Module 19: Recommendation Systems

Video Transcripts

Video 1: Introduction

Netflix is a video streaming service available in nearly every country around the world. In fact, as of the time of this video, it's available in every country in the world — except China, North Korea, and Syria. Netflix has tens of thousands of movies and TV shows, which is more than any individual person would ever watch. So, when I log into Netflix, I'm presented with a screen similar to the one shown.

Here, Netflix gives me 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 something called 'Abby Hatcher.' As I have a young daughter, this category is undoubtedly recommended to me because of her viewing. Interestingly, I don't think my daughter's watched any of these specific shows. Now the next category that we see is dramas. And, again, I've never seen any of these shows, but Netflix would like me to watch them for some reason. And on the bottom row, we see Watch in One Weekend. Here we see two things that I have actually seen: 'The Great British Baking Show,' which is great, and 'The Queen's Gambit,' which I'd also heartily endorse. 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?

Well, the decision was made by a so-called recommender system, which knows about my previous viewing habits, including any ratings I explicitly gave the shows with the thumbs-up or thumbs-down buttons. Now consider what happens when I go to the online product marketplace, Amazon. Here we see a number of categories, each of which again contains a set of items, all in the hope of enticing me into making a purchase. So, as someone who recently purchased a pack of batteries and a water boiler, I see that the algorithm is keen that I should acquire more of these, as well as a coffee grinder. I have no idea why it's offering me bedding, chairs, or Japanese products, but perhaps these are popular on the platform at the moment. Now, I did recently buy knives and whiteboard markers, and a month ago purchased an inflatable unicorn sled. So that explains the 'Inspired by your purchases' showing knives and some sort of large inflatable dinosaur.

Now, again, a recommender system is behind all of this. Some model of me exists and it wants me to buy these goodies. As a third and final example, consider the music streaming service, Spotify. I listen to quite a lot of music, especially when preparing lectures. Indeed, at the very moment that I wrote these words, I was listening to Neu Chicago by Clive Tanaka y su Orquesta, which, as it happens, automatically came on when my playlist by the artist Brothertiger finished. Indeed, Spotify's recommender system that picked this song for me is so good that I thought Brothertiger was still playing. And I was a bit surprised when I went to look at Spotify to write these words, and I saw it was this other artist I'd never heard of. So when I open Spotify, among the many recommendations that it makes are daily mixes that exist for only one day. Here we see four different daily mixes, each a different genre. The first is '70s, '80s popular rock music artists. The second are a bunch of artists and men's choruses from the late 1950s, early

1960s. And that showed up for reasons I won't bore you with right now. The third mix is some excellent electronic music from the 2010s. And that whole playlist is anchored by the indomitable Paul Kalkbrenner. And then that fourth and final mix that appears is some kind of New Wave playlist, which is largely music from the 1980s. Now, all of these four mixes were generated based on my music history from the recent past. And I have to say, Spotify's recommendation engine has been a truly amazing and enjoyable part of my music listening life. It has broadened my musical palate, and it's introduced me to so much really great music from around the world. Though, that diversity is not necessarily reflected in these four specific mixes.

So now that we've seen a few of these, I have to say, recommender systems — they are among the most absolutely important applications of AI ML on the planet. Even just 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's not unlikely that if you go into the field of AI or ML, that you'll find yourself working on a recommender system at some point. Today, then, we will try to unmask these systems and work towards understanding what makes them tick. Let our journey begin.

Video 2: Conceptual Framing

Despite the incredible commercial importance of recommender systems, they do not lend themselves nearly so well to the sorts of nice and neat academic treatments that we're used to in our course. Nor do they receive the level of research attention at major universities that you might expect.

Recommender systems are big, and they're messy, and they're often more of an art than a science.

Today, I'll focus primarily on one of the nice math problems that underlie the messiness of real recommender systems. And then we'll talk about real ones at the very end. So, we'll start by observing that one rather popular approach is to try to serve items, or recommend items, that the user would rate highly. So, for example, the e-commerce platform, Amazon, might try to recommend products to me that I would rate a five out of five. Or Netflix, which — at least right now — uses a thumbs-up, thumbs-down system, might try to recommend shows to me that I would thumbs-up.

