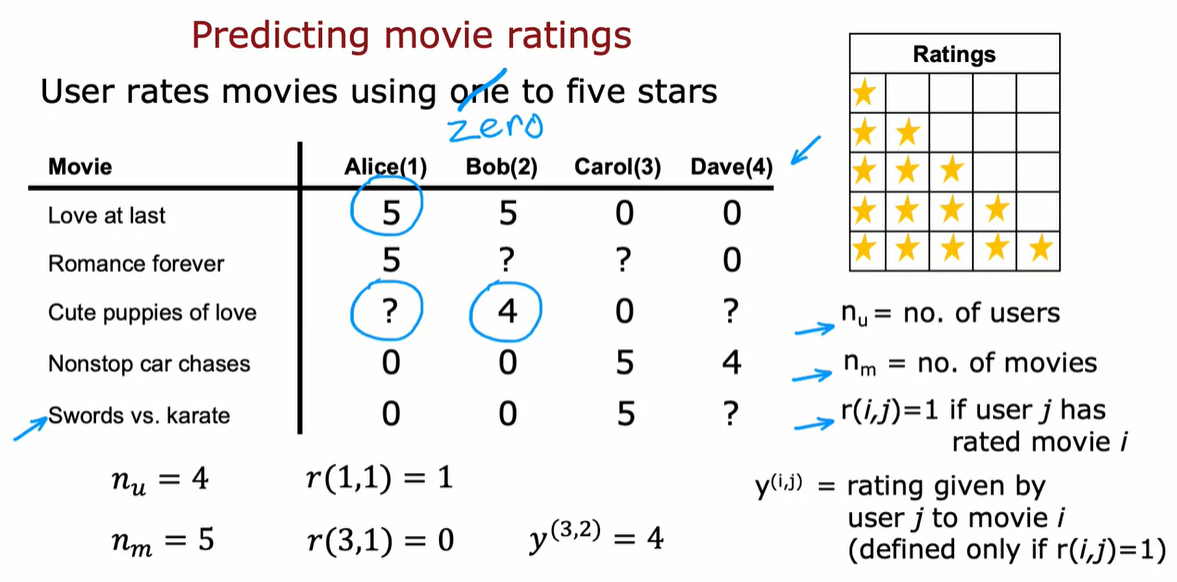
RECOMMENDATION SYSTEM

################################COLABORATIVE FILTERING ############################

# Video Making Recommendations



This is one of the topics that has received quite a bit of attention

in academia.

But the commercial impact and

the actual number of practical use cases of recommended systems seems to me to be

even vastly greater than the amount of attention it has received in academia.

Every time you go to an online shopping website like Amazon or

a movie streaming sites like Netflix or go to one of the apps or

sites that do food delivery.

Many of these sites will recommend things to you that they think you may want to buy

or movies they think you may want to watch or

restaurants that they think you may want to try out.

And for many companies,

a large fraction of sales is driven by their recommended systems.

So today for many companies, the economics or the value driven by recommended systems

is very large and so what we're doing this week is take a look at how they work.

So with that let's dive in and take a look at what is a recommended system.

I'm going to use as a running example, the application of predicting movie ratings.

So say you run a large movie streaming website and

your users have rated movies using one to five stars.

And so in a typical recommended system you have a set of users,

here we have four users Alice, Bob Carol and Dave.

Which have numbered users 1,2,3,4.

As well as a set of movies Love at last, Romance forever,

Cute puppies of love and then Nonstop car chases and Sword versus karate.

And what the users have done is rated these movies one to five stars.

Or in fact to make some of these examples a little bit easier.

I'm not going to let them rate the movies from zero to five stars.

So say Alice has rated Love and last five stars, Romance forever five stars.

Maybe she has not yet watched cute puppies of love so

you don't have a rating for that.

And I'm going to denote that via a question mark and

she thinks nonstop car chases and sword versus karate deserve zero stars bob.

Race at five stars has not watched that, so

you don't have a rating race at four stars, 0,0.

Carol on the other hand, things that deserve zero stars has not

watched that zero stars and she loves nonstop car chases and

swords versus karate and Dave raise the movies as follows.

In the typical recommended system,

you have some number of users as well as some number of items.

In this case the items are movies that you want to recommend to the users.

And even though I'm using movies in this example, the same logic or the same thing.

Work for recommending anything from products or website myself to restaurants,

to even which media articles, the social media articles to show,

to the user that may be more interesting for them.

The notation I'm going to use is I'm going to use nu to denote the number of users.

So in this example nu is equal to four because you have four users and

nm to denote the number of movies or really the number of items.

So in this example nm is equal to five because we have five movies.

I'm going to set r(i,j)=1,

if user j has rated movie i.

So for example, use a one Dallas Alice has rated movie one but

has not rated movie three and so r(1,1) =1,

because she has rated movie one, but

r( 3,1)=0 because she has not rated movie number three.

Then finally I'm going to use y(i,j).

J to denote the rating given by user j to movie i.

So for example,

this rating here would be that movie three was rated by user 2 to be equal to four.

Play video starting at :4:19 and follow transcript4:19

Notice that not every user rates every movie and it's important for

the system to know which users have rated which movies.

That's why we're going to define r(i,j)=1 if user j has rated movie i and

will be equal to zero if user j has not rated movie i.

So with this framework for recommended systems one possible way to approach

the problem is to look at the movies that users have not rated.

And to try to predict how users would rate those movies because then we can try

to recommend to users things that they are more likely to rate as five stars.

And in the next video we'll start to develop an algorithm for

doing exactly that.

But making one very special assumption.

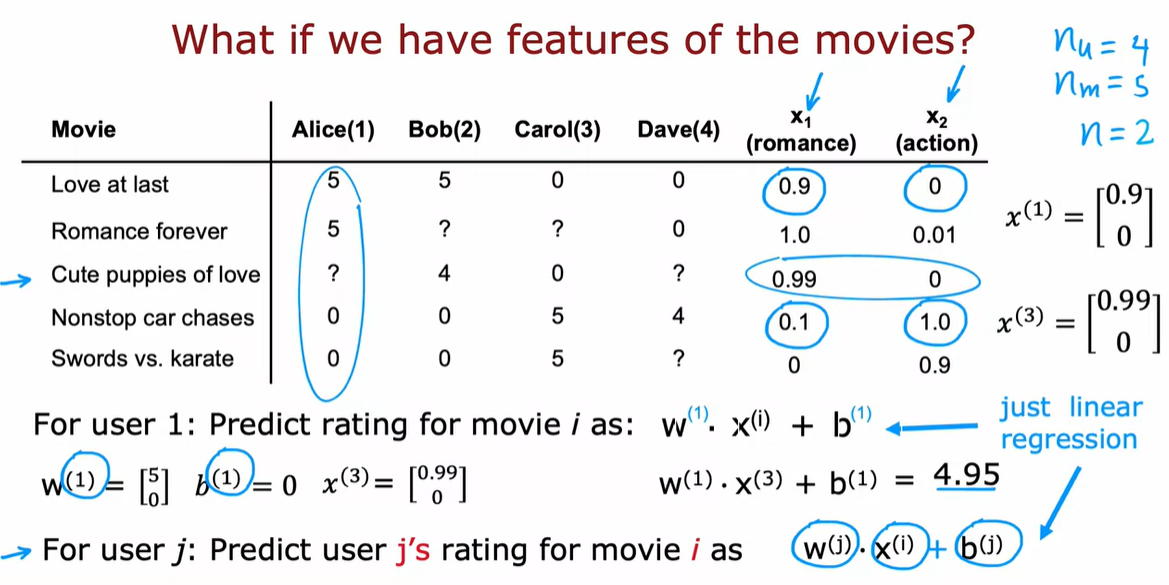
Which is we're going to assume temporarily that we have access to features or

extra information about the movies such as which movies are romance movies,

which movies are action movies.

