Module 19: Recommendation Systems

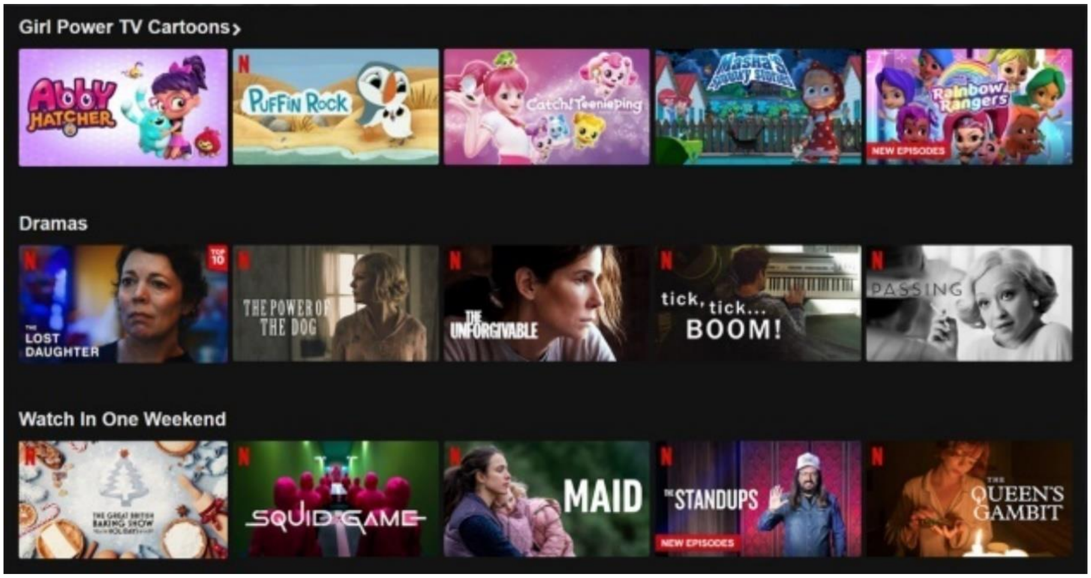
Quick Reference Guide

Learning Outcomes:

1. Compute missing ratings from a rating dataset
2. Implement the Simon Funk SVD gradient descent algorithm for collaborative filtering
3. Evaluate the trade-off between the number of factors and model performance
4. Use the SURPRISE library to build and analyze recommender systems on real datasets
5. Recommend items to users based on predicted ratings and item similarity
6. Diagnose a real-life recommendation system

Introduction

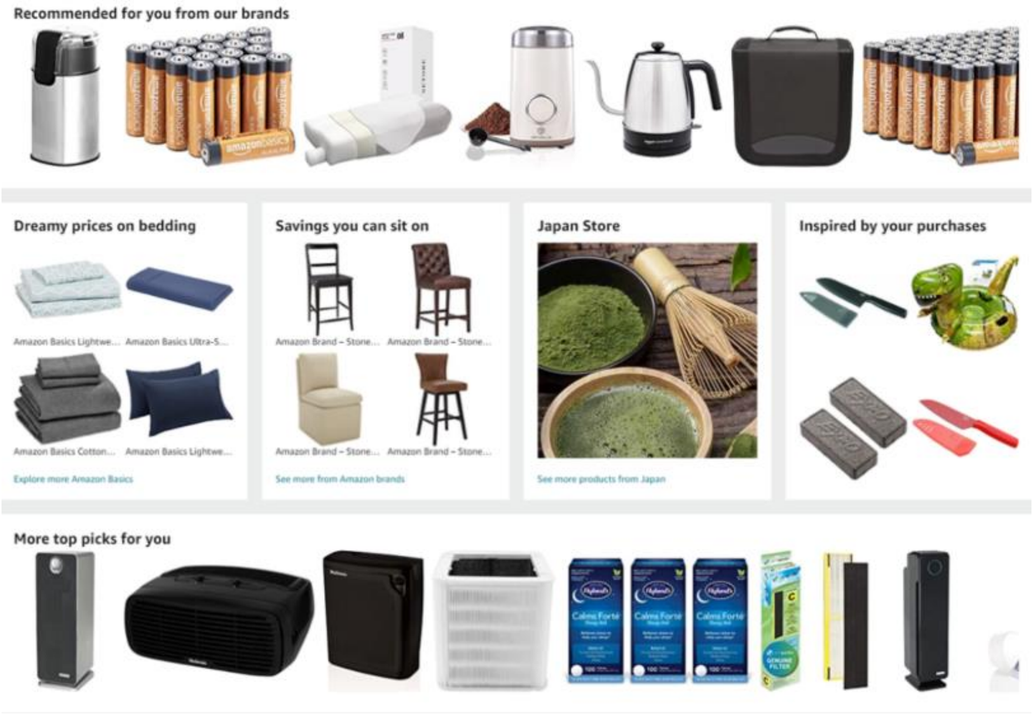
Netflix is a video streaming service available around the world, which has tens of thousands of movies and TV shows. When you log into Netflix, you are presented with recommendations for shows to watch in several categories.



In this case, one category is Girl Power TV Cartoons, with the most prominent rating being for 'Abby Hatcher.' This category could be recommended based on the viewing habits of a younger daughter, for example. The next category is Dramas. You have never seen any of these shows, but Netflix would like you to watch them. And in the bottom row, you see Watch in One Weekend. You have seen two of these shows: 'The Great British Baking Show,' and 'The Queen's Gambit.'

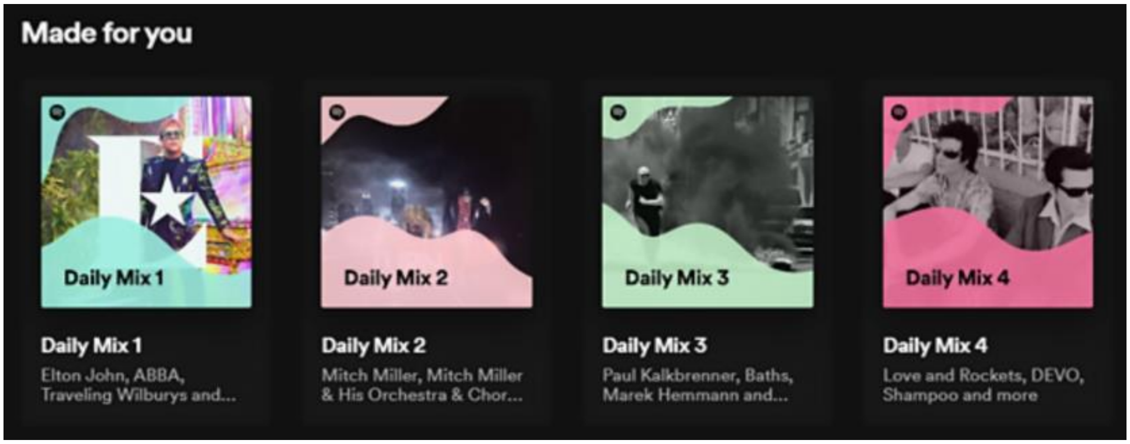
So, how did Netflix decide to recommend these specific categories? And in these categories, how did it decide to recommend these shows, and in this order? The decision was made by a **recommender system**, which knows about your previous viewing habits, including any ratings you explicitly gave the shows with the thumbs-up or thumbs-down buttons.

