already gave some rating we can compare it to our prediction:

- True rating: r_{ui}
- Predicted rating: \hat{r}_{ui}

Collecting data to make our recommendations

When thinking about creating a Recommendation System, we can gather data from several sources. In an e-commerce, we could find relevant data about product ratings, upvotes/downvotes or relevant feedback from customers. Also, we could use traffic data about access logs, sessions lengths, time spent on pages, clicks or purchase history. While if we are working for a social platform, we could use follows/unfollows, favourites pages/posts, playlists/lists created, between others.

So, taken into account that there're a lot of sources, the first and most crucial step for building a recommendation engine is to define which data are we going to use. The data can be collected by two means: explicitly and implicitly.

Explicit data is information that is provided intentionally, i.e. input from the users such as movie ratings. Implicit data is information that is not provided intentionally but gathered from available data streams like search history, clicks, order history, etc.

Designing our Recommendation System

There are basically two kinds of approaches:

1. Content-based filtering

This algorithm recommends products which are similar to the ones that a user has liked in the past. For example, if a person has liked the movie “Inception”, then this algorithm will recommend movies that fall under the same genre. But how does the algorithm understand which genre to pick and recommend movies from?

The content-based filtering algorithm uses one similarity measure (we’ll talk more about this later) and based on that measures (which usually ranges between -1 to 1), the movies are arranged in descending order and one of the two below approaches is used for recommendations:

- Top-n approach: where the top n movies are recommended (Here n can be decided by the business)
- Rating scale approach: Where a threshold is set and all the movies above that threshold are recommended

A major drawback of this algorithm is that it is limited to recommending items that are of the same type. It will never recommend products which the user has not bought or liked in the past. So if a user has watched or liked only action movies in the past, the system will recommend only action movies. It’s a very narrow way of building an engine.

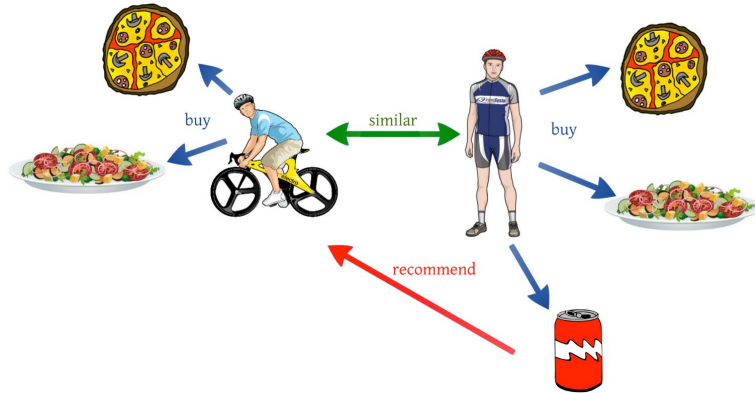
To improve on this type of system, we need an algorithm that can recommend items not just based on the content, but the behaviour of users as well.

2. Collaborative filtering

Let us understand this with an example. If person A likes 3 movies, say Interstellar, Inception and Predestination, and person B likes Inception, Predestination and The Prestige, then they have almost similar interests. We can say with some certainty that A should like The Prestige and B should like Interstellar. The collaborative filtering algorithm uses “User Behavior” for recommending items. This is one of the most commonly used algorithms in the industry as it is not dependent on any additional information.

a. User-user collaborative filtering

This algorithm first finds the similarity score between users. Based on this similarity score, it then picks out the most similar users and recommends products which these similar users have liked or bought previously.



The prediction $P_{u,i}$ is given by:

$$P_{u,i} = \frac{\sum_v (r_{v,i} * s_{u,v})}{\sum_v s_{u,v}}$$

Here,

- $P_{u,i}$ is the prediction of an item
- $R_{v,i}$ is the rating given by a user v to a movie i
- $S_{u,v}$ is the similarity between users

However, this algorithm is quite time-consuming as it involves calculating the similarity for each user and then calculating a prediction for each similarity score. One way of handling this problem is to select only a few users (neighbours) instead of all to make predictions, i.e. instead of making predictions for all similarity values, we choose only few similarity values. There are various ways to select neighbours:

- Select a threshold similarity and choose all the users above that value
- Randomly select the users
- Arrange the neighbours in desc. order of their similarity value and choose top-N users
- Use clustering for choosing neighbors

In case of selecting of looking at the k most similar users, the predictions equations would be transformed into the following:

$$\hat{r}_{ui} = \mu_u + \frac{\sum_{v \in N_i^k(u)} \text{sim}(u, v)(r_{vi} - \mu_v)}{\sum_{v \in N_i^k(u)} \text{sim}(u, v)}$$

Here $N_i^k(u)$ denotes the k most similar users to user u who rated item i .