And using that will start to develop an algorithm.

# Video Using per-item features



So let's take a look at how we can develop a recommended system if we had

features of each item, or features of each movie.

So here's the same data set that we had previously with the four users

having rated some but not all of the five movies.

What if we additionally have features of the movies?

So here I've added two features X1 and X2, that tell us how much

each of these is a romance movie, and how much each of these is an action movie.

So for example Love at Last is a very romantic movie, so

this feature takes on 0.9, but it's not at all an action movie.

So this feature takes on 0.

But it turns out Nonstop Car chases has just a little bit of romance in it.

So it's 0.1, but it has a ton of action.

So that feature takes on the value of 1.0.

So you recall that I had used the notation nu to denote the number of users,

which is 4 and m to denote the number of movies which is 5.

I'm going to also introduce n to denote the number of features we have here.

And so n=2, because we have two features X1 and X2 for each movie.

With these features we have for example that the features for

movie one, that is the movie Love at Last, would be 0.90.

And the features for the third movie

Cute Puppies of Love would be 0.99 and 0.

And let's start by taking a look at how we might make predictions for

Alice's movie ratings.

So for user one, that is Alice,

let's say we predict the rating for

movie i as w.X(i)+b.

So this is just a lot like linear regression.

For example if we end up choosing the parameter w(1)=[5,0] and

say b(1)=0, then the prediction for

movie three where the features are 0.99 and 0,

which is just copied from here, first feature 0.99, second feature 0.

Our prediction would be w.X(3)+b=0.99

times 5 plus 0 times zero,

which turns out to be equal to 4.95.

And this rating seems pretty plausible.

It looks like Alice has given high ratings to Love at Last and Romance Forever,

to two highly romantic movies, but given low ratings to the action movies,

Nonstop Car Chases and Swords vs Karate.

So if we look at Cute Puppies of Love,

well predicting that she might rate that 4.95 seems quite plausible.

And so these parameters w and b for

Alice seems like a reasonable model for predicting her movie ratings.

Just add a little the notation because we have not just one user but

multiple users, or really nu equals 4 users.

I'm going to add a superscript 1 here to denote that this is the parameter w(1) for

user 1 and add a super strip 1 there as well.

And similarly here and here as well, so that we would actually have

different parameters for each of the 4 users on data set.

And more generally in this model we can for user j,

not just user 1 now, we can predict user j's rating for

movie i as w(j).X(i)+b(j).

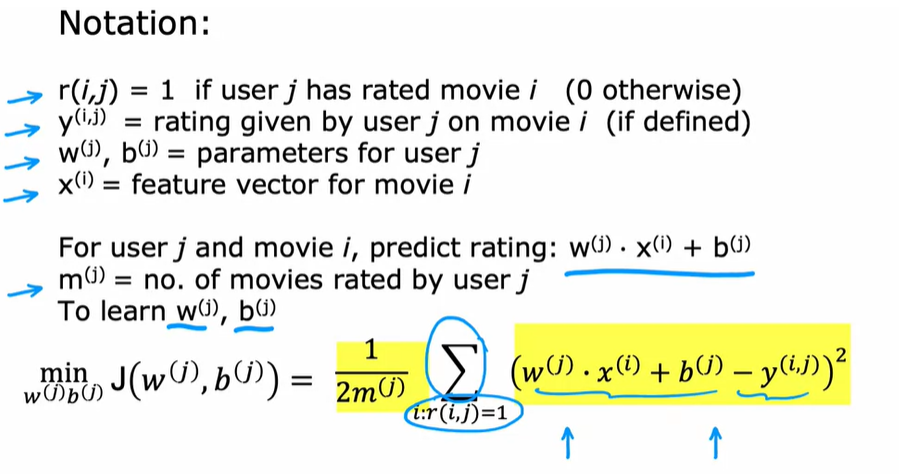
So here the parameters w(j) and

b(j) are the parameters used to predict user j's rating for

movie i which is a function of X(i), which is the features of movie i.

And this is a lot like linear regression, except that we're fitting a different

linear regression model for each of the 4 users in the dataset.



So let's take a look at how we can formulate the cost function for

this algorithm.

As a reminder on notation is that r(i.,j)=1

if user j has rated movie i or 0 otherwise.

And y(i,j)=rating given by user j on movie i.

And on the previous side we defined w(j), b(j) as the parameters for user j.

And X(i) as the feature vector for movie i.

So the model we have is for user j and

movie i predict the rating to be w(j).X(i)+b(j).

I'm going to introduce just one new piece of notation,

which is I'm going to use m(j) to denote the number of movies rated by user j.

So if the user has rated 4 movies, then m(j) would be equal to 4.

And if the user has rated 3 movies then m(j) would be equal to 3.

So what we'd like to do is to learn the parameters w(j) and

b(j), given the data that we have.

That is given the ratings a user has given of a set of movies.

So the average we're going to use is very similar to linear regression.

So let's write out the cost function for learning the parameters w(j) and

b(j) for a given user j.

And let's just focus on one user on user j for now.

I'm going to use the mean squared error criteria.

So the cost will be the prediction, which is w(j).X(i)+b(j)

minus the actual rating that the user had given.

So minus y(i,j) squared.

And we're trying to choose parameters w and

b to minimize the squared error between the predicted rating and

the actual rating that was observed.

But the user hasn't rated all the movies, so

if we're going to sum over this, we're going to

sum over only over the values of i where r(i,j)=1.

So we're going to sum only over the movies i that user j has actually rated.

So that's what this denotes, sum of all values of i where r(i,j)=1.

Meaning that user j has rated that movie i.

And then finally we can take the usual normalization 1 over m(j).

And this is very much like the cost function we have for

linear regression with m or really m(j) training examples.

Where you're summing over the m(j) movies for which you have a rating taking

a squared error and the normalizing by this 1 over 2m(j).