Now consider what happens when you go to the online product marketplace, Amazon. Here you see a number of categories, each of which contains a set of items, all in the hope of enticing you into making a purchase.



So, as someone who recently purchased a pack of batteries and a water boiler, you see that the algorithm is keen for you to acquire more of these, as well as a coffee grinder. The bedding, chairs, or Japanese products could be recommended because they are popular on the platform at the moment. Suppose you recently bought knives and whiteboard markers, and a month ago purchased an inflatable unicorn sled. So that explains the 'Inspired by your purchases' category showing knives and a large inflatable dinosaur. Again, a recommender system is behind all of this. Some model of you exists and it wants you to buy these goodies.

As a final example, consider the music streaming service, Spotify. When you open Spotify, among the many recommendations that it makes are daily mixes that exist for only one day. Here you see four different daily mixes, each of a different genre.


Made for you
Daily Mix 1
Daily Mix 2
Daily Mix 3
Daily Mix 4
Daily Mix 1
Elton John, ABBA, Traveling Wilburys and...
Daily Mix 2
Mitch Miller, Mitch Miller & His Orchestra & Chor...
Daily Mix 3
Paul Kalkbrenner, Baths, Marek Hemmann and...
Daily Mix 4
Love and Rockets, DEVO, Shampoo and more

The first mix is '70s, '80s popular rock music artists. The second is artists and men's choruses from the late 1950s, early 1960s. The third mix is electronic music from the 2010s. And the fourth and final mix is a New Wave playlist, which is largely music from the 1980s.

All four of these mixes are generated based on your music history from the recent past. Spotify's recommendation engine is truly amazing. It can broaden your musical palate, and introduce you to so much really great music from around the world. Though, that diversity is not necessarily reflected in these four specific mixes.

Recommender systems are among the most important applications of AI ML. Even tiny improvements to a recommender system used by, say, Amazon or Taobao, can generate massive amounts of revenue. For many companies, recommender systems are their lifeblood. And it is not unlikely that if you go into the field of AI or ML, that you will find yourself working on a recommender system at some point. You will try to unmask these systems and work towards understanding what makes them tick.

Conceptual Framing

Recommender systems are big, messy, and often more of an art than a science. Next, you will focus primarily on one of the math problems that underlies the messiness of real recommender systems and then you will explore real ones.

You can start by observing one rather popular approach, which is to try to recommend items that the user would rate highly. For example, the e-commerce platform, Amazon, might try to recommend products to you that you would rate a five out of five. Or Netflix, which currently uses a thumbs-up, thumbs-down system, might try to recommend shows to you that you would thumbs-up.

When recommending new items that you have never engaged with—for example, a show you have never watched, or a product you have never purchased—the platform does not know your ratings. Thus, it must try to guess—based on everything it knows about you—what you would rate. To get a sense of what this problem looks like, consider this table of made-up data.

Album Ratings

User	Ommazh	Melt-Banana	BTS	Zhou Shen	Sanam
An	5	5	0	1	1
Bhavana	4	5	?	1	1
Cordelia	?	2	4	5	5
Diego	2	?	5	?	4

Four different users have rated five albums by five different artists. The artists are Ommazh, Melt-Banana, BTS, Zhou Shen, and Sanam. An gave Ommazh and Melt-Banana the maximum rating of five out of five, but gave low ratings to BTS, Zhou Shen, and Sanam. Bhavana, who seems to have similar music tastes, rated Ommazh and Melt-Banana highly as well, and rated Zhou Shen and Sanam low. Note that Bhavana has not rated BTS. However, based on the data you have from An, you might imagine that Bhavana might also dislike BTS. That is the essence of what you will be trying to do with recommender systems.

Cordelia and Diego look like they do not particularly care for Ommazh and Melt-Banana, but really like BTS, Zhou Shen, and Sanam. Of course, you do not know everything about Cordelia and Diego, but this narrative is consistent with what little data you do have available. You will discuss different ways to try and fill in these question marks, and ultimately, how to use them to make recommendations.

Content-Based Filtering

BTS, Zhou Shen, and Sanam have something important in common: They are all wildly successful and highly-produced musical acts. By contrast, Ommazh and Melt-Banana are independently produced lo-fi music. In other words, Ommazh and Melt-Banana would have a large value for what you might call the lo-fi indie feature. And BTS, Zhou Shen, and Sanam would have a large value for what you might call the slick pop feature.

'**Lo-fi**' refers to music where imperfections are a deliberate aesthetic choice. And '**indie**' refers to music that is made independently of major record labels. The English word '**slick**' represents music that is crafted to as near perfection as possible. There are many other possible features that an album could have, like speed, Latin influence, vocal, chillwave, and has pianos. In this running example, you will just use the lo-fi and slick pop features. Suppose a panel of experts sat down with each of these albums and assigned a rating from zero to one for these two features. This might yield the numbers shown:

Album Features

Feature	Ommazh	Melt-Banana	BTS	Zhou Shen	Sanam
Lo-fi indie	0.80	0.95	0.05	0.02	0.20
Slick pop	0.10	0.01	0.99	0.95	0.90

No album was given a score of zero or one in any given category. Also, these are not probabilities since they do not sum to one. They are also not mutually exclusive, though they appear to be opposed to each other in this case. In some cases, you may have two factors that are almost entirely independent. For example, if your first two features were female vocalist and in English, you would expect no necessary correlation. In the context of recommender systems, these features are often called **item factors**. In other words, they are factors that describe each of the items that you might recommend. So, given these features, one approach is to simply train a linear regression model for each of your users on those item factors.

For example, consider the user, An. You know An’s ratings for these five albums:

Album Ratings

User	Ommazh	Melt-Banana	BTS	Zhou Shen	Sanam
An	5	5	0	1	1

To fit a linear regression model, you could rearrange this data into a DataFrame as shown:

Album Features and Ratings

Artist	Lo-fi_indie	Slick_pop	An_rating
Ommazh	0.80	0.10	5
Melt-Banana	0.95	0.01	5
BTS	0.05	0.99	0
Zhou Shen	0.02	0.95	1
Sanam	0.20	0.90	1

You would then fit a linear model, which predicts An's ratings from the Lo-fi_indie and Slick_pop columns. Fitting this linear regression model, you get back an intercept of 7.7 and θ_1 and θ_2 values of -2.63 and -7.2, respectively. Here, θ_1 indicates how much An enjoys lo-fi indie. And θ_2 is how much An enjoys slick pop.