When recommending new items that I've never engaged with — for example a show I've never watched, or a product I've never purchased — the platform does not know my ratings, of course. Thus, it must try to guess — based on everything it knows about me — what I would rate. So, to get a sense of what this problem looks like, consider this table of made-up data that I'm showing here. Here we have four different users who have rated five albums by five different artists. The artists are Ommazh, which is a music group from Tatarstan, Melt-Banana from Japan, BTS from Korea, Zhou Shen from China, and Sanam from India. Sorry if my pronunciation was wrong. We see that 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 we have from An, we might imagine that Bhavana might also dislike BTS. That is the essence of what we'll be trying to do with recommender systems. We also see that we have two users, Cordelia and Diego, who look like they perhaps do not particularly care for Ommazh and Melt-Banana, but who really like BTS, Zhou Shen, and Sanam. Of course, we don't know everything about Cordelia and Diego, but the narrative I've given here is consistent with what little data we do have available. So, in the following videos we'll talk about different ways that we'll try to fill in these question marks, and ultimately talk about how we can use that to make recommendations.

Video 3: Content-Based Filtering

Knowing what I know about music, I'd say that 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, the first two items would have a large value for something we might call the lo-fi indie feature — that's Ommazh and Melt-Banana. And the last three items would have a large value for some feature we might call the slick pop feature.

Here, when I say 'lo-fi,' that's a term for music where imperfections are a deliberate aesthetic choice. And 'indie' refers to the fact that the music is made independently of major record labels. By contrast, I'm using the English word 'slick' to represent music that is crafted to as near perfection as possible. Of course, there are many other possible features that an album could have, like speed, Latin influence, vocal, chillwave, has pianos. And so, in our running example, we'll just use the two features that I personally came up with. Suppose we had a panel of experts who sat down with each of these albums and assigned a rating from zero to one for these two features. This might yield the number shown.

We see that no album was given a score of zero or one in any given category. We see also that these are not probabilities. In other words, they do not sum to one. They're also not mutually exclusive, though in this case, they appear to be opposed to each other. Now in some cases, you may have two factors that are almost entirely independent. For example, imagine my first two features were female vocalist and in English, you would expect no necessary correlation. Now 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 we might recommend. So, given these features, one approach is to simply train a linear regression model for each of our users on those item factors.

For example, consider the user, An. We know that An has rated these albums five, five, zero, one, and one respectively. To fit a linear regression model, we could rearrange this data into a DataFrame as shown. We would then fit a linear model, which predicts An's ratings from the Lo-fi_indie and the Slick_pop columns. Fitting this linear regression model, as usual, we get back an intercept of 7.7 and we get 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.

So in the context of recommender systems, these parameters are called the user factors. The resulting predictions are as shown. 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 a 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. We then add An's intercept of 7.7 yielding 4.88. Now, since the real rating that An actually gave was 5, we can compute the squared error of this prediction. It would be 5 minus 4.88 squared. Now note that the errors in this model are non-zero and that's normal. As with most linear regression problems, it's impossible to get 100% accuracy with the given features.

So, we can use this approach to predict a user's rating of an album that they have not yet listened to. For example, consider our user Bhavana. We don't have Bhavana's rating for BTS. So to fit a model we could, in principle, create a DataFrame like the one shown. Now this time we only have four rows, because Bhavana has not rated all five artists. She's only rated four of them. Specifically, observe that Bhavana's rating for BTS is not in the DataFrame, because we just don't know it. Fitting a linear regression model like before, this gives us 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. And, again, the predictions are not perfect.

So, now that we have a model for Bhavana, we can make a prediction for the BTS album that Bhavana has never heard or rated. We know that BTS has a Lo-fi_indie value of 0.05 and a Slick_pop value of 0.99. So, we'll use these values to guess at 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's 0.74 times 0.05, minus 3.4 times 0.99. And that gives us a prediction of 0.724. Now repeating this process for Cordelia and Diego, we'd get the guessed prediction value as shown. So, in our dataset, we only have these four missing values. But in a real dataset, you'll actually have a much larger number of unknown ratings than known ones. So, now let's think about how we might use this information in a recommender system. Suppose we want to offer a choice to Diego of 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, between those two choices, our system would do better to recommend Zhou Shen.

Now note that our predicted ratings are sometimes outside of the scale. For example, we saw that Zhou Shen is predicted to have a rating of 5.46, which is bigger than the max of 5. So if you don't like that idea in your model, you can just use the min or max functions to constrain them if you don't want them to go outside the range. For example, we might just replace the 5.46 with a 5. So, this approach I've described is known as content-based filtering. The word 'filtering' is used because we are using the predicted ratings as a way to filter out or move choices that we don't want. So for example, when Spotify creates a mix for me, there are many millions of songs to choose from. So, there's some sort of filtering algorithm that removes 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 us whether or not the content is slick pop or lo-fi indie. Note that in my example, the two categories were more or less mutually exclusive, but this is not usually the case. For example, if we had in English as a feature, then Melt-Banana, for example, 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, we don't typically have our items categorized into nice content categories. Somehow, if we used this approach, we'd have to decide in advance what all the categories should be, and then we'd have to have some way of determining the values in each category. So, in the next set of videos, we'll discuss an alternate approach.