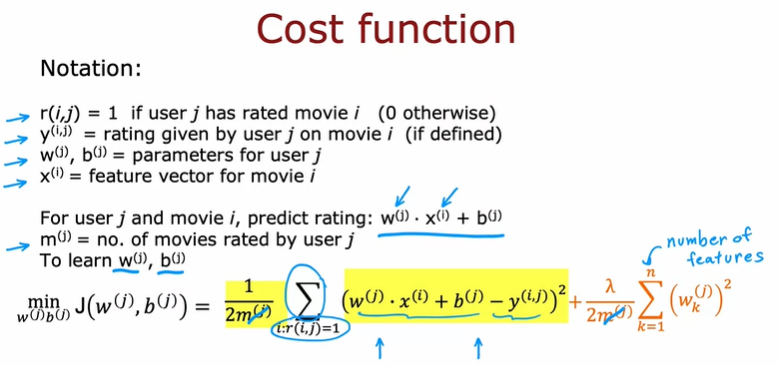
And this is going to be a cost

function J of w(j), b(j).

And if we minimize this as a function of w(j) and b(j),

then you should come up with a pretty good choice of parameters w(i) and b(j).

For making predictions for user j's ratings



Let me have just one more term to this cost function,

which is the regularization term to prevent overfitting.

And so here's our usual regularization parameter,

lambda divided by 2m(j) and

then times as sum of the squared values of the parameters w.

And so n is a number of numbers in X(i) and

that's the same as a number of numbers in w(j).

If you were to minimize this cost function J as a function of w and

b, you should get a pretty good set of parameters for

predicting user j's ratings for other movies.

Now, before moving on, it turns out that for recommended systems it would

be convenient to actually eliminate this division by m(j) term,

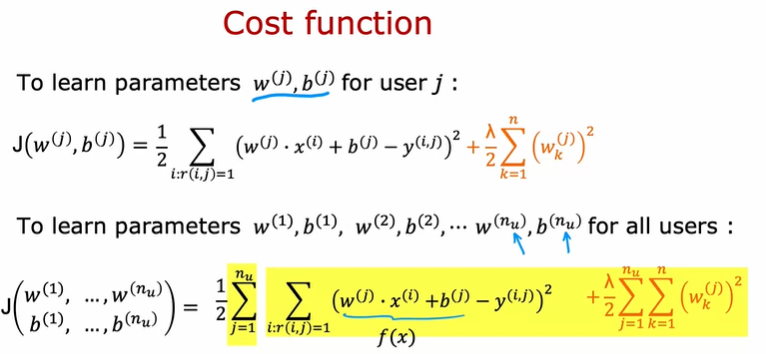
m(j) is just a constant in this expression.

And so, even if you take it out,

you should end up with the same value of w and b.

Now let me take this cost function down here to the bottom and

copy it to the next slide.



.

So we have that to learn the parameters w(j), b(j) for user j.

We would minimize this cost function as a function of w(j) and b(j).

But instead of focusing on a single user,

let's look at how we learn the parameters for all of the users.

To learn the parameters w(1), b(1), w(2),

b(2),...,w(nu), b(nu),

we would take this cost function on top and sum it over all the nu users.

So we would have sum from j=1 one to nu of the same

cost function that we had written up above.

And this becomes the cost for

learning all the parameters for all of the users.

And if we use gradient descent or

any other optimization algorithm to minimize this as a function of w(1),

b(1) all the way through w(nu), b(nu), then you have a pretty

good set of parameters for predicting movie ratings for all the users.

And you may notice that this algorithm is a lot like linear regression,

where that plays a role similar to the output f(x) of linear regression.

Only now we're training a different linear regression model for each of the nu users.

So that's how you can learn parameters and predict movie ratings,

if you had access to these features X1 and X2.

That tell you how much is each of the movies, a romance movie, and

how much is each of the movies an action movie?

But where do these features come from?

And what if you don't have access to such features that give you enough detail

about the movies with wish to make these predictions?

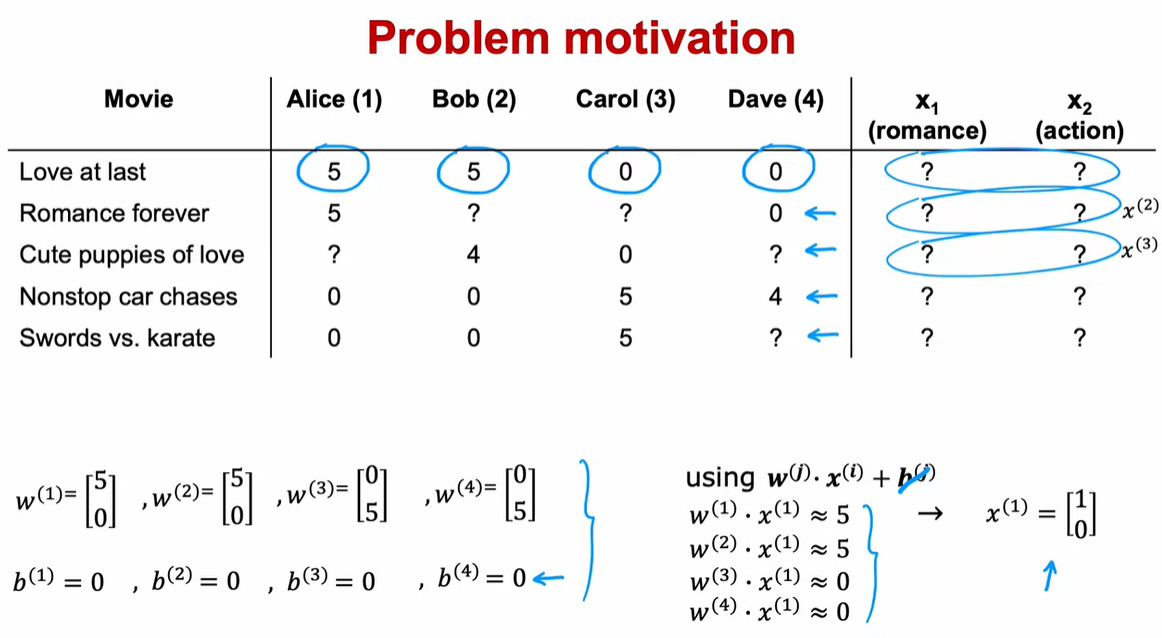
In the next video, we'll look at the modification of this algorithm. They'll let you make predictions that you make recommendations.

Even if you don't have an advanced features that describe the items of

the movies in sufficient detail to run the algorithm that we just saw.

Let's go on and take a look at that in the next video

# Video Collaborative Filtering Algorithm



In the last video, you saw how

if you have features for each movie,

such as features x\_1 and x\_2 that tell you how much is

this a romance movie and how

much is this an action movie.

Then you can use basically linear regression

to learn to predict movie ratings.

But what if you don't have those features, x\_1 and x\_2?

Let's take a look at how you can learn or come up

with those features x\_1 and x\_2 from the data.

Here's the data that we had before.

But what if instead of

having these numbers for x\_1 and x\_2,

we didn't know in advance what

the values of the features x\_1 and x\_2 were?

I'm going to replace them with question marks over here.

Now, just for the purposes of illustration,

let's say we had somehow already

learned parameters for the four users.

Let's say that we learned parameters w^1

equals 5 and 0 and b^1 equals 0, for user one.

W^2 is also 5, 0 b^2, 0.