In the context of recommender systems, these parameters are called the **user factors**. The resulting predictions are shown:

Album Features, Ratings, and Predictions

Artist	Lo-fi_indie	Slick_pop	An_rating	Prediction

Ommazh	0.80	0.10	5	4.885215
Melt-Banana	0.95	0.01	5	5.138695
BTS	0.05	0.99	0	0.451998
Zhou Shen	0.02	0.95	1	0.818614
Sanam	0.20	0.90	1	0.705478

Each prediction is computed in a normal way. In other words, it is the dot product of the parameters of the model with the features of the observation plus the bias term. Or in recommender system terminology, the prediction is the dot product of An's user factors with the item's factors. For example, Ommazh's item factors are 0.80 and 0.10 respectively. The dot product of these with An's user factors of −2.63 and −7.2 yield −2.824. You then add An's intercept of 7.7, yielding 4.88. Since the real rating that An gave was 5, you can compute the squared error of this prediction. It would be 5 minus 4.88 squared. Note that the errors in this model are non-zero and that is normal. As with most linear regression problems, it is impossible to get 100% accuracy with the given features.

You can use this approach to predict a user's rating of an album they have not yet listened to. For example, consider the user, Bhavana:

Album Ratings

User	Ommazh	Melt-Banana	BTS	Zhou Shen	Sanam
Bhavana	4	5	?	1	1

You do not have Bhavana's rating for BTS. So, to fit a model you could, in principle, create a DataFrame like the one shown:

Album Features and Ratings

Artist	Lo-fi_indie	Slick_pop	Bhavana _rating
Ommazh	0.80	0.10	4
Melt-Banana	0.95	0.01	5
Zhou Shen	0.02	0.95	1
Sanam	0.20	0.90	1

This time you only have four rows, because Bhavana has only rated four artists. Specifically, Bhavana's rating for BTS is not in the DataFrame, because you do not know it. Fitting a linear regression model like before, this gives you an intercept for Bhavana of 4.05, and user factors of θ_1 and θ_2 equal to 0.74 and −3.4, respectively. These parameters yield the predictions shown:

Album Features, Ratings, and Predictions

Artist	Lo-fi_indie	Slick_pop	Bhavana_rating	Prediction
Ommazh	0.80	0.10	4	4.302190
Melt-Banana	0.95	0.01	5	4.719237
Zhou Shen	0.02	0.95	1	0.837453
Sanam	0.20	0.90	1	1.141120

Now that you have a model for Bhavana, you can make a prediction for the BTS album that Bhavana has never heard or rated. You know that BTS has a Lo-fi_indie value of 0.05 and a Slick_pop value of 0.99. You use these values to guess Bhavana's rating for BTS. Specifically, the prediction for Bhavana's rating of BTS is the intercept 4.05 plus the dot product of the user's factors and the item factors. So that is 0.74 times 0.05, minus 3.4 times 0.99. And that gives you a prediction of 0.724.

Album Features, Rating, and Prediction

Artist	Lo-fi_indie	Slick_pop	Bhavana_rating	Prediction
BTS	0.05	0.99	NaN	0.724073

Repeating this process for Cordelia and Diego, you get the guessed prediction values shown:

Album Ratings

User	Ommazh	Melt-Banana	BTS	Zhou Shen	Sanam
An	5	5	0	1	1
Bhavana	4	5	**0.724**	1	1
Cordelia	**2.23**	2	4	5	5
Diego	2	**1.00**	5	**5.46**	4

In this dataset, there are only four missing values. But in a real dataset, you will have a much larger number of unknown ratings than known ones.

Think about how you might use this information in a recommender system. Suppose you want to offer Diego a new album to listen to. Since Zhou Shen has a higher predicted rating of 5.46 versus a rating of 1 for Melt-Banana, your system would do better to recommend Zhou Shen.

Note that the predicted ratings are sometimes outside of the scale. For example, Zhou Shen is predicted to have a rating of 5.46, which is bigger than the max of 5. If you do not like that in your model, you can use the **min** or **max** functions to constrain values from going outside the range. For example, you might replace the 5.46 with a 5.

The approach described is known as **content-based filtering**. The word '**filtering**' is used because you are using the predicted ratings as a way to filter out choices you do not want. For example, when Spotify creates a mix for you, there are millions of songs to choose from. So, it uses some sort of filtering algorithm to remove all but a few dozen. The words '**content-based**' are used because the filtering is based on the content of the items. In other words, the item factors tell whether or not the content is slick pop or lo-fi indie.

Note that, in this example, the two categories are more or less mutually exclusive but this is not usually the case. For example, if you had in English as a feature, then Melt-Banana would have a score of maybe 0.95 for this feature, since their music is mostly in English.

Content-based filtering works fine, except for one big problem. In real-world scenarios, you do not typically have items categorized into nice content categories. Somehow, if you used this approach, you would have to decide in advance what all the categories should be, and then have some way of determining the values in each category.

Collaborative Filtering

Content-based filtering is simple and works beautifully, but it requires you to somehow collect item factors for every item in your dataset. Consider the opposite situation. Suppose you start with the same incomplete set of ratings:

Album Ratings

User	Ommazh	Melt-Banana	BTS	Zhou Shen	Sanam
An	5	5	0	1	1
Bhavana	4	5	?	1	1
Cordelia	?	2	4	5	5
Diego	2	?	5	?	4

However, you have the complete set of user factors and some totally unknown set of item factors. In other words, instead of hiring experts to review each album and assign them item factors, you survey your users and ask them: How much do you like lo-fi indie, or slick pop? Users do this in advance when creating their account, so you know the user factors. Suppose such a survey yields the values shown:

User Factors

User	Lo-fi_indie	Slick_pop
An	5	1
Bhavana	4	0

Cordelia	2	5
Diego	2	4

For example, An says: "I love indie music," giving it a 5 out of 5 rating, but only gives slick pop a rating of 1 out of 5. If you think about this a little bit, it turns out you can do what you did before. You build a linear regression model to predict ratings from your user factors. In this case, the predictions of your model will be the ratings as before. But now the parameters will be the inferred item factors.

For example, start with the Ommazh album. You know An's, Bhavana's, and Diego's ratings, but you do not know Cordelia's. Using everyone's ratings of lo-fi indie, and slick pop, you can try to predict Cordelia's ratings. One way to do that, is to form the DataFrame shown:

User Factors and Ratings			
User	**Lo-fi_indie**	**Slick_pop**	**Ommazh_rating**
An	5	1	5.0
Bhavana	4	0	4.0
Cordelia	2	5	NaN
Diego	2	4	2.0

An has a lo-fi indie preference of 5 and a slick pop preference of 1, and rated Ommazh as a 5. But Bhavana has a lo-fi indie preference of 4, and a slick pop preference of 0, and rated Ommazh as a 4. And lastly, Diego has a lo-fi indie preference of 2 out of 5, and a slick pop preference of 4 out of 5, and rated Ommazh as a 2. So, if you fit a linear regression model on these three observations, you get back an Ommazh prediction of 2.0 for Cordelia.