Video 4: Collaborative Filtering

In the previous video, we saw content-based filtering. In this approach, we start with an incomplete set of ratings, a complete set of known item factors, and a totally unknown set of user factors. We then fit a linear regression model for each user. For each model, the parameters of the model are the inferred user factors for that user. And the outputs of the model are our predictions for the missing ratings for that user. This approach is simple and it works beautifully, but it requires us to somehow collect item factors for every item in our dataset.

So as a thought experiment, let's consider the opposite situation. Suppose we instead start with the same incomplete set of ratings, but we also have the complete set of user factors. And then out there is some totally unknown set of item factors. In other words, rather than the first scenario where we hire experts to review each of our albums and assign them item factors. In this new scenario, we've instead surveyed our users and asked them: How much, say, do you like lo-fi indie, or slick pop? And they do that in advance when they create their account. So, now we know the user factors. So, let's suppose such a survey yields the values shown. 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. Now if you think about this a little bit, turns out we can do what we did before.

Now we'll build a linear regression model to predict ratings from our user factors. In this case, the predictions of our model will be the ratings as before. But now the parameters will be the inferred item factors. It's the opposite of the situation from before. So, for example, let's start with our Ommazh album. We know An's, Bhavana's, and Diego's ratings, but we don't know Cordelia's. Now using everyone's ratings of lo-fi indie, and slick pop, we can try to predict Cordelia's ratings. So, one way to do that, is to form the given DataFrame that's shown. So, we see that 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 then lastly, we see that 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 we fit a linear regression model on these three observations, we get back an Ommazh prediction of 2.0 for Cordelia. Now, as mentioned earlier, in this approach the parameters of our model, they're the item factors which we've now inferred. For example, if we request the intercept in the parameters for the Ommazh model that we just fit, we get an intercept of 0, a θ_1 of 1, and a θ_2 of 0. That is, our model considers Ommazh to have a lo-fi indie value of 1 and a slick pop value of 0. So, we've just done the same thing as before, but now in reverse. So, as another example, consider BTS. This time, we've fit a model to the DataFrame shown. Here we're 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. And this time the predictions have non-zero loss. In other words, it predicts — this model — and this model predicts An would give a rating of 0.1 to BTS, but An actually gave zero. So, there's a little bit of error there. Note that if we don't like the fact that our model predicts that Bhavana would give a negative score to BTS. Like before, we can simply replace that predicted score that's negative with a value of 0. So, let's again interpret our model. If we look at our models, we see that our intercept is 0 and our item factors, θ_1 and θ_2, are −0.19 and 1.06 respectively. We can think of these values as the degree to which BTS belongs to our two categories. Here our model has decided that BTS is so not lo-fi indie, that the value is actually negative. And θ_2 tells us that BTS is very slick pop.

Now this approach might also reasonably be called content filtering. The only difference is that rather than having to rate all of our items in advance under some common scale to create a set of known item factors, we have instead had all of our users rate their preferences on some common scale, to create a set of known user factors. And in each case, we used linear regression to infer the factors that we don't yet know. Now, just like our first approach, generating some set of known factors is challenging. That's because users, when they sign up for a platform, don't 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. How much do you like instrumental music? How much do you like music with female singers? How much do you like music in the Tatar language, and so forth.

So, this brings us to our next approach, collaborative filtering. In this approach, we'll start with only the ratings. That is, we do not have a known set of user factors, and we do not have a set of known item factors. Now naturally, without these we can't do any sort of linear regression. Or can we?

Here's where we're going to get very clever, and it's going to seem strange. So, we're going to start by randomly making up some item factors. Now these have absolutely no connection with reality and they're just random numbers. There could be one item factor, two item factors, 50 item factors, whatever. These factors, they have no specific meaning at all.

Now here, I've chosen to have two item factors for consistency with the earlier example. But you should not think of these as lo-fi indie or slick pop. These are just some arbitrary feature. And the values that are generated here are just entirely random values drawn from some distribution. So then we're going to fit a linear regression model for each user. For example, we'll 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. And so these two parameters are our inferred factors about An. Now note that for simplicity, I've set fit intercept to false. In other words, every user's intercept is 0. So, what do these factors mean? They mean that An does not like music with factor 1, but does like music with factor 2.