W^3 is 0,

5 b^3 is 0,

and for user four W^4 is also 0,

5 and b^4 0, 0.

We'll worry later about how we might have

come up with these parameters, w and b.

But let's say we have them already.

As a reminder, to predict music j's rating on movie i,

we're going to use w^j dot product,

the features of x\_i plus b^j.

To simplify this example,

all the values of b are actually equal to 0.

Just to reduce a little bit of writing,

I'm going to ignore b for the rest of this example.

Let's take a look at how we can try to guess

what might be reasonable features for movie one.

If these are the parameters you have on the left,

then given that Alice rated movie one, 5,

we should have that w^1.x^1 should be about equal to

5 and w^2.x^2 should

also be about equal to 5 because Bob rated it 5.

W^3.x^1 should be close to 0

and w^4.x^1 should be close to 0 as well.

The question is, given

these values for w that we have up here,

what choice for x\_1 will cause these values to be right?

Well, one possible choice would be if

the features for that first movie,

were 1, 0 in which case,

w^1.x^1 will be equal to 5,

w^2.x^1 will be equal to 5 and similarly,

w^3 or w^4 dot product with

this feature vector x\_1 would be equal to 0.

What we have is that if you

have the parameters for all four users here,

and if you have four ratings

in this example that you want to try to match,

you can take a reasonable guess at what lists a feature

vector x\_1 for movie one that

would make good predictions

for these four ratings up on top.

Similarly, if you have these parameter vectors,

you can also try to come up with

a feature vector x\_2 for the second movie,

feature vector x\_3 for the third movie,

and so on to try to make the algorithm's predictions on

these additional movies close

to what was actually the ratings given by the users.

Let's come up with a cost function for actually

learning the values of x\_1 and x\_2.

By the way, notice that this works only

because we have parameters for four users.

That's what allows us to try

to guess appropriate features, x\_1.

This is why in a typical linear regression application

if you had just a single user,

you don't actually have

enough information to figure

out what would be the features,

x\_1 and x\_2, which is why in

the linear regression contexts that you saw in course 1,

you can't come up with features x\_1 and x\_2 from scratch.

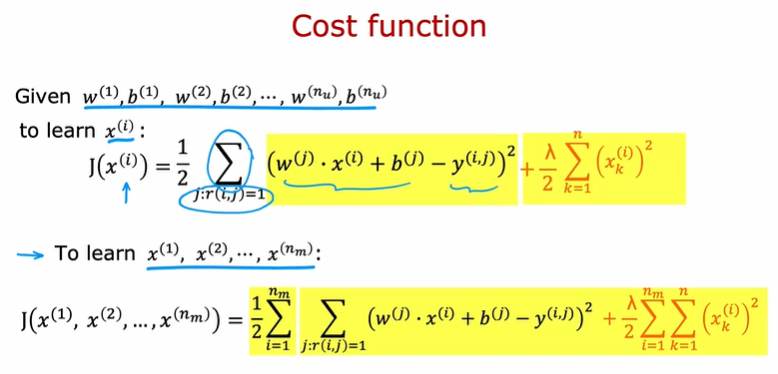
But in collaborative filtering,

is because you have ratings from

multiple users of the same item with the same movie.

That's what makes it possible to try to

guess what are possible values for these features.



Given w^1, b^1, w^2, b^2,

and so on through w^n\_u and b^n\_u,

for the n subscript u users.

If you want to learn

the features x^i for a specific movie,

i is a cost function we could use which is that.

I'm going to want to minimize squared error as usual.

If the predicted rating by

user j on movie i is given by this,

let's take the squared difference

from the actual movie rating y,i,j.

As before, let's sum over all the users j.

But this will be a sum over

all values of j, where r, i,

j is equal to I. I'll add a 1.5 there as usual.

As I defined this as a cost function for x^i.

Then if we minimize this as a function of x^i

you be choosing the features for movie i.

So therefore all the users J that have rated movie i,

we will try to minimize the squared difference between

what your choice of features x^i results in terms of

the predicted movie rating minus

the actual movie rating that the user had given it.

Then finally, if we want to add a regularization term,

we add the usual plus Lambda over 2,

K equals 1 through n,

where n as usual is the number of

features of x^i squared.

Lastly, to learn all the features

x1 through x^n\_m because we have n\_m movies,

we can take this cost function on

top and sum it over all the movies.

Sum from i equals 1 through

the number of movies and then just take

this term from above and this becomes

a cost function for learning

the features for all of the movies in the dataset.

So if you have parameters w and b, all the users,

then minimizing this cost function as a function

of x1 through x^n\_m

using gradient descent or cellular algorithm,

this will actually allow you to take a pretty good guess

at learning good features for the movies.

This is pretty remarkable

for most machine learning applications

the features had to be

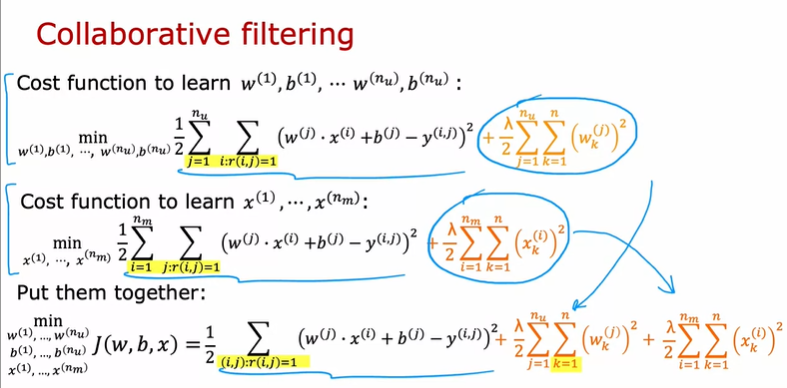
externally given but in this algorithm,

we can actually learn the features for a given movie.

But what we've done so far in this video,

we assumed you had

those parameters w and b for the different users.



Where do you get those parameters from?

Well, let's put together the algorithm from

the last video for learning w and b and what we

just talked about in this video for learning x and

that will give us our collaborative filtering algorithm.

Here's the cost function for learning the features.

This is what we had derived on the last slide.

Now, it turns out that if we put these two together,

this term here is exactly the same as this term here.

Notice that sum over j of all values

of i is that r,i,j equals 1 is the

same as summing over all values of

i with all j where r,i,j is equal to 1.

This summation is just summing over

all user movie pairs where there is a rating.

What I'm going to do is put

these two cost functions together and have

this where I'm just writing out the summation more

explicitly as summing over all pairs i and j,

where we do have a rating

of the usual squared cost function and then let

me take the regularization term

from learning the parameters w and b,

and put that here and take

the regularization term from

learning the features x and put them

here and this ends up being

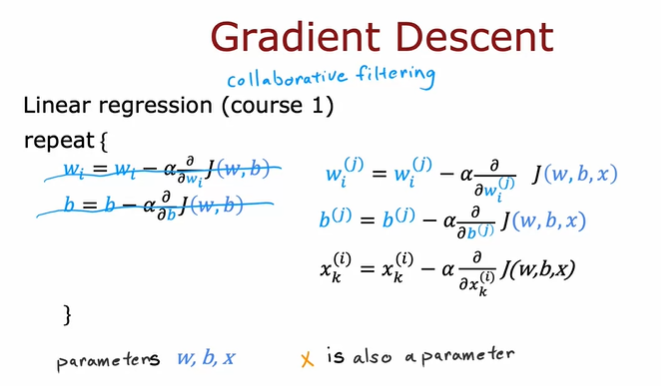
our overall cost function for learning w, b, and x.