User Factors, Ratings, and Predictions

User	Lo-fi_indie	Slick_pop	Ommazh_rating	Prediction
An	5	1	5.0	5.0
Bhavana	4	0	4.0	4.0
Cordelia	2	5	NaN	**2.0**
Diego	2	4	2.0	2.0

In this approach, the parameters of your model are the item factors you have inferred. For example, if you request the intercept and the parameters for the Ommazh model that you just fit, you get an intercept of 0, a θ_1 of 1, and a θ_2 of 0. That is, your model considers Ommazh to have a lo-fi indie value of 1 and a slick pop value of 0. So, you have just done the same thing as before, but now in reverse.

As another example, consider BTS. This time, you have fit a model to the DataFrame shown:

User Factors and Ratings

User	Lo-fi_indie	Slick_pop	BTS_rating
An	5	1	0.0
Bhavana	4	0	NaN
Cordelia	2	5	4.0

Diego	2	4	5.0

Here you are using An's, Cordelia's, and Diego's known ratings of 0, 4, and 5 respectively. This linear regression model for BTS yields these predictions:

User Factors, Ratings, and Predictions

User	Lo-fi_indie	Slick_pop	BTS_rating	Prediction
An	5	1	0.0	0.100350
Bhavana	4	0	NaN	−0.765461
Cordelia	2	5	4.0	4.0903151
Diego	2	4	5.0	3.845974

This time the predictions have non-zero loss. In other words, this model predicts An would give a rating of 0.1 to BTS, but An actually gave zero. So, there is a little bit of error there. Note that if you do not like that your model predicts Bhavana would give a negative score to BTS, you can simply replace that negative predicted score with a value of 0.

Interpreting this model, you see that the intercept is 0 and the item factors, θ_1 and θ_2, are −0.19 and 1.06 respectively. You can think of these values as the degree to which BTS belongs to the two categories. The model has decided that BTS is so not lo-fi indie, that the value is actually negative. And θ_2 tells you that BTS is very slick pop.

This approach might also reasonably be called content filtering. The only difference is that, rather than having to rate all of your items in advance under some common scale to create a set of known item factors, you have instead had all your users rate their preferences on some common scale, to create a set of known user factors. In each case, you used linear regression to infer the factors that you do not yet know.

Just like the first approach, generating some set of known factors is challenging. That is because users, when they sign up for a platform, do not really want to sit down and rank all of their preferences. Imagine signing up for a music streaming service and having to give ratings in dozens or hundreds of categories.

This brings you to the next approach, **collaborative filtering**. In this approach, you will start with only the ratings. That is, you do not have a known set of user factors or item factors. Naturally, without these you cannot do any sort of linear regression. Or can you? This is where you are going to get very clever, and it is going to seem strange. You are going to start by randomly making up some item factors. These are random numbers that have absolutely no connection with reality. There could be one item factor, two item factors, 50 item factors, whatever. These factors have no specific meaning at all.

For consistency with the earlier example, you will have two item factors.

Random Item Factors

User	Ommazh	Melt-Banana	BTS	Zhou Shen	Sanam
F1	0.172	0.9	−0.1	1.2	0.53
F2	0.96	0.91	0.6	0.71	0.21

Do not think of these as lo-fi indie or slick pop. They are arbitrary features. And the values that are generated are random values drawn from some distribution. You are going to fit a linear regression model for each user.

User Factors

User	F1	F2
An	−0.165	3.98
Bhavana	−1.72	5.20
Cordelia	1.56	3.53
Diego	3.59	3.76

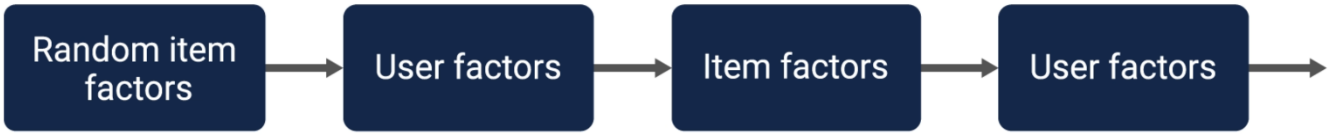
For example, you will try to predict An's ratings from the set of made-up random item factors. And, in this case, you get θ_1 is −0.165 and θ_2 is 3.98. So these two parameters are your inferred factors about An. Note that for simplicity, **fit_intercept** is set to False. So every user's intercept is 0. These factors mean that An does not like music with factor 1, but does like music with factor 2.

Naturally, if you compute the error for this model, it will have poor performance, since the supposedly known item factors were made up. As the old saying goes: Garbage in, garbage out. The clever part is what you do next. Using the user factors inferred from the randomly-generated item factors, you now fit linear regression models for each of the items. That yields a new set of item factors that are based on the user factors, that were based on the randomly-generated item factors. If you do this, you get new item factors that replace your random ones. For example, the new item factors for Ommazh are −0.245 and 0.862.

New Item Factors

User	Ommazh	Melt-Banana	BTS	Zhou Shen	Sanam
F1	-0.245	...	...	...	...
F2	0.862	...	...	...	...

You do that for Melt-Banana, BTS, Zhou Shen, and Sanam to get a new set of item factors for each album. Then you repeat this process using the new item factors to create a new set of user factors. And then the new set of user factors to get the new item factors. And so forth.


Random item factors → User factors → Item factors → User factors →

As you repeat this process over and over again, you will see the squared error between your predicted ratings and the true ratings continue to drop. The resulting model is often called a **collaborative filtering model**. The idea being that the users collaboratively gave you the data needed to understand the users and the items.

This may seem impossible. How can you start from a set of totally random item factors and get good results? The key is that during each iteration, you are using the original set of incomplete ratings data to generate the next set of factors. And thus, at each iteration of the process, the model is learning from the data. And that is no different from the usual machine learning models, which start with some arbitrary starting parameters, and then use gradient descent to minimize loss.

What is cool about this process is that you do not even have to specify the meaning of the item factors in advance. Though you do have to pick the number of factors you want to learn. This example used two, but you could pick any number. In general, the factors you get back from this algorithm are hard to interpret. In other words, once the process is done, you may not be able to learn anything at all about your data by looking at the values of the factors, because you do not know what they mean.

Gradient Descent View of Collaborative Filtering

The collaborative filtering algorithm works but it is inefficient. Every time you generate the next set of factors, you are running an entire pass of gradient descent. So, rather than performing dozens, hundreds, or thousands of separate gradient descent steps, you can instead do a single gradient descent that operates on user factors and item factors all at once.