As an example, let's take user similarity (e.g. correlation similarity) and we consider the two nearest neighbours of user 1. Let's say we have

$$\mu_1 = 3, \mu_2 = 2, \mu_3 = 4, \text{sim}(1, 2) = 0.8, \text{sim}(1, 3) = 0.5,$$

and want to predict for item 1 with

$$r_{21} = 3.2 \text{ and } r_{31} = 3.8.$$

Then we obtain

$$r_{11} = 3 + \frac{0.8(3.2 - 2) + 0.5(3.8 - 4)}{0.8 + 0.5} = 3 + \frac{0.8 \cdot 1.2 + 0.5 \cdot (-0.2)}{1.3} \approx 3.66$$

b. Item-item collaborative filtering

In this algorithm, we compute the similarity between each pair of items.



This algorithm works similar to user-user collaborative filtering with just a little change – instead of taking the weighted sum of ratings of “user-neighbors”, we take the weighted sum of ratings of “item-neighbours”. The prediction is given by:

$$P_{u,i} = \frac{\sum_N (s_{i,N} * R_{u,N})}{\sum_N (|s_{i,N}|)}$$

Similarity measures

In any kind of algorithm, the most common similarity measure is finding the cosine of the angle between vectors, i.e. cosine similarity. Suppose A is user's A list of movies rated and B is user's B list of movies rated, then the similarity between them can be calculated as:

$$\text{similarity}(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^n A_i \times B_i}{\sqrt{\sum_{i=1}^n A_i^2} \times \sqrt{\sum_{i=1}^n B_i^2}}$$

Mathematically, the cosine similarity measures the cosine of the angle between two vectors projected in a multi-dimensional space. When plotted on a multi-dimensional space, the cosine similarity captures the orientation (the angle) of each vector and not the magnitude. If you want the magnitude, compute the Euclidean distance instead.

The cosine similarity is advantageous because even if the two similar documents are far apart by the Euclidean distance because of the size (like one word appearing a lot of times in a document or a user seeing a lot of times one movie) they could still have a smaller angle between them. Smaller the angle, the higher the similarity.

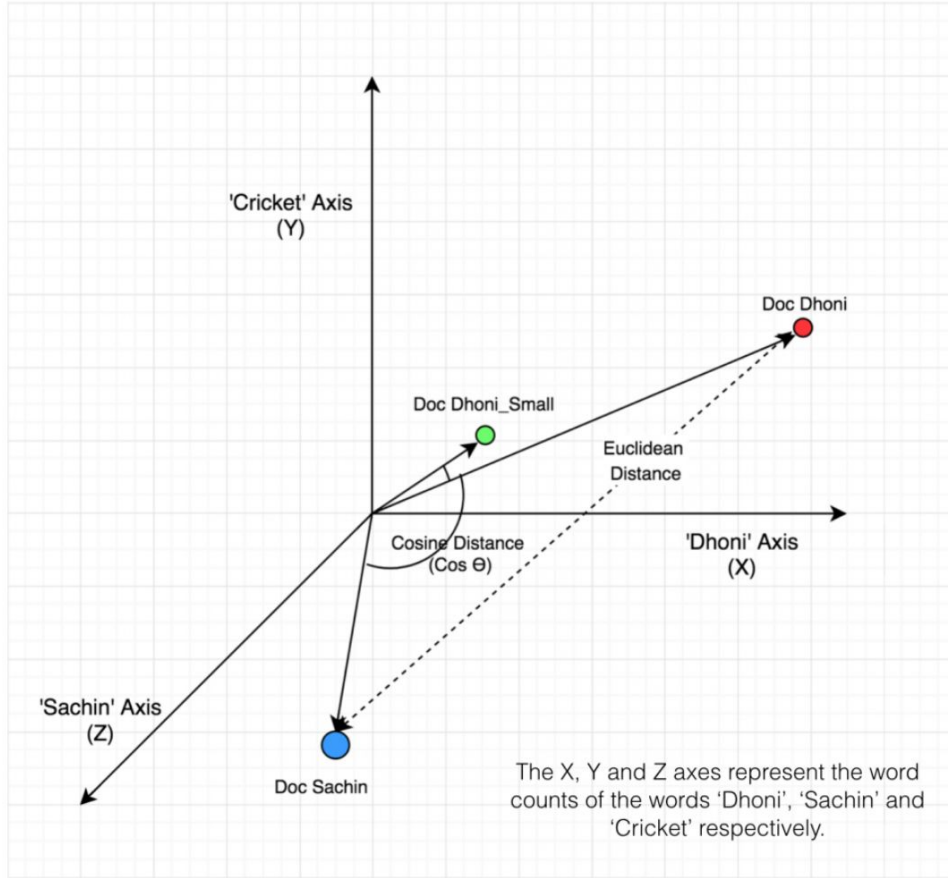
Take the following example from www.machinelearningplus.com:



Considering only the 3 words from the above documents: 'sachin', 'dhoni', 'cricket'

Doc Sachin: Wiki page on Sachin Tendulkar	Doc Dhoni: Wiki page on Dhoni	Doc Dhoni_Small: Subsection of wiki on Dhoni
Dhoni - 10	Dhoni - 400	Dhoni - 10
Cricket - 50	Cricket - 100	Cricket - 5
Sachin - 200	Sachin - 20	Sachin - 1

The above image is counting the number of appearances of the word 'sachin', 'dhoni' and 'cricket' in the three documents shown. According to that, how would we plot these vectors and how would we measure different kind of distances? Let's see:

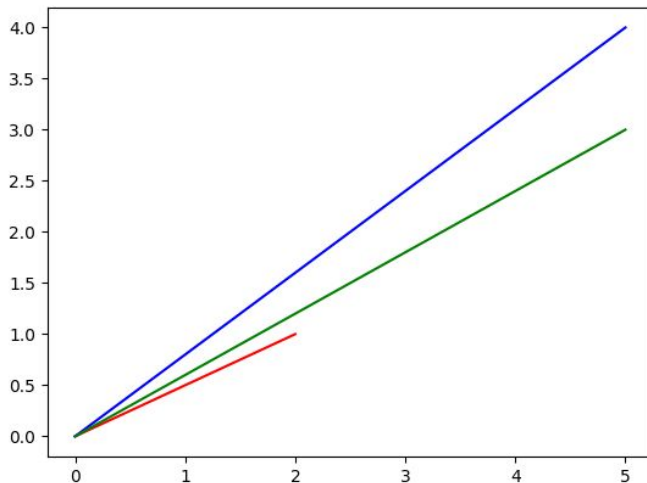


Now, Regular Cosine Similarity by the definition reflects differences in direction, but not the location. Therefore, using cosine similarity metric for item-based CF approach does not consider the difference in ratings of users. Adjusted cosine similarity offsets this drawback by subtracting respective user's average rating from each co-rated pair, and is defined as below:

$$sim(i, j) = \frac{\sum_{u \in U} (R_{u,i} - \bar{R}_u)(R_{u,j} - \bar{R}_u)}{\sqrt{\sum_{u \in U} (R_{u,i} - \bar{R}_u)^2} \sqrt{\sum_{u \in U} (R_{u,j} - \bar{R}_u)^2}}.$$

Let's assume a user give scores in 0~5 to two movies.

```
from scipy import spatial
import numpy as np
a = np.array([2.0,1.0])
b = np.array([5.0,3.0])
1 - spatial.distance.cosine(a,b)
#-----
# 0.99705448550158149
#-----
c = np.array([5.0,4.0])
1 - spatial.distance.cosine(c,b)
#-----
# 0.99099243041032326
#-----
```



Intuitively we would say user b and c have similar tastes, and a is quite different from them. But the regular cosine similarity tells us a wrong story. Let's calculate the Adjusted Cosine Similarity, first minus the mean of x and y:

```
mean_ab = sum(sum(a,b)) / 4
# mean_ab : 3.5
# adjusted vectors : [-1.5, -2.5] , [1.5, -0.5]
1 - spatial.distance.cosine(a - mean_ab, b - mean_ab)
#-----
# -0.21693045781865616
#-----
mean_cb = sum(sum(c,b)) / 4
# mean_cb : 6.5
# adjusted vectors : [-1.5, -3.5] , [-1.5, -2.5]
1 - spatial.distance.cosine(c - mean_cb, b - mean_cb)
#-----
# 0.99083016804429891
#-----
```