It turns out that if you minimize

this cost function as a function of w and b as y as x,

then this algorithm actually works.

Here's what I mean.



How do you minimize this cost function

as a function of w, b, and x?

One thing you could do is to use gradient descent.

In course 1 when we learned about linear regression,

this is the gradient descent algorithm you had seen,

where we had the cost function J,

which is a function of the parameters w and b,

and we'd apply gradient descent as follows.

With collaborative filtering, the cost function is in

a function of just w and b is now a function of w,

b, and x. I'm using

w and b here to denote the parameters for all

of the users and x here just

informally to denote the features of all of the movies.

But if you're able to take

partial derivatives with respect

to the different parameters,

you can then continue to update

the parameters as follows.

But now we need to optimize

this with respect to x as well.

We also will want to update each of

these parameters x using gradient descent as follows.

It turns out that if you do this,

then you actually find pretty good values

of w and b as well as x.

In this formulation of the problem,

the parameters of w and b,

and x is also a parameter.

Then finally, to learn the values of x,

we also will update x as x

minus the partial derivative respect to x of the cost w,

b, x. I'm using the notation here a little bit

informally and not keeping

very careful track of the superscripts and subscripts,

but the key takeaway I hope you have from this is

that the parameters to this model are w and b,

and x now is also a parameter,

which is why we minimize the cost function as

a function of all three of these sets of parameters,

w and b, as well as x.

The average we derived is

called collaborative filtering,

and the name collaborative filtering

refers to the sense that

because multiple users have

rated the same movie collaboratively,

given you a sense of what this movie maybe like,

that allows you to guess what

are appropriate features for that movie,

and this in turn allows you to

predict how other users that

haven't yet rated that same movie

may decide to rate it in the future.

This collaborative filtering is

this gathering of data from multiple users.

This collaboration between users to help you predict

ratings for even other users in the future.

So far, our problem formulation has used

movie ratings from 1- 5 stars or from 0- 5 stars.

A very common use case of recommended systems is when you

have binary labels such as that the user favors,

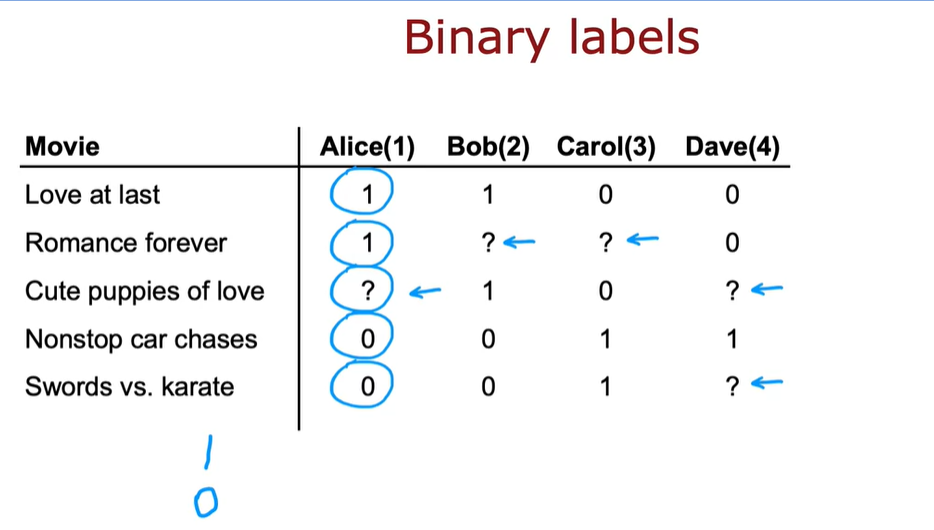
or like, or interact with an item.

In the next video, let's take a look at a generalization

of the model that you've seen so far to binary labels.

Let's go see that in the next video.

# Video Binary labels: Favs, Likes, and Clicks



Many important applications of recommended systems or

collective filtering algorithms involved binary labels where instead of

a user giving you a one to five star or zero to five star rating, they just

somehow give you a sense of they like this item or they did not like this item.

Let's take a look at how to generalize the algorithm you've seen to this setting.

The process we'll use to generalize the algorithm will be very much reminiscent

to how we have gone from linear regression to logistic regression, to predicting

numbers to predicting a binary label back in course one, let's take a look.

Here's an example of a collaborative filtering data set with binary labels.

A one the notes that the user liked or engaged with a particular movie.

So label one could mean that Alice watched the movie Love at last all the way to

the end and watch romance forever all the way to the end.

But after playing a few minutes of nonstop car chases decided to stop the video and

move on.

Or it could mean that she explicitly hit like or

favorite on an app to indicate that she liked these movies.

But after checking out nonstop cloud chasers and

swords versus karate did not hit like.

And the question mark usually means the user has not yet seen the item and so

they weren't in a position to decide whether or not to hit like or

favorite on that particular item.

So the question is how can we take the collaborative filtering average that you

saw in the last video and get it to work on this daaset.

And by predicting how likely Alice, Bob carol and

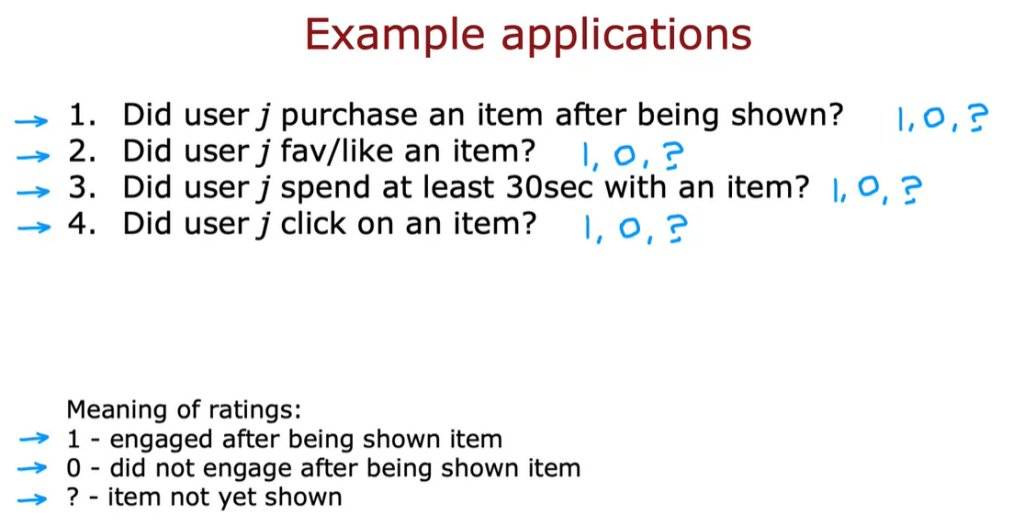
Dave are to like the items that they have not yet rated,

we can then decide how much we should recommend these items to them.