To do this, you need to write out a function to be minimized. And to do that, you need some notation. Start by defining the matrices P and Q. P represents the matrix of user factors, and has a size $M \times Z$ — where M is the number of users and Z is the number of factors.

$$P: M \times\ Z$$

User Factors		
User	**F1**	**F2**
An	−0.52	1.35
Bhavana	−1.51	1.11
Cordelia	1.15	2.16

Diego	1.19	2.15

The matrix Q represents the matrix of item factors. It has a size $Z \times N$ — where Z is the number of factors and N is the number of items.

$$Q{:}\, Z \times N$$

Item Factors					
User	**Ommazh**	**Melt-Banana**	**BTS**	**Zhou Shen**	**Sanam**
F1	−0.153	−1.77	−1.16	1.09	0.73
F2	1.77	1.88	1.45	1.74	1.69

For this specific example, M is equal to the four users, N is equal to the five albums, and Z is chosen arbitrarily to equal the two factors. You could have chosen any Z you wanted, such as 100 or 1,000.

Recall that predictions for this model—for any given user, for a given item—is just the dot product of that user's factors with that item's factors. You can express this with the equation shown:

$$\hat{\mathrm{r}}_{\mathrm{i},j} = \mathrm{row}_{\mathrm{i}}(\mathrm{P}) \cdot \mathrm{col}_{\mathrm{j}}(\mathrm{Q})$$

You can give the squared error of a prediction $e^2_{i,j}$, as the difference between $\left(\hat{r}_{i,j} - r_{i,j}\right)^2$. Where $\hat{r}_{i,j}$ is the true rating by user i for item j.

$$e^2_{i,j} = \left(\hat{r}_{i,j} - r_{i,j}\right)^2$$

To test your understanding, try computing $e_{i,j}^2$, for $i = 3, j = 2$. In other words, the error for this model's prediction of Cordelia's rating of Melt-Banana. Note that Cordelia's true rating is 2.

The first thing you need to do is compute the dot product of the third row of P, with the second row of Q. That dot product is 1.15 times −1.77 plus 2.16 times 1.88, or 2.02. Since the true value is 2, the squared error is 2.02 minus 2, all squared, which yields 0.0004.

If you wanted the mean squared error for the model on the whole dataset, you would simply repeat this process for all combinations of users and items:

$$MSE = \sum_{i=1}^{M} \sum_{j=1}^{N} e_{i,\,j}$$

However, there is one important caveat: Consider ratings which do not exist in the dataset. For example, Diego has not rated Melt-Banana. It would seem that $e_{i,j}$ is undefined for $i = 4, j = 2$. One approach would be to simply assign a true ratings value of $\hat{r}_{4,2}$ to 0, but this is a terrible idea. This will tell the algorithm a way to learn parameters such that all unseen items have a reading of 0, which is not what you want. You do not want to assume your users have a rating of 0 for unseen items.

To avoid this, you modify the double summation slightly. Instead of summing over all j, you only sum over j in the set R_i:

$$MSE = \sum_{i=1}^{M} \sum_{j \in R_i}^{N} e_{i,\,j}$$

The set R_i is all the items which user i has rated. For example, R_4 is the set 1, 3, and 5 — because Diego has rated artists 1, 3, and 5, but not 2 and 4. Note that the N has been left on top of the sum symbol over j. That is meant as a mnemonic reminder that the j-values can go between 1 and N. In other words, you are iterating over the columns of Q. This is a somewhat non-standard use of the summing notation. A more formal mathematical treatment would omit this N.

Expanding this out, the mean squared error is the sum over i equal the 1 to M, of the sum over j in the set R_i, of the square of the ith row of P, dot the jth column of Q, minus $r_{i,j}$.

$$MSE = \sum_{i=1}^{M} \sum_{j \in R_i}^{N} \left(\mathrm{row_i}(P) \cdot \mathrm{col_j}(Q) - r_{i,j}\right)^2$$

The mean squared error is the sum of all the dot products, minus the corresponding true rating value.

How many terms will there be in the mean squared error expression for this example? In other words, how many different dot products will you need to compute to get the mean squared error? In total, there will be 16. While there are 20 total possible dot products, only 16 get computed. The other four correspond to ratings that do not exist in the original dataset. Thus, they will not be included in the mean squared error. The first of these squared errors is 0.0004, which you computed in an earlier example, and that will be followed by 15 more.

This brings you to your goal, finally. You want to pick values for P and Q, such that the mean squared error is minimized. Here you need to introduce a bit more notation. So let $P_{i,j}$ be the value of the ith row, and the jth column of P. And likewise with Q. You thus have a total of eight different P parameters: $P_{1,\,1}$, $P_{1,\,2}$, $P_{2,\,1}$, $P_{2,\,2}$, $P_{3,\,1}$, $P_{3,\,2}$, $P_{4,1}$, and $P_{4,2}$. You also have a total of ten Q parameters: $Q_{1,\,1}$, $Q_{1,\,2}$, $Q_{1,\,3}$, $Q_{1,\,4}$, $Q_{1,\,5}$, $Q_{2,\,1}$, $Q_{2,2}$, $Q_{2,3}$, $Q_{2,4}$, and $Q_{2,5}$. In all then, your gradient descent is going to optimize the mean squared error over an 18-dimensional space, corresponding to these 18 different matrix values.

From these equations, you can derive the gradient descent update rule for your algorithm:

$$P_{a,b} \coloneqq P_{a,b} - \alpha \sum_{j \in R_a}^{N} e_{a,j} Q_{b,j}$$

Specifically, you can show that at each gradient descent step, you want to update user factor $P_{a,b}$ so that it is equal to the old $P_{a,b}$, minus the learning rate α, times the sum of all the errors for user a; that is $e_{a,j}$. But each of those errors should be weighted by the corresponding item factor, $Q_{b,j}$.

The update rule relies on the fact that the partial derivative of the squared $e_{i,j}$ is equal to 2 times $e_{i,j}$, times $Q_{i,j}$.

$$\frac{\partial e_{i,j}^{2}}{\partial_{a,b}} = 2 e_{a,j} Q_{b,j}$$

Note that if you want to regularize your model, you can also include a regularization term. And in that case, the update rules will be slightly different. At any rate, this is one way you can use gradient descent to update P and Q.

This equation is hard to understand, so look at a small example to help reinforce what the update rule actually means. Pick a specific P that you want to update, say, $P_{4,1}$. What the update rule tells you is that Diego's first factor—that is: $P_{4,1}$—should be updated as follows:

$$P_{4,\,1} \coloneqq 1.19 - \alpha\left(e_{4,1}Q_{1,1} + e_{4,3}Q_{1,3} + e_{4,5}Q_{1,\,5}\right)$$

You take the old value of 1.19 and subtract α times the error for the prediction for An's reading of Ommazh, scaled by the first factor of Ommazh. Plus the error of the prediction of An's rating of BTS, times the first factor of BTS. Plus the error of An's rating for Sanam, times the first factor of Sanam.