Other methods that can be used to calculate the similarity are:

- Euclidean distance: similar items will lie in close proximity to each other if plotted in n-dimensional space.

$$\text{Euclidean Distance} = \sqrt{(x_1 - y_1)^2 + \dots + (x_N - y_N)^2}$$

- Pearson's correlation or correlation similarity: it tells us how much two items are correlated. Higher the correlation higher will be also the similarity.

$$\text{sim}_{\cos}(u, v) = 1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{\|u\| \|v\|}$$

- Mean squared difference: is about finding the average squared divergence in between users ratings. MSE puts more weight into penalizing larger errors.

$$\text{msd}(u, v) = \frac{1}{|I_{uv}|} \sum_{i \in I_{uv}} (r_{ui} - r_{vi})^2$$

And then:

$$\text{msd_sim}(u, v) = \frac{1}{\text{msd}(u, v) + 1}$$

Where $|I_{uv}|$ is just the number of items rated by both users u and v .

Examples of user-user and item-item similarities

- User-user similarity

Consider the following matrix:

User/Movie	x1	x2	x3	x4	x5	Mean User Rating
A	4	1	–	4	–	3
B	–	4	–	2	3	3
C	–	1	–	4	4	3

Here we have a user movie rating matrix. To understand this in a more practical manner, let's find the similarity between users (A, C) and (B, C) in the above table. Common movies rated by A and C are movies x2 and x4 and by B and C are movies x2, x4 and x5.

Let's find the Pearson's correlation or correlation similarity:

$$r_{AC} = [(1-3)*(1-3) + (4-3)*(4-3)] / [((1-3)^2 + (4-3)^2)^{1/2} * ((1-3)^2 + (4-3)^2)^{1/2}] = 1$$

$$r_{BC} = [(4-3)*(1-3) + (2-3)*(4-3) + (3-3)*(4-3)] / [((4-3)^2 + (2-3)^2 + (3-3)^2)^{1/2} * ((1-3)^2 + (4-3)^2 + (4-3)^2)^{1/2}] = -0.866$$

The correlation between user A and C is more than the correlation between B and C. Hence users A and C have more similarity and the movies liked by user A will be recommended to user C and vice versa.

- Item-item similarity

User/Movie	x1	x2	x3	x4	x5
A	4	1	2	4	4
B	2	4	4	2	1
C	–	1	–	3	4
Mean Item Rating	3	2	3	3	3

Here the mean item rating is the average of all the ratings given to a particular item (compare it with the table we saw in user-user filtering). Instead of finding the user-user similarity as we saw earlier, we find the item-item similarity.

To do this, first we need to find such users who have rated those items and based on the ratings, similarity between the items is calculated. Let us find the similarity between movies (x1, x4) and (x1, x5). Common users who have rated movies x1 and x4 are A and B while the users who have rated movies x1 and x5 are also A and B.

$$C_{14} = [(4-3)*(4-3) + (2-3)*(2-3)] / [((4-3)^2 + (2-3)^2)^{1/2} * ((4-3)^2 + (2-3)^2)^{1/2}] = 1$$

$$C_{15} = [(4-3)*(4-3) + (2-3)*(1-3)] / [((4-3)^2 + (2-3)^2)^{1/2} * ((4-3)^2 + (1-3)^2)^{1/2}] = 0.94$$

The similarity between movie x1 and x4 is more than the similarity between movie x1 and x5. So based on these similarity values, if any user searches for movie x1, they will be recommended movie x4 and vice versa. Before going further and implementing these concepts, there is a question which we must know the answer to – what will happen if a new user or a new item is added in the dataset? It is called a Cold Start. There can be two types of cold start:

- Visitor Cold Start
- Product Cold Start

Visitor Cold Start means that a new user is introduced in the dataset. Since there is no history of that user, the system does not know the preferences of that user. It becomes harder to recommend products to that user. So, how can we solve this problem? One basic approach could be to apply a popularity based strategy, i.e. recommend the most popular products. These can be determined by what has been popular recently overall or regionally. Once we know the preferences of the user, recommending products will be easier.