There are many ways of defining what is the label one and what is the label zero,

and what is the label question mark in collaborative filtering with binary

labels.



Let's take a look at a few examples.

In an online shopping website, the label could denote whether or

not music they chose to purchase an item after they were exposed to it,

after they were shown the item.

So one would denote that they purchase it zero would denote that they did

not purchase it.

And the question mark would denote that they were not even shown were not even

exposed to the item.

Or in a social media setting, the labels one or

zero could denote did the user favorite or like an item after they were shown it.

And question mark would be if they have not yet been shown the item or

many sites instead of asking for explicit user rating will use

the user behavior to try to guess if the user like the item.

So for example, you can measure if a user spends at least 30 seconds of an item.

And if they did, then assign that a label one because the user found the item

engaging or if a user was shown an item but

did not spend at least 30 seconds with it, then a sign that a label zero.

Or if the user was not shown the item yet, then assign it a question mark.

Another way to generate a rating implicitly as a function

of the user behavior will be to see that the user click on an item.

This is often done in online advertising where if the user has been shown an ad,

if they clicked on it assigned to the labor one,

if they did not click assigned to the label zero and the question mark were

referred to if the user has not even been shown that ad in the first place.

So often these binary labels will have a rough meaningless follows.

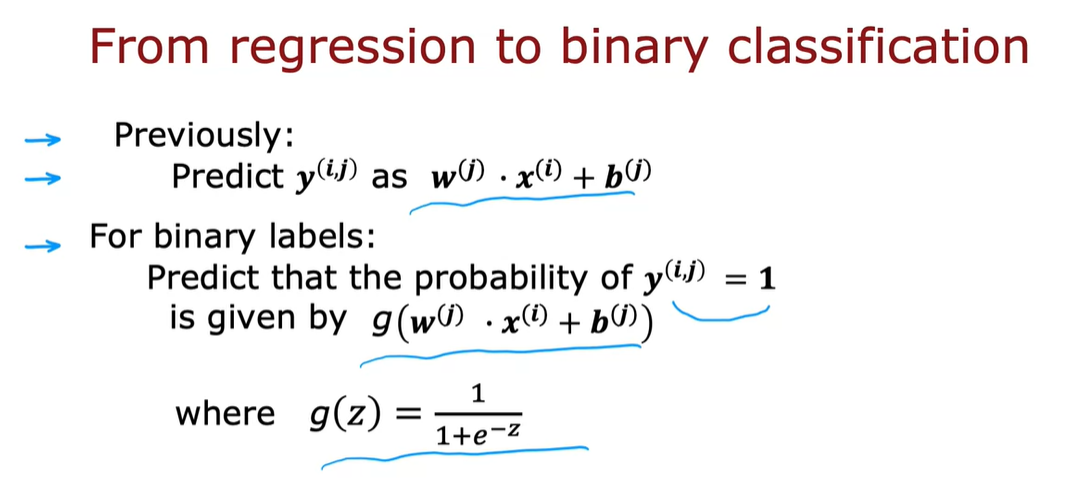
A labor of one means that the user engaged after being shown an item And

engaged could mean that they clicked or spend 30 seconds or

explicitly favorite or like to purchase the item.

A zero will reflect the user not engaging after being shown the item,

the question mark will reflect the item not yet having been shown to the user.



So given these binary labels,

let's look at how we can generalize our algorithm which is a lot like linear

regression from the previous couple videos to predicting these binary outputs.

Previously we were predicting label yij as wj.xi+b.

So this was a lot like a linear regression model.

For binary labels, we're going to predict

that the probability of yijb=1 is given by not wj.xi+b.

But it said by g of this formula,

we're now g(z) 1/1 +e to the -z.

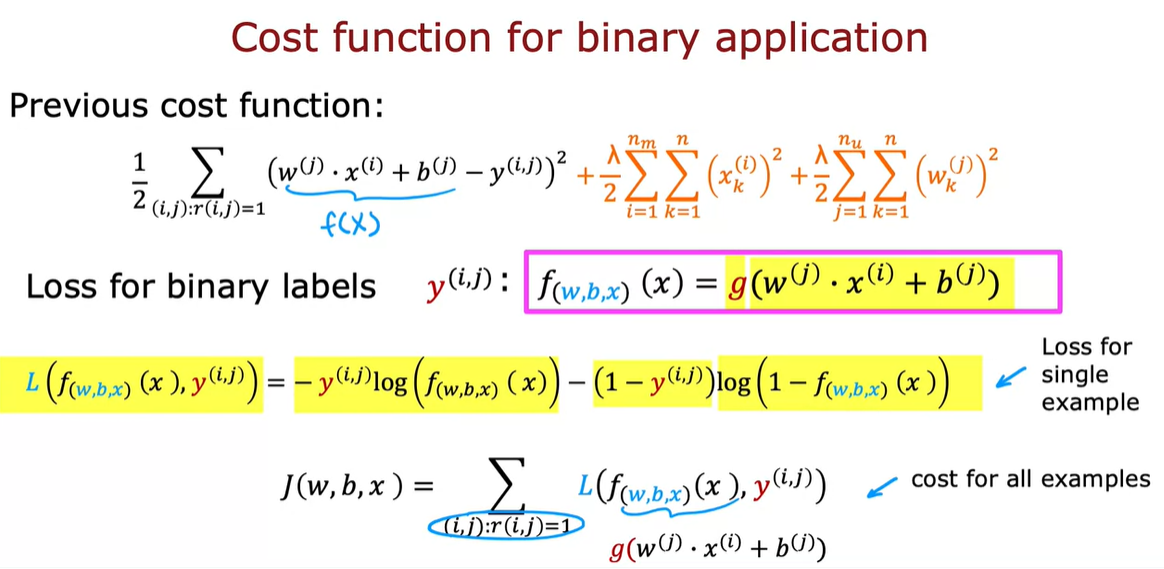
So this is the logistic function just like we saw in logistic regression.

And what we would do is take what was a lot like a linear regression model and

turn it into something that would be a lot like a logistic regression

model where will now predict the probability of yij being

1 that is of the user having engaged with or like the item using this model.



In order to build this algorithm,

we'll also have to modify the cost function from the squared error

cost function to the cost function that is more appropriate for

binary labels for a logistic regression like model.

So previously, this was the cost function that we had where this term

play their role similar to f(x), the prediction of the algorithm.

When you now have binary labels,

yij when the labels are one or zero or

question mark, then the prediction f(x)

becomes instead of wj.xi+b j it becomes g

of this where g is the logistic function.

And similar to when we had derived logistic regression,

we had written out the following loss function for

a single example which was at the loss if the algorithm predicts f(x) and

the true label was y, the loss was this.