You have the Q-values readily available. They are the first item factor for each of the three albums. So you can expand those out easily, as shown in this equation:

$$1.19 - \alpha(-1.53e_{4,1} + 1.16e_{4,3} + 0.73e_{4,5})$$

That leaves you with $e_{4,1}$, $e_{4,3}$, and $e_{4,5}$, which you need to compute. To test your understanding, try to find the value of this expression, if α is equal to 0.1. In order to do that, you need to know the true ratings Diego gave to Ommazh, BTS, and Sanam. Those are 2, 5, and 4 respectively.

To compute $e_{4,1}$, you compute the dot product of Diego's factors with Ommazh's factors, and then subtract off the true rating. This gives you a value of approximately 1.98. That is the prediction, minus 2, which is the true value, giving you −0.02.

To compute $e_{4,3}$, you compute the dot product of Diego's factors with BTS's factors, and subtract off the true rating. This gives you a value of approximately 4.5 minus 5, or −0.5.

To compute $e_{4,5}$, you compute the dot product of Diego's factors with Sanam's factors, and subtract off the true rating. In this case, you get a value of approximately 4.5 minus 5, or −0.5.

$$1.19 - 0.1(-1.53 \times -0.02 + 1.16 \times -0.5 + 0.73 \times -0.5)$$

Multiplying out all of these weighted errors, you get the new value of $P_{4,1}$ which is 1.19 plus 0.0914, or 1.28. To complete an entire gradient descent pass, you do the exact same thing for the other 17 parameters.

After finishing a gradient descent pass, you repeat this process again and again, generating new P's and Q's until the error is low enough that you are happy.

The ultimate result of this whole procedure is that you will convert the original matrix of ratings, R, and the two matrices P and Q — where P represents the inferred user factors and Q represents the inferred item factors. This algorithm is called **Funk SVD**.

Alternate View: Matrix Factorization

To check your understanding, do a quick review exercise. Consider the item and user factors computed earlier:

$$P: M \times Z$$

User Factors

User	F1	F2
An	−0.52	1.35
Bhavana	−1.51	1.11
Cordelia	1.15	2.16
Diego	1.19	2.15

$$Q : Z \times N$$

Item Factors

User	Ommazh	Melt-Banana	BTS	Zhou Shen	Sanam
F1	−0.153	−1.77	−1.16	1.09	0.73
F2	1.77	1.88	1.45	1.74	1.69

Suppose you want to compute An's rating for Ommazh, what would it be? Based on these values, it is the dot product of the Ommazh column with the An row. That gives you −1.53 times −1.52 plus 1.77, times 1.35, giving you a final value of 4.72 for the predicted rating.

Generalizing from this review exercise, observe an important relationship between the hidden ratings, user factors, and item factors. Specifically, if you represent the predicted ratings by the matrix $R\sim$, the user factors by the matrix P, and the item factors by the matrix Q, then $R\sim$ is just the matrix product of P and Q.

Note that factoring a matrix into two matrices does not yield a unique solution. In fact, there is not even a unique size for the two matrices. For example, in the review exercise, P was 4×2, and Q was 2×5. Suppose you had picked 50 factors — that is Z equal to 50. In that case, they would have been 4×50 and 50×5, respectively. So there are many different possible factorizations. The specific factorization you derived came from performing a gradient descent, which found values for a 4×2, P, and 2×5, Q, such that the mean squared error was minimized.

This algorithm is often called **Funk SVD**, or even just **SVD** for short. However, it is not the same thing as singular value decomposition, despite the fact that it has the same name. You can tell that it is not SVD for a couple of reasons. First of all, recall that SVD decomposes an $M \times N$ matrix into three matrices of size $M \times M$, $M \times N$, and $N \times N$, respectively. Whereas Funk SVD allows you to arbitrarily decompose into two matrices of size $M \times Z$ and $Z \times N$ — where Z is any size of your choosing. Secondly, the matrices that the real singular value decomposition gives have a property called **orthogonality**, but Funk SVD matrices do not have that property.

There are other matrix factorization techniques out there which you can use, such as **SVD++**. This does raise the interesting question: Why not just use the real SVD for matrix decomposition instead of a new technique that involves gradient descent, and all these complicated equations? The reason is simple. The real singular value decomposition does not have any way to deal with missing entries.

If you did happen to have a matrix of ratings with no missing entries, using singular value decomposition would indeed be a reasonable way to get user and item factors. But for real-life problems, where you have a huge number of missing entries, you need some alternate matrix factorization technique in order to do your decomposition.

Mini-Lesson: Other Matrix Factorization Techniques

In this module, you have learned some matrix factorization techniques for recommender systems. These techniques discover latent features in users and items. With this method, cold start problems and data sparsity can be reduced. This lesson will introduce you to some other techniques for matrix factorization methods, such as:

- Singular value decomposition (SVD)
- Probabilistic matrix factorization (PMF)
- Non-negative matrix factorization (NMF)
- Bayesian probabilistic matrix factorization (BPMF)

Singular value decomposition (SVD)

The famous SVD algorithm, as popularized by Simon Funk during the Netflix Prize, is a technique for reducing dimensions and creating superior quality recommendations for users. The SVD is commonly used to produce low-rank approximations before computing neighborhoods in collaborative filtering. SVD can also be used in collaborative filtering to discover latent associations between users and items to predict the probability of users selecting certain things.

SVD has also been successful with the following variants:

- **Funk SVD**
    - Simon Funk's original algorithm factored the user-item rating matrix as the product of two lower dimensional matrices
- **SVD++**

    - The SVD++ algorithm is an optimized SVD algorithm designed to provide implicit feedback to improve prediction accuracy
  - **Regularized SVD**
    - RSVD is a fast probability-based algorithm that can compute the near-optimal low-rank singular value decomposition of vast amounts of data with high accuracy. The idea of RSVD is to represent the data as compressed to capture the essential information. From this compressed representation, a low-rank singular value decomposition can be derived.
  - **Iterative SVD**
    - ISVD is used to estimate the singular value decomposition of an incomplete given matrix. ISVD uses first-order optimization over orthogonal manifolds and automatically estimates the rank of SVD. The purpose here is to estimate the singular vectors by optimizing the suitable space, which is the space of the orthogonal matrix manifolds.

Probabilistic Matrix Factorization (PMF)

The PMF approach assumes that a small number of unobserved factors determine a user's attitudes and preferences. User preferences are modeled by linearly combining item factor vectors with user-specific coefficients in a linear factor model. This technique has proven successful on huge, sparse, and imbalanced datasets.

Non-negative Matrix Factorization (NMF)

In NMF, the principal components of a set of non-negative data vectors are automatically extracted. The principal components can be used to extract significant and sparse features from those vectors.

NMF can reduce prediction errors compared to other techniques, such as SVD. Furthermore, when used in collaborative filtering, the NMF technique always leads to interpretable and sparse decompositions of non-negative matrices.

Bayesian Probabilistic Matrix Factorization (BPMF)

BPMF is a model in which capacity is automatically controlled by integrating all model parameters and hyperparameters, thereby allowing it to avoid parameter tuning and providing predictive distribution. Thus, the concept of BPMF is extended to recommendations where top N queries are recommended to users. This allows for more efficient and accurate predictions.