On the other hand, Product Cold Start means that a new product is launched in the market or added to the system. User action is most important to determine the value of any product. More the interaction a product receives, the easier it is for our model to recommend that product to the right user. We can make use of Content based filtering to solve this problem. The system first uses the content of the new product for recommendations and then eventually the user actions on that product.

Evaluation of Recommendation Systems

- **MAE and MSE**

We can compare all existing ratings to our prediction for example using the root mean squared error (RMSE) or the mean absolute error (MAE):

$$\text{MAE} = \frac{1}{|R|} \sum_{(u,i) \in R} |r_{ui} - \tilde{r}_{ui}|$$
$$\text{RMSE} = \left(\frac{1}{|R|} \sum_{(u,i) \in R} (r_{ui} - \tilde{r}_{ui})^2 \right)^{1/2}$$

Here, R stands for the set of all user-item pairs. $|\cdot|$ indicates the cardinality of the set (here the number of user-item pairs).

- **Correlations**

Alternatively one can use correlations between true and predicted values for model evaluation:

the Pearson correlation
the Spearman rank correlation
Kendall's tau

You can call these three for example with panda's `.corr()` function by setting the method argument.

The above scores are alright to obtain a model assessment if we have explicit ratings, but in the case of implicit ratings, we might only be able to rank the items. In general, we would like to recommend the top-ranked items, but we have to evaluate if the top-ranked ones are really the ones relevant to the user, or if for some irrelevant items we predicted higher ratings. In that regard, we can use the usual classification metrics.

- **Precision@k and recall@k**

Often users will not really care about all the rating predictions we are making, but instead they will have a major interest only in a few top-ranked items, let's say the k top-ranked items. So it is appropriate to take only these k ratings into account. We then ask how many of these k items are relevant to the user. The relevance is in general difficult to measure, but we can for example ask how many out of the k top-ranked items have a score beyond a certain threshold.

We can then define the so-called precision@k and recall@k for k recommended items:

$$\text{precision@k} = \frac{\text{Recommended items that are relevant}}{\text{Recommended items}}$$

$$\text{recall@k} = \frac{\text{Recommended items that are relevant}}{\text{Relevant items}}$$

Out of these scores we can define an F1@k score in the usual way.

- **Inter-user diversity**

We can compare how similar the recommendations are that we make for different users. We would like our recommender to make individual predictions based on user preferences, so predicting always the same top items would not be a good sign.

We can measure the inter-user diversity by calculating the cosine-similarity between the k top-ranked items and then average over all user pairs.

- **Intra-user diversity**

We can measure how similar the k items are that we recommend to a particular user to obtain the intra-user diversity:

$$I_u(k) = \frac{1}{k(k-1)} \sum_{i \neq j} \text{sim}(\text{item}_i, \text{item}_j)$$

Averaging over all users gives the mean intra-similarity of the recommendation list.

- **Novelty**

For example we could take the degree of each item in the bipartite user-item network and average this degree over the recommendation list for each user before averaging these numbers over all users:

$$\text{Novelty}(k) = \frac{1}{Mk} \sum_{u=1}^M \sum_{i \text{ in top } k \text{ of } u} \text{degree}_i$$

- **Coverage**

Finally we can measure the so-called coverage, the fraction of all the distinct items N_{distinct} that appear in all of our top-k recommendation lists:

$$\text{Coverage}(k) = \frac{N_{\text{distinct}}}{N}$$

- **Baseline Prediction**

As we are predicting ratings \hat{r}_{ui} of user u on item i , the baseline should be the mean of all ratings, μ . And as we will always be considering a specific user or a specific item, we can determine how much each user's or item's ratings are above the average. Therefore we add a bias term b_u for each user and b_i for each item, so that our baseline is:

$$\text{baseline}_{ui} = \mu + b_u + b_i$$

Let's say for example that the average rating is $\mu=3.52$. Our user is very enthusiastic and on average evaluates items better than the average user by $b_u=0.3$. The item is very popular receiving above average ratings with $b_i=0.5$. So we would have a baseline prediction of 4.32.