It was -y log

f-y log 1-f.

This is also sometimes called the binary cross entropy cost function.

But this is a standard cost function that we used for logistic regression as was for

the binary classification problems when we're training neural networks.

And so to adapt this to the collaborative filtering setting,

let me write out the cost function which is now a function of all

the parameters w and b as well as all the parameters x

which are the features of the individual movies or items of.

We now need to some over all the pairs ij where riij=1 notice

this is just similar to the summation up on top.

And now instead of this squared error cost function,

we're going to use that loss function.

There's a function of f(x), yij.

Where f(x) here?

That's my abbreviation.

My shorthand for g(w) j.x1+ej.

As we plug this into here, then this gives you the cost function

they could use for collaborative filtering on binary labels.

So that's it.

That's how you can take the linear regression,

like collaborative filtering algorithm and generalize it to work with binary labels.

And this actually very significantly opens up the set of applications you can address

with this algorithm.

Now, even though you've seen the key structure and

cost function of the algorithm, there are also some implementation,

all tips that will make your algorithm work much better.

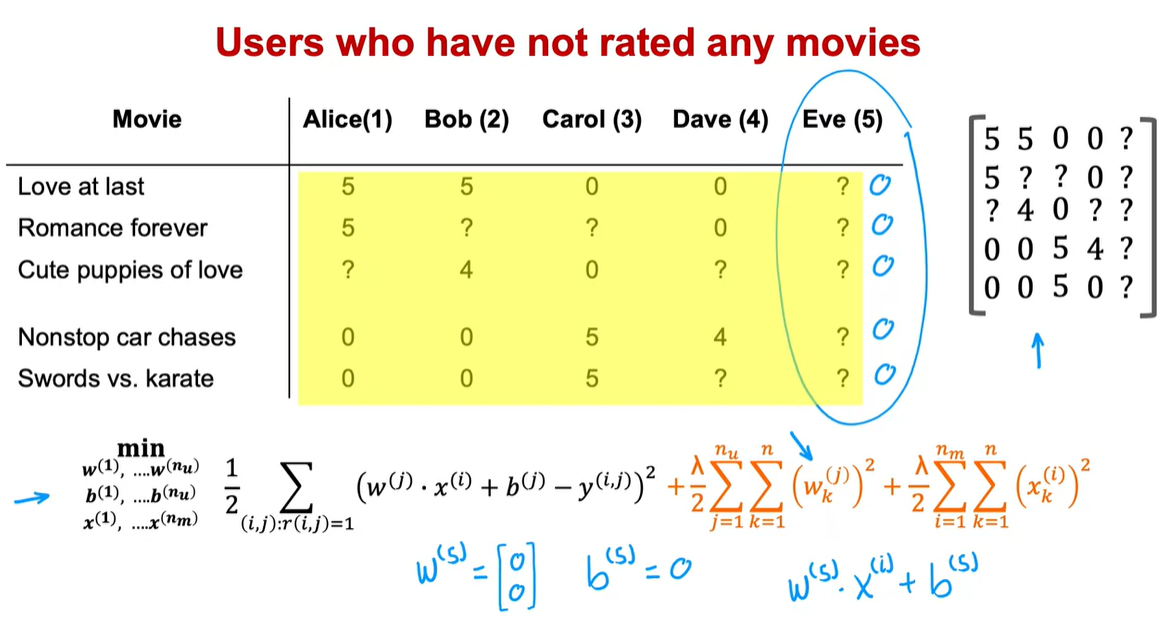
Let's go on to the next video to take a look at some details of how you implement

it and some little modifications that make the album run much faster.

Let's go on to the next video.

################### Recommender sytems implementation detail ##################

# Video Mean Normalization



Back in the first course, you have seen how for linear regression,

future normalization can help the algorithm run faster.

In the case of building a recommended system

with numbers wide such as movie ratings from one to five or

zero to five stars, it turns out your algorithm will run more efficiently.

And also perform a bit better if you first carry out mean normalization.

That is if you normalize the movie ratings to have a consistent average value,

let's take a look at what that means.

So here's the data set that we've been using.

And down below is the cost function you used to learn the parameters for

the model.

In order to explain mean normalization,

Im ctually going to add fifth user Eve who has not yet rated any movies.

And you see in a little bit that adding mean normalization will

help the algorithm make better predictions on the user Eve.

In fact, if you were to train a collaborative filtering algorithm

on this data, then because we are trying to make the parameters w

small because of this regularization term.

If you were to run the algorithm on this dataset,

you actually end up with the parameters w for the fifth user,

for the user Eve to be equal to [0 0] as well as quite likely b(5) = 0.

Because Eve hasn't rated any movies yet, the parameters w and

b don't affect this first term in the cost function because none of

Eve's movie's rating play a role in this squared error cost function.

And so minimizing this means making the parameters w as small as possible.

We didn't really regularize b.

But if you initialize b to 0 as the default, you end up with b(%) = 0 as well.

But if these are the parameters for user 5 that is for

Eve, then what the average will end up doing is predict

that all of Eve's movies ratings with be w(%).x for movie i + b(5).

And this is equal to 0 if w and b above equals 0.

And so this algorithm will predict that if you have a new user that has not yet

rated anything, we think they'll rate all movies with zero stars and

that's not particularly hopeful.

So in this video, we'll see that mean normalization will help this

algorithm come up with better predictions of the movie ratings for

a new user that has not yet rated any movies.

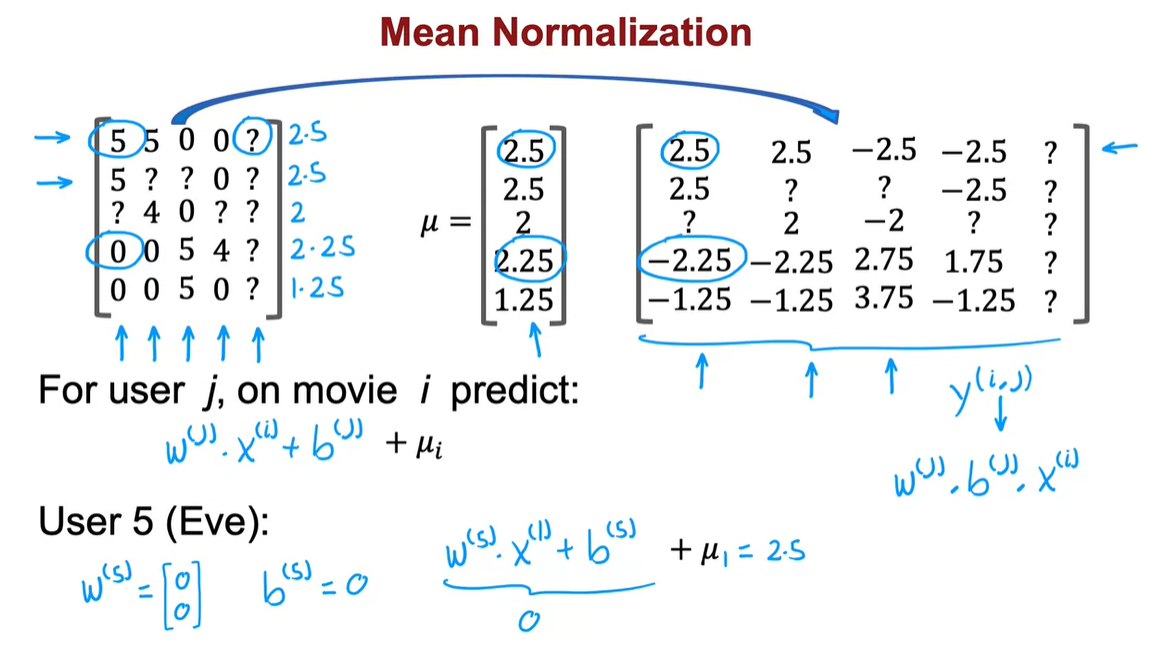
In order to describe mean normalization,

let me take all of the values here including all the question marks for

Eve and put them in a two dimensional matrix like this.