The SURPRISE Library

Scikit-learn does not currently support the Funk SVD matrix decomposition algorithm, or many other common ratings prediction algorithms. Nor does it significantly support recommender systems. However, there is a library created by Nicolas Hug, called **SURPRISE**. SURPRISE is intended to let users design and analyze recommender systems. It has a lot in common conceptually with scikit-learn, but there are some significant API differences.

Before using SURPRISE, start by looking at a pandas DataFrame that you will use as your dataset.

Here, only the first so many rows are shown:

DataFrame

UserID	ItemID	Rating
An	Ommazh	5
Bhavana	Ommazh	4
Diego	Ommazh	2
An	Melt-Banana	5
Bhavana	Melt-Banana	5
Cordelia	Melt-Banana	2
An	BTS	0
Cordelia	BTS	4

Note that this DataFrame is simply the matrix for ratings from before. Only now it is given as a list of ratings, rather than a grid of values.

Album Ratings

User	Ommazh	Melt-Banana	BTS	Zhou Shen	Sanam
An	5	5	0	1	1
Bhavana	4	5	?	1	1
Cordelia	?	2	4	5	5
Diego	2	?	5	?	4

SURPRISE does not know how to work with pandas DataFrames natively. So before you can use this DataFrame in SURPRISE, you need to convert it into what SURPRISE calls a **DatasetAutoFolds** object, which is part of the SURPRISE library.

To do this, you start by creating a SURPRISE **Reader** object and specifying the **rating_scale**. In this case, between 0 and 5. You then use the Dataset module's **load_from_df** function as shown:

```
from surprise import Dataset
from surprise import Reader
reader = Reader(rating_scale=(0, 5))
sf = Dataset.load_from_df(df[['userID', 'itemID', 'rating']],
                          reader)
```

The resulting **sf** object is now almost ready for use. Before you can take this **sf** data and feed it to an algorithm like Funk SVD, however, you have to create a training set from the **DatasetAutoFolds** object **sf** that you just created. In a real-world context where you want to avoid overfitting, you would create a separate training and test set. But for the purposes of learning how to use the library, you use the entire dataset as the training set.

To do this, use the line of code shown to call the **build_full_trainset** function of your dataset **autofolds** object:

```
training_data = sf.build_full_trainset()
```

Now that you have your training set, you can train your algorithm. The syntax is somewhat similar to scikit-learn:

```
import surprise
model = surprise.SVD(n_factors = 2,
                     n_epochs = 10000,
                     biased = False)
model.fit(training_data)
```

First, you create a model—in this case a **surprise.SVD** object. And you set **n_factors** to 2—meaning that your Funk SVD algorithm is going to come up with two hidden factors, just like in the running example. You could pick any number. For example, you could have said **n_factors** equal 50.

You also specify that you want 10,000 training epochs. The reason for this is that the termination conditions for the SURPRISE implementation of Funk SVD are unclear in terms of how long it is going to run. But what you do know is that, on this dataset, the default value of 20 training epochs yields poor results if you only pick two factors.

Finally, you set the **biased** argument to False, which is similar to the **fit_intercept** equal to False parameter in scikit-learn. And by doing that, you now match the notation you saw in your derivation of Funk SVD.

The next step is to call **fit**. Note that unlike linear regression, or logistic regression, or other such algorithms in scikit-learn, you do not specify a separate x or y. Instead, you just give the **trainset** object as the only parameter to the **fit** function. That is because when SURPRISE stores dataset objects, it already has everything annotated in a form that SURPRISE's models know how to read. In other words, some of the data is already noted as being x or y or whatever else.

Once you have done this, the model has been fit. In this case, **surprise.SVD** implements that Funk SVD algorithm from before that does gradient descent to factorize a matrix. Recall that this is not the same thing as singular value decomposition, despite it being called SVD.

Similar to scikit-learn models, you can now use the **predict** method to make predictions. For example, if you call **model.predict("an", "ommazh")** you get back 4.83, which is fairly close to the true rating of 5. You can also compute predictions for hidden ratings that you do not know. For example, **model.predict("cordelia", "ommazh")**, that will give you back a value of 5. And so even though Cordelia has not rated Ommazh, the algorithm believes that she would rate Ommazh a 5.

Now, unlike scikit-learn, the **predict** method in SURPRISE can only generate a single prediction at a time. In scikit-learn, you saw you could give an entire dataset. So, if you wanted to know the overall error, you need to make predictions for the entire dataset. And what SURPRISE requires you to do, is to generate a test set. In this case, you want to evaluate the error on the full set of data. So what you do to create a test set, is simply call the **build_testset** method on your **trainset** object:

test_data = training_data.build_testset()

After you make that test set, you need to generate a list of predictions. You do that by calling **model.test** on the test set:

predictions_list = model.test(test_data)

So it is quite a bit different from scikit-learn. Note that this **model.test** function does not work on objects of type **trainset**, which is why you had to convert the training set into a test set.

The first few entries in the list of predictions that you get back from **model.test** are shown:

```
[Prediction(uid='an', iid='ommazh', r_ui=5.0 est=4.830198623624999,
details={'was_impossible': False}),
Prediction(uid='an', iid='melt banana', r_ui=5.0, est=4.998687111661422,
details={'was_impossible': False}),
Prediction(uid='an', iid='bts', r_ui=0.0, est=0, details={'was_impossible':
False}),
```

Note that each of these predictions includes both the true value and the predicted value. If you want to compute the mean squared error on these predictions, you then call the **surprise.accuracy.mse** function on that list of predictions.

Also note that, unlike scikit-learn, the mean squared error function does not require you to specify a separate prediction and expected argument. That is because each prediction contains both. One way to think about it is that the SURPRISE library's interface likes to package things up into annotated objects, rather than leaving them relatively naked like scikit-learn. And in this case, the resulting value you get back is 0.0235. That is the error.

If you were to try with more factors, you would get a lower mean squared error. But you would start to run the risk of overfitting. To avoid overfitting, you do the usual workflow, where you train on a training set. But then you use a validation set to decide which hyperparameter to choose. In this case, the hyperparameter would be Z, the number of factors.

SURPRISE also supports k-fold cross-validation and even its own **GridSearchCV**. You can read the SURPRISE documentation and explore the library to figure out how **GridSearchCV** works.

You can also access your model's parameters. To get your user factors you call **model.pu**:

```
array([[-1.60210028, -1.0831017  ],
       [-1.66159317, -0.72972927],
       [ 1.72500423, -1.92593702],
       [-0.0378224 , -2.69335633]])
```

And to get your item factors you call **model.qi**:

```
array([[-1.44352162, -2.32437267],
       [-2.64083923, -0.70888811],
       [ 1.10555829, -1.53744304],
       [ 0.71994672, -1.85701467],
       [ 0.35161794, -1.79698869]])
```

These are just the P and Q matrices from before. Though Q was actually the transpose of what SURPRISE calls **qi**. Since Funk SVD is based on gradient descent, which starts from a random starting point, you will get different values—possibly dramatically so—if you try running this code.