Now naturally, if we computed the error for our model, this model will have poor performance, since the supposedly known item factors were just made up. As the old saying goes: Garbage in, garbage out. The clever part is what we do next. Using the user factors that we just inferred from the randomly-generated item factors, we'll now fit linear regression models for each of the items. That will yield a new set of item factors, that are based on the user factors, that were based on the randomly-generated item factors. So, for example, if we do this, we get new item factors that replace our random ones. For Ommazh, for example, that are −0.245 and 0.862. We then repeat that for Melt-Banana, for BTS, for Zhou Shen, and Sanam —

giving us a new set of item factors for each of our four albums. If you've gotten a bit confused, make sure to consult the notebook provided for this lecture to see exactly how I generated these numbers.

We then repeat this process using the new item factors to create a new set of user factors. And then a new set of user factors to get the new item factors. And then back to user, and so forth. As we repeat this process over, and over, and over, we'll see the squared error between our predicted ratings and the true ratings will drop, and drop, and drop. The resulting model is often called a collaborative filtering model. The idea being that the users collaboratively gave us the data we needed to understand the users and the items. Now this may seem impossible, like black magic. How can we start from a set of totally random item factors and get good results? Well, the key is that during each iteration, we're using that original set of incomplete ratings data to generate the next set of factors.

And thus, at each iteration of the process, our model is learning from the data. And that's in fact no different from our usual machine learning models, which start with some arbitrary starting parameters, and then they use gradient descent to minimize loss. Now what's cool about this process, is that we don't even have to specify the meaning of the item factors in advance. Though we do have to pick the number of factors that we want to learn. Recall that I picked two, but you could pick any number. Now in general, the factors you get back from this algorithm or 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 don't know what they mean. So, in the homework, you'll get a chance to implement this algorithm and see how the error evolves with each step of the algorithm, as well as how the number of features affects performance.

Video 5: Gradient Descent View of Collaborative Filtering

The collaborative filtering algorithm — as I described it in the previous video — works but it's inefficient. Every time we generate the next set of factors, we're running an entire pass of gradient descent. So, rather than performing dozens, or hundreds, or thousands of separate gradient descent steps, we can instead do a single gradient descent that operates on user factors and item factors all at once.

To do this, we'll need to write out a function to be minimized. And to do that, we're going to need some notation. Let's start by defining our matrices P and Q. P represents our 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. We'll also have our matrix Q, which represents our 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. For our specific example, we have M equal the four users, and N equal the five albums. We also arbitrarily chose Z equal the two factors. Though, as noted earlier, we could have chosen any Z we wanted — a hundred, a thousand, whatever.

Recall that predictions for our model, or for any given user, for a given item, is just the dot product of that user's factors with that item's factors. So, we can express this with the equation shown: $\hat{\mathrm{r}}_{\mathrm{i},j}$ is equal to $\mathrm{row}_{\mathrm{i}}(\mathrm{P}) \cdot \mathrm{col}_{\mathrm{j}}(\mathrm{Q})$. We 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 here, $\hat{r}_{i,j}$ is the true rating by user i for item j. So, 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 the true rating that Cordelia gave was 2. I recommend pausing the video and using a calculator of some kind to compute a value, or just watch and I'll spoil it. So, the first thing we 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.

Now since the true value is 2, the squared error is 2.02 minus 2, all squared, which yields 0.0004. Pretty low error. Now if we wanted the mean squared error for the model on the whole dataset, we would simply repeat this process for all combinations of users and items, but with 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 $i = 4$, $j = 2$. Now 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 don't want to assume your users have a rating of 0 for unseen items.

So, instead what I'll do is modify this double summation slightly. Specifically, instead of summing over all j, we'll only sum over j in the set R_i. The set R_i is all the items which user i has rated. So for example, R_4 is the set 1, 3, and 5 — because Diego has rated artists 1, 3, and 5, but not users 2 and 4. Note that I've left the N on the top of this sum symbol over j. That's just meant as a mnemonic reminder that the j-values can go between 1 and N. In other words, we're iterating over, iterating over the columns of Q. Now, I'll admit that's a somewhat non-standard use of the summing notation, and a more formal mathematical treatment would just omit this N on top. Expanding this out, we have that 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 $r_{i,j}$.

Take a moment to pause the video and really reflect on what this expression is saying. The mean squared error will be the sum of all of the dot products, minus the corresponding true rating value. To test your understanding, how many terms will there be in the mean squared error expression for the example from this lecture? In other words, how many different dot products will we need to compute to get the mean squared error? So, 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 don't exist in the original dataset. And thus they will not be included in the mean squared error. The first of these squared errors is 0.0004, which we computed in our earlier example, and that will be followed by 15 more. So this brings us to our goal, finally. We want to pick values for P and Q, such that the mean squared error is minimized.