Slope One Predictor

This is just another way of predicting for which we will obtain a number out of the combination of both the information from other users who rated the same item and from the other items rated by the same user to predict a rating:

$$\hat{r}_{ui} = \mu_u + \frac{1}{|R_i(u)|} \sum_{j \in R_i(u)} \text{dev}(i, j)$$

where $R_i(u)$ is the set of items rated by user u and the average difference between the ratings of item i and j is:

$$\text{dev}(i, j) = \frac{1}{|U_{ij}|} \sum_{u \in U_{ij}} (r_{ui} - r_{uj})$$

and U_{ij} is the set of all users that have rated both items i and j .

Example

Consider the following example of ratings

	item 1	item 2
user 1	2	1.8
user 2	1	?

Then we have

$\mu_{\text{user2}} = 1, |U_{12}| = 1, r_{11} = 2, r_{12} = 1.8$ and

$$r_{22} = 1 + \frac{1}{1}(1.8 - 2) = 0.8$$


Singular Value Decomposition

One way to handle the scalability and sparsity issue created by CF is to leverage a latent factor model to capture the similarity between users and items. In short, we are talking about applying PCA on our matrix of users and ratings. Just as in any other case, PCA can reduce our original dataset into a much smaller set of features which describes the variance in our data. And often, the dimensions itself represent features that humans have learned to associate with whatever

we're trying to recommend. For example, in movies, how romantic it is, how funny it is, etc. PCA will find latent features in the data, that differentiate items between them. PCA won't know what all these features represent, but it will find them anyway.

Let's take the following example:

	Indiana Jones	Star Wars	Empire Strikes Back	Incredibles	Casablanca
Bob	4	5	5	4	4
Ted	3	3	3	5	4
Ann	4	5	5	5	2




	"Action"	"Sci-Fi"	"Classic"
Bob	0.3	0.5	0.2
Ted	0.1	0.1	0.8
Ann	0.3	0.6	0.1

This matrix it's gonna be U, which describes typical users for each latent feature we produce.

And such as we can run PCA on our users-ratings matrix to find typical kinds of users, we can flip things around and run PCA to find profiles of typical kinds of movies. We call this data the transpose matrix of our original rating matrix, or RT in short:

	Bob	Ted	Ann
Indiana Jones	4	3	4
Star Wars	5	3	5
Empire Strikes Back	5	3	5
Incredibles	4	5	5
Casablanca	4	4	2



	"Action"	"Sci-Fi"	"Classic"
Indiana Jones	0.6	0.3	0.1
Star Wars	0.4	0.6	0
Empire Strikes Back	0.4	0.6	0
Incredibles	0.8	0.2	0
Casablanca	0.2	0	0.8

The resulting matrix after applying PCA it's gonna be M. And again, PCA doesn't know anything about the movies but can classify them according to its latent features.

So how do these matrices, that have typical kind of users and typical kind of movies, help us to predict ratings? Well, it results that our original rating matrix is a result of U and M:

$$R = M\Sigma U^T$$

So if we have M and we have U we can reconstruct R in such a way of fill in the blanks for ratings we don't have. This is called matrix factorization.

The sigma symbol in the middle is a diagonal matrix with the eigenvalues of R on the diagonal, and it will help us to scale our values into the correct scale according to our rating matrix. It describes the strength of each latent factor.

And once we have M and U, we could also do predictions for a user just multiplying the corresponding row in M for the user and the corresponding column in U^T for the item.

When using SVM or Singular Value Decomposition, what we're doing is finding M, sigma and U^T together, all at once very efficiently. So what SVM is doing is applying PCA on both the users and the items, and giving us the matrices we need to get the ratings we want.

But how does it compute M and U^T when we have incomplete data? We cannot run PCA with missing values. We need a complete table to use Principal Component Analysis. In the beginning, people just started completing the missing cells with the mean or some other reasonable value. However, there's a better way of doing this. Suppose the following example:

	Indiana Jones	Star Wars	Empire Strikes Back	Incredibles	Casablanca
Bob	4	5	?	?	?
Ted	?	?	?	?	1
Ann	?	5	5	5	?

$$R = M\Sigma U^T$$

$$R_{Bob, Empire Strikes Back} = M_{Bob} \cdot U_{Empire Strikes Back}^T$$

We can treat this a minimization problem when we try to find the values for those incomplete rows and columns, that best minimize the errors.