To interpret these tables, consider the top-left entry of **pu**, −1.6. This is how much the first user, An, enjoys music with the first factor. Which has no specific meaning. Next to it, the value shown is how much An enjoys music with the second factor. Both are negative, but the first factor is more negative. So, you might say that An prefers music with the second factor over music which contains the first factor.

Similarly, consider the top-left entry of **qi** of −1.44. This is how much the first artist, Ommazh, has the first factor. It is negative, implying that Ommazh is the opposite of whatever the first factor means. The second factor is even more negative, at a value of −2.32. By contrast, consider the row starting with 1.1 in **qi**. This represents BTS. It has a large positive value for this factor. So, you might interpret the first factor as the pop factor. Ommazh and Melt-Banana are negative, because they are the opposite of pop music. BTS, Zhou Shen, and Sanam are all pop music to varying degrees. It is difficult to know how to interpret the second factor. And that is common. Often you do not know what the factors mean.

To generate a prediction, you simply compute the dot product of the appropriate factors. For example, if you take the dot product of the first entry and model that **pu** with the first entry, and model that **qi**, you are computing the rating for An for Ommazh. In this case, you get −1.6 times −1.4, minus 1.08 times −2.32, yielding the value of around 4.8 that you got before. In other words, since An does not like pop music, or whatever the second factor is, and since Ommazh is the opposite of both of these factors, then An actually really likes Ommazh.

More generally, you can form all of your predictions by computing the matrix product of P with the transpose of Q:

```
model.pu @ model.qi.T
```

```
array([[ 4.83019839, 4.99868719, -0.10600807, 0.8579089 , 1.3829943 ],
       [ 4.09470842, 4.90529682, -0.71507092, 0.15885939, 0.72706926],
       [ 1.98651449, -3.19018497, 4.86811119, 4.81840444, 4.06742948],
       [ 6.31496129, 2.00917115, 4.09906708, 4.9743721 , 4.82663182]])
```

One last note: The SURPRISE library also supports many other techniques for collaborative filtering besides the Funk SVD algorithm. For example, it contains another matrix factorization approach called **SVD++**, which was inspired by the Funk SVD algorithm.

It also supports non-matrix factorization-based techniques. For example, one natural approach is to simply use a natural extension of the **k-nearest neighbors** technique, which SURPRISE supports in the predictions algorithm package.

Hybrid Recommender Systems

You have learned a lot about fancy mathematical techniques for predicting unknown ratings. That is the science part. And you have seen how you can use the SURPRISE library to generate these ratings in a user-friendly way. But the more important question is the art side of things. After you compute all these ratings, how do you decide what to recommend to your users?

One obvious choice is to simply create a list of the highest-rated items, and recommend those. But there are other possibilities. For example, the Spotify music streaming service creates many different lists of recommendations for its users, such as daily mixes. Each mix consists of a set of items that are similar to each other, and which Spotify thinks you would enjoy. In other words, Spotify is considering not just the rating that you might give to the items, but the similarity of the items themselves to each other.

You can also imagine Spotify, or some other service, trying to create playlists or suggestions that are explicitly dissimilar to your previously rated items—but which it anticipates you would rate highly. This would encourage you to explore and discover new things, which might make you appreciate the service even more. That raises the interesting question of: How do you rate the similarity of items? You will come back to that in a moment.

There are also other metrics you might use for deciding what to recommend. For example, you might boost the profile of items that advertisers have paid you to promote. Or you might give more weight to items you have recently added to your service. You might simply recommend items that users have already engaged with, such as music on Spotify or products recently viewed on Amazon. And you might just recommend things that are popular across the platform as a whole.

There are even more niche possibilities, like recommending products you think the user is likely to buy at the moment, based on recurring user behavior. For example, a user buys toothpaste about every two months, and it has been about two months since they bought toothpaste.

Systems that use a combination of such metrics are called **hybrid recommender systems**. In practice, every real-world recommender system is a hybrid system. That is, rather than focusing only on ratings, they also focus on one or more of these other metrics.

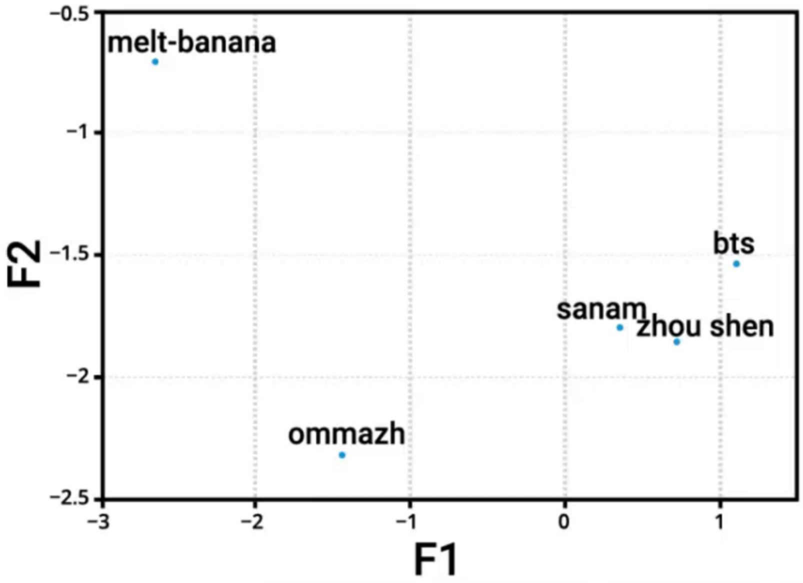
Now with regard to the similarity question, how do you know how similar two items are? Well, it turns out that the inferred item factors you compute are perfect for this. Each item is just a point in a Z-dimensional space. For example, consider the items factors matrix you had from before, which shows the two-factor values for five different albums.

Here, Z equals two.

model.qi

```
array([[−1.44352162, −2.32437267],
       [-2.64083923, −0.70888811],
       [ 1.10555829, −1.53744304],
       [ 0.71994672, −1.85701467],
       [ 0.35161794, −1.79698869]])
```

Also note, here you have a plot of all five albums in this two-dimensional space:



Item Factors					
User	**Ommazh**	**Melt-Banana**	**BTS**	**Zhou Shen**	**Sanam**
F1	−1.44	−2.64	1.11	0.72	0.35
F2	−2.32	−0.71	−1.54	−1.86	−1.80

To compute the similarity between any pair of albums, you simply compute the distance between two points. For example, the distance between

Ommazh in the top row and Melt-Banana in the row below, is the square root of −1.44 plus 2.64 squared, plus −2.32 plus 0.7 squared.

Now that you know how similarity works and how to predict unknown readings, you can use those two metrics and many others to generate a final recommendation for your users.