Here we need to introduce just 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. We 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 $P_{4,2}$, as shown. We also have a total of ten Q parameters, which I will read off just to be very clear. We have $Q_{1,1}$, $Q_{1,2}$, $Q_{1,3}$, $Q_{1,4}$, $Q_{1,5}$. And then we also have the second row, $Q_{2,1}$, $Q_{2,2}$, $Q_{2,3}$, $Q_{2,4}$, and $Q_{2,5}$. So in all then, our gradient descent is going to be optimizing the mean squared error over an 18-dimensional space, corresponding to these 18 different matrix values.

From these equations, we can derive the gradient descent update rule for our algorithm.

Specifically, you can show that at each gradient descent step, we 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 — $e_{a,j}$. But each of those errors should be weighted by the corresponding item factor — $Q_{b,j}$. Now this big mysterious equation relies on the fact that the partial derivative of the squared $e_{i,j}$ is equal to 2 times $e_{i,j}$, times $Q_{i,j}$. This proof is somewhat involved and tedious. So, if you really want to understand it, I'm not going to cover it here, but you can work it out on your own at some point. And we can also derive a similar rule for Q, but I'm not going to show it here. I should note that if you wanted to regularize your model, you could also include a regularization term.

And in that case, the update rules will be slightly different. At any rate. This is one way that we can use gradient descent to update P and Q. Now this equation is hard to understand as I've presented it, I've just dropped a big equation on you. So to help reinforce what this update rule actually means, let's work at a small example and then it will make more sense. Let's pick a specific P that we want to update. So I'll say $P_{4,1}$, arbitrarily. What the update rule tells us is that Diego's first factor — that's $P_{4,1}$ — should be updated as follows. We take our old value of 1.19 and we 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. OK?

Now we have these Q-values readily available. They are the first item factor for each of our three albums. So, we can expand those out easily, as shown in this equation. So, that leaves us with $e_{4,1}$, $e_{4,3}$, and $e_{4,5}$, which we need to compute. Then to test your understanding, I would recommend you actually pause here, and try to find the value of this expression, if α is equal to 0.1. Now in order to do that, you need to know the true ratings that Diego gave to Ommazh, BTS, and Sanam. And those are 2, 5, and 4 respectively, as shown on the screen. So, now that hopefully you've taken a moment to try and work out the numbers yourself, I will tell you the answer.

To compute $e_{4,1}$, we simply compute the dot product of Diego's factors with Ommazh's factors. And then we subtract off the true rating. This gives us a value of approximately 1.98. That's our prediction, minus 2, which is the true value, giving us −0.02. To compute $e_{4,3}$, we simply compute the dot product of Diego's factors with BTS's factors. And again, subtract off the true rating. This gives us a value of approximately 4.5 minus 5, or −0.5. To compute $e_{4,5}$, we simply compute the dot product of Diego's factors with Sanam's factors, and again subtract off the true rating. And in this case, we get a value of approximately 4.5 minus 5, or −0.5.

So now multiplying out all of these weighted errors, we get that the new value of $P_{4,1}$ should be 1.19 plus 0.0914, or 1.28. And that's assuming it hasn't messed up the algebra there somewhere. Now to complete an entire gradient descent pass, we do this exact same thing for the other 17 of our parameters. Now note that I have not yet given you the expression for the update rule for Q. And I will also not give you the proof for the expression of P. And it's not that that proof is difficult, it's just long and not interesting to watch in a video.

So, after finishing a gradient descent pass, we would repeat this process again, and again, and again, generating new P's and Q's until our error is low enough that we're happy.

The ultimate result of this whole procedure is that we will convert our original matrix of ratings, R, and the two matrices P and Q — where P represents our inferred user factors and Q represents our inferred item factors. The name of this algorithm that I've described here is called Funk SVD for reasons I'll describe a bit more shortly. Note that in our example here, I used a choice of Z equal the two factors, so it'll all fit on the screen. But in general, you can make Z whatever you want. It could be, in general, hundreds or thousands. In the next video, we'll reflect a bit more carefully on what this whole procedure means from a linear algebra perspective.

Video 6: Alternate View: Matrix Factorization

Before we get started with this video, let's do a quick review exercise to check your understanding. Consider the item and user factors computed earlier. Suppose we wanted to compute An's rating for Ommazh, what would it be? I just want you to think about that, so make sure you really get that. So, based on these values — I hope you've paused to do this work — it's just going to be the dot product of the Ommazh column with the An row. That gives us −1.53 times −1.52 plus 1.77, times 1.35, giving us a final value of 4.72 for our predicted rating.

So generalizing from that review exercise we just did, I want to observe an important relationship between our hidden ratings, the user factors, and the item factors. Specifically, if we represent our predicted ratings between...by the matrix $R\sim$, the user factors by the matrix P, and the item factors by the matrix Q. We have the $R\sim$: Our predictions is just the matrix product of P and Q. Now, I should note that factoring a matrix into two matrices does not yield a unique solution. In fact, there's not even a unique size for the two matrices. For example, in the…what we saw before, we had that P was 4×2, and Q was 2×5. But let's say we'd picked 50 factors — that's Z equal to 50. And in that case, they would have been 4×50 and 50×5, respectively. So there's many different possible factorizations. And the specific factorization that we derived, it 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.

Now, as I mentioned in the previous video, this algorithm is often called Funk SVD, or even just SVD for short. However, it is not the same thing as singular value decomposition, which Gabriel talked about in an earlier module, despite the fact that it has the same name. So it's pretty confusing. Now how did that happen? Well, this technique — Funk SVD — it originally was used to great success in the Netflix challenge in 2006 by an author who used the name Simon Funk. At that time, Netflix offered a $1 million prize for the team that had the best performance on their ratings dataset. Now, they had the caveat that whoever participated in this challenge would only be eligible for the prize, if they exceeded Netflix's own algorithm by at least 10% in the error on the unseen ratings. Now, interestingly, it was in 2009, that's the third year of the contest, that someone finally passed that threshold. In fact, there were two teams that passed the 10% threshold and they exactly tied on their error. They both got a root mean squared error of 0.8567. However, one of those two teams submitted their solution 20 minutes earlier than the other, which earned them the $1 million prize. So a three-year contest, 20 minute tie break.

Now, Simon Funk was a pseudonym for a researcher named Brandon Webb. In December 2006, in the very early months of the contest, he wrote this blog post describing this algorithm, Funk SVD, that we've used today. Now in his description of his matrix factorization process, he claims that it is equivalent to singular value decomposition. And so he just called it SVD for short. But it is not singular value decomposition. We can tell that it's not SVD as Gabriel described 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 us to arbitrarily decompose into two matrices of size $M \times Z$ and $Z \times N$ — where Z is any size of our choosing. Just a totally different result. Secondly, the matrices that singular value decomposition gives us, the real singular value decomposition, they have a property called orthogonality, but Funk SVD matrices do not have that property.

Now, despite the fact that this gradient descent-based matrix factorization technique, is not actually singular value decomposition, it is still frequently called SVD, because it was misidentified as such by the author who popularized the technique. It's a strange historical artifact. Now there are other matrix factorization techniques out there which you can use, such as SVD++. We're not going to talk about those in the videos for this module, but you'll have a chance to try them out on the homework. Now this does raise the interesting question: Why not just use the real SVD for matrix decomposition instead of this new technique from a 2006 blog post that involves gradient descent, and all these equations, and just seems really complicated? Well, the reason is simple. The real singular value decomposition doesn't have any way to deal with missing entries.

Now, 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 our real-life problems, where we have a huge number of missing entries, we need some alternate matrix factorization technique in order to do our decomposition.

Video 7: The SURPRISE Library

At least at the time of this video, scikit-learn does not 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, no relation to me, that is called SURPRISE.

SURPRISE is intended to let users design and analyze recommender systems. Now it has a lot in common conceptually with sklearn. And in fact, Nicolas Hug works on sklearn these days. But there are some significant API differences. In this video, I'll walk through a simple example of the SURPRISE library, so that you're prepared for the homework. I won't go into a great amount of detail, because the documentation for SURPRISE is quite good. Now before we get to using SURPRISE, let's start by looking at a pandas DataFrame that we'll use as our dataset. Here I'm only showing the first so many rows. Note that this DataFrame is simply the matrix for ratings from before. Only now it's given as a list of ratings, rather than a grid of values.

SURPRISE doesn't know how to work with pandas DataFrames natively. So before we can use this DataFrame in SURPRISE, we need to convert it into what SURPRISE calls a DatasetAutoFolds object, which is just part of the SURPRISE library. To do this, we start by creating a SURPRISE Reader object and specifying the rating_scale. In this case between 0 and 5. We then use the Dataset module's load_from_df function as shown. The resulting sf object is now almost ready for use. Now before we can take this sf data and feed it to an algorithm like Funk SVD, we have to create a training set from the DatasetAutoFolds object sf that we just created. Now in a real-world context where you want to avoid overfitting, we'd create a separate training and test set. But since, in this video, I'm just trying to show you how to use the library, I'm going to use the entire dataset as our training set.

To do this, I'll use the line of code shown, where we call the build_full_trainset function of our dataset autofolds object. You'll learn how to do a proper train test split in SURPRISE on the homework. Now that we have our training set, we can train our algorithm. The syntax is somewhat similar to sklearn. First, we create a model — in this case a surprise.SVD object. And here I've set n_factors to 2 — meaning that our Funk SVD algorithm is going to come up with two hidden factors, just like in our running example. Now, of course, you could pick any number here. For example, you could have said n_factors equals 50. And on the homework you'll explore how Funk SVD models behave as a function of the number of factors. I've also specified that I want 10,000 training epochs.

The reason I've done that, is that the termination conditions for the SURPRISE implementation of Funk SVD are unclear to me in terms of how long it's going to run. But what I do know is that on this dataset, the default value of 20 training epochs yields poor results if we only pick two factors. Finally, I've set the biased argument to False, which is similar to the fit intercept equal to false parameter in sklearn. And by doing that, we now match the notation that we saw in our 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, we don't specify a separate x or y. Instead, we just give the trainset object as the only parameter to the fit function. And that's because the way that 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's already noted as being x or y or whatever else. Once we've 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, we can now use the predict method to make predictions. For example, if we call model.predict("an", "ommazh") we get back 4.83, which is fairly close to the true rating of 5. We can also compute predictions for hidden ratings that we don't know. For example, model.predict("cordelia", "ommazh"), that will give us 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 one thing is that unlike scikit-learn, the predict method in SURPRISE can only generate a single prediction at a time. In scikit-learn earlier we saw we could give an entire dataset. So, if we wanted to know the overall error, we need to make predictions for the entire dataset. And what SURPRISE requires us to do, is to generate a test set. Now in our case, we want to evaluate the error on the full set of data of our not-that-big table. And so what we'll do to create a test set, is simply call the build_testset method on our trainset object. That's just the way SURPRISE works.

So, our next step, after we make that test set, using the code you see on the screen, we need to generate a list of predictions. We do that by calling model.test on the test set. And so it's quite a bit different from scikit-learn. And I'll note that this function — model.test — does not work on objects of type trainset, which is why I had to do this seemingly arbitrary task of converting our training set into a test set. Now the first few entries in the list of predictions that we get back from model.test are shown. Note that each of these predictions includes both the true value and the predicted value. It's all packaged together. If we want to compute the mean squared error on these predictions, we would then call the surprise.accuracy.mse function on that list of predictions.

So note that unlike scikit-learn, the mean squared error function we have doesn't require us to specify a separate prediction and expected argument. And that's because each prediction contains both. Philosophically, the way I think about it, is that the SURPRISE library's interface, it likes to package things up into annotated objects, rather than leaving them relatively naked like sklearn. And in this case, what we get back is that the resulting value is 0.0235. That's the error. Now, if we were to try with more factors, we would get a lower mean squared error. But we would of course start to run the risk of overfitting. To avoid overfitting, we do the usual workflow, where we train on a training set. But then we use a validation set to decide which hyperparameter to choose. In this case, our hyperparameter would be Z, the number of factors. So you'll get a chance to explore this workflow on the homework. SURPRISE also supports k-fold cross-validation and even its own GridSearchCV.

Now, as one of our goals in this class is to give you practice with learning to read documentation and explore libraries on your own, I will not be explaining how GridSearchCV works. Instead, you'll be expected to read the SURPRISE documentation to figure out how all this works. Though, support will be provided, of course, if you get stuck on the homework. Naturally, we can also access our model's parameters. To get our user factors we call model.pu. And to get our item factors we call model.qi. These are just our P and Q matrices from before. Though our Q was actually the transpose of what SURPRISE calls qi. For example, when I ran the code, I got the model.pu and model.qi that are shown on the screen.

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

Similarly, continue...consider the top-left entry of qi of −1.44. This is how much our first artist, Ommazh, has the first factor. It's 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, I might interpret the first factor as the pop factor. Ommazh and Melt-Banana are negative, because they are the opposite of pop music. By contrast, BTS, Zhou Shen, and Sanam are all pop music to varying degrees. And I'll admit, I have no idea how I might interpret the second factor. And that's common. Often you don't know what the factors mean.

To generate a prediction, we simply compute the dot product of the appropriate factors. So, for example, if we take the dot product of the first entry and model that pu with the first entry, and model that qi, we're computing the rating for An for Ommazh. In this case, we get −1.6 times −1.4, minus 1.08 times −2.32, yielding the value of around 4.8 that we got before. In other words, since An doesn't 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. Now, more generally, we can form all of our predictions by computing the matrix product of P with the transpose of Q. One last note: The SURPRISE library also supports many other techniques for collaborative filtering other than the Funk SVD algorithm that I described in the lecture. For example, it contains another matrix factorization approach called SVD++, which was inspired by the Funk SVD algorithm.

However, 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. So, on the homework you'll have a chance to try out all of those different predictions algorithms if you'd like, and see how they compare.

Video 8: Hybrid Recommender Systems

We have talked a lot about fancy mathematical techniques for predicting unknown ratings. That's the science part. And we've seen how we can use the SURPRISE library to generate these ratings in a user-friendly way — at least once you learn how to use the SURPRISE library. But the obvious and more important question is the art side of things. After we compute all these ratings, how do we decide what to recommend to our users? So one obvious choice is to simply create a list of the highest-rated items, and recommend those. But there's other possibilities. For example, the Spotify music streaming service actually creates many different lists of recommendations for its users. As we saw earlier, it creates these daily mixes for me every day.

Now thinking about those mixes, each of them consists of a set of items that are similar to each other, and which Spotify thinks I would enjoy. In other words, Spotify is considering not just the rating that I might give to the items, but the similarity of the items themselves to each other. We can also imagine Spotify, or some other service, trying to create playlists or suggestions that are explicitly dissimilar to my previously rated items — but which it anticipates I would rate highly. This would encourage me as a user to explore and discover new things, which might make me appreciate the service even more. So, that raises the interesting question of: How do you rate the similarity of items? And we'll get back to that in a moment. Now, there are also other metrics you might use for deciding what to recommend.

An obvious example is we might boost the profile of items that advertisers have paid us to promote. Or we might give more weight to items that we've recently added to our service. We might simply recommend items that users have already engaged with. For example, music on Spotify or products recently viewed on Amazon. And we might just recommend things that are popular across the platform as a whole. There are even more niche possibilities, like recommending products that we 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's been about two months since they bought toothpaste. Now systems that use a combination of such metrics are called hybrid recommender systems. And in practice, basically 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 regards to the similarity question, how do we know how similar two items are? Well, it turns out that the inferred item factors that we compute are perfect for this. Each item is just a point in a Z-dimensional space. For example, consider the items factors matrix that we had from before, which shows the two-factor values for our five different albums. Here, Z equals two. Also note, here I have a plot of all five of our albums in this two-dimensional space. So to compute the similarity between any pair of albums, we simply compute the distance between two points. So 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. So now that we know how similarity works and how to predict unknown readings, we can use those two metrics and many others in order to generate a final recommendation for our users.

Video 9: Conclusion

Recommender systems, also known as recommendation systems, have one job. They recommend items to users. As input, they take knowledge about users, items, and interactions between users and items. It's very common in recommender systems to have access to ratings data. Ratings could be star ratings from zero to five stars, or they could be a simpler scale, such as like / dislike.

Recommender systems that use ratings, as we saw today, try to predict ratings for items that users have not yet rated. We discussed two approaches for predicting unknown user ratings. First, we saw content-based filtering, which relies on having a known set of item factors or user factors. Note that the definition of content-based filtering is a bit slippery, and other sources may use that term in a different sense than I have today.

Second, we saw collaborative filtering, where the algorithm infers both user and item factors from a set of ratings data. We can implement collaborative filtering using iterated passes of linear regression — alternately generating user factors, then item factors, then better user factors, then better item factors, and so forth. Or we could implement collaborative filtering using the Funk SVD algorithm, which isn't actually singular value decomposition, despite the name. We saw that the Funk SVD algorithm is a form of matrix factorization — taking an original matrix of ratings and decomposing it into the product of two matrices, one of user factors and another of item factors.

On a practical basis, we saw how Nicolas Hug's SURPRISE library can be used to build and evaluate ratings models. And you'll explore this library in more detail on the homework. Now, though we focused a lot on ratings, they are only one facet of the broader problem of recommender systems. While recommending highly-rated items to users is obviously a powerful tool, there's other metrics out there that are often used to decide what to recommend — such as similarity to other items, frequently viewed items, and more. Trying to decide which metrics to focus on when building a recommender system is an art, not a science. Unlike tasks such as classification, regression, or rating prediction, there's no mathematically defined loss function. Instead, each company or other entity that builds a recommender system, must employ some guesswork — engaging in user studies, evaluating downstream metrics like revenue satisfaction or engagement, and employing techniques like AB testing.

And all of those techniques are well outside of the scope of the videos that I put together for this course. Now if you ever end up working to build recommender systems, you'll find that at a company or other entity, you have to learn a totally unique toolchain, and some set of techniques, for evaluating the quality of your recommendation. So this module, I hope, will provide you with the foundation you'll need if you ever need to work on such a recommender system, or if you just simply wanted to know how recommender systems work. Thanks for watching, everybody, and I will see you next time.