Just to write out all the ratings including the question marks in a more

sustained and more compact way.



To carry out mean normalization,

what we're going to do is take all of these ratings and for

each movie, compute the average rating that was given.

So movie one had two 5s and two 0s and so the average rating is 2.5.

Movie two had a 5 and a 0, so that averages out to 2.5.

Movie three 4 and 0 averages out to 2.

Movie four averages out to 2.25 rating.

And movie five not that popular, has an average 1.25 rating.

So I'm going to take all of these five numbers and

gather them into a vector which I'm going to call μ because this is the vector

of the average ratings that each of the movies had.

Averaging over just the users that did read that particular movie.

Instead of using these original 0 to 5 star ratings over here, I'm

going to take this and subtract from every rating the mean rating that it was given.

So for example this movie rating was 5.

I'm going to subtract 2.5 giving me 2.5 over here.

This movie had a 0 star rating.

I'm going to subtract 2.25 giving me a -2.25 rating and so on for all

of the now five users including the new user Eve as well as for all five movies.

Then these new values on the right become your new values of Y(i,j).

We're going to pretend that user 1 had given a 2.5 rating to movie one and

the -2.25 rating to movie four.

And using this, you can then learn w(j),

b(j) and x(i) same as before for user j on movie i,

you would predict w(j).x(i) + b(j).

But because we had subtracted off µi for movie i during this mean

normalization step, in order to predict not a negative star

rating which is impossible for user rates from 0 to 5 stars.

We have to add back this µi which is just the value we have subtracted out.

So as a concrete example, if we look at what happens with user

5 with the new user Eve because she had not yet rated any movies,

the average might learn parameters w(5) = [0 0] and say b(5) = 0.

And so if we look at the predicted rating for

movie one, we will predict that Eve will

rate it w(5).x1 + b(5) but this is 0 and

then + µ1 which is equal to 2.5.

So this seems more reasonable to think Eve is likely to rate this movie

2.5 rather than think Eve will rate all movie zero stars just because she

hasn't rated any movies yet.

And in fact the effect of this algorithm is it will cause

the initial guesses for the new user Eve to be just equal to

the mean of whatever other users have rated these five movies.

And that seems more reasonable to take the average rating of the movies

rather than to guess that all the ratings by Eve will be zero.

It turns out that by normalizing the mean of the different movies ratings

to be zero, the optimization algorithm for

the recommended system will also run just a little bit faster.

But it does make the algorithm behave much better for

users who have rated no movies or very small numbers of movies.

And the predictions will become more reasonable.

In this example,

what we did was normalize each of the rows of this matrix to have zero mean and

we saw this helps when there's a new user that hasn't rated a lot of movies yet.

There's one other alternative that you could use which is to instead

normalize the columns of this matrix to have zero mean.

And that would be a reasonable thing to do too.

But I think in this application, normalizing the rows so

that you can give reasonable ratings for

a new user seems more important than normalizing the columns.

Normalizing the columns would hope if there was a brand new movie that no one

has rated yet.

But if there's a brand new movie that no one has rated yet,

you probably shouldn't show that movie to too many users initially because you

don't know that much about that movie.

So normalizing columns the hope with the case of a movie with no ratings

seems less important to me than normalizing the rules

to hope with the case of a new user that's hardly rated any movies yet.

And when you are building your own recommended system in this week's

practice lab, normalizing just the roles should work fine.

So that's mean normalization.

It makes the algorithm run a little bit faster.

But even more important, it makes the algorithm give much better, much

more reasonable predictions when there are users that rated very few movies or

even no movies at all.

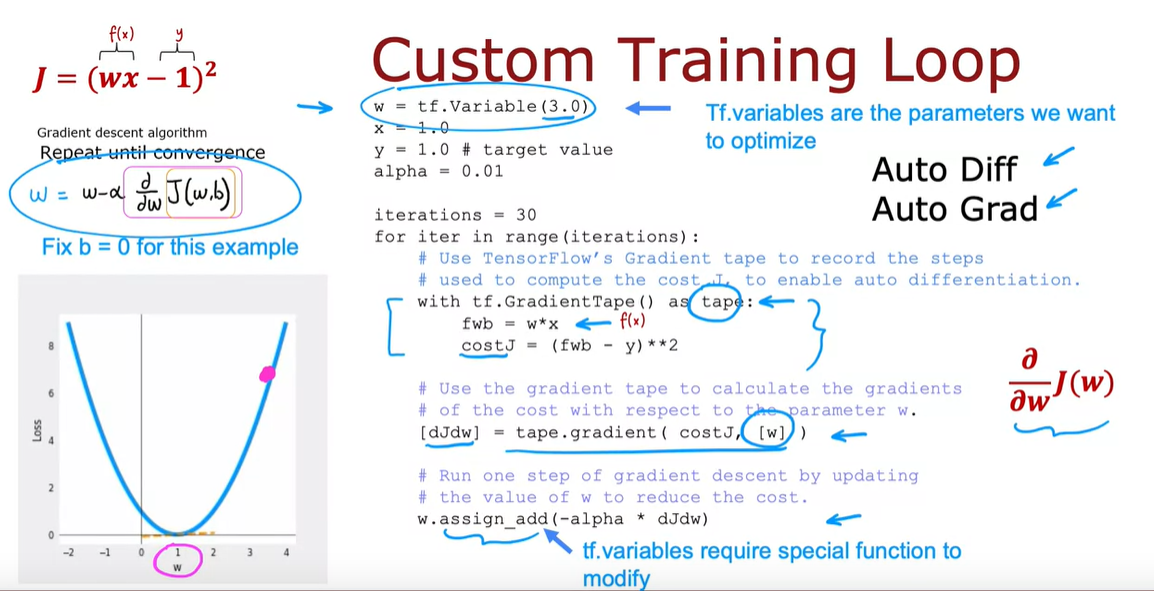
This implementation detail of mean normalization will make your recommended

system work much better.

Next, let's go into the next video to talk about how you can implement this for

yourself intensive flow.

# Video Tensorflow Implementation of collaborative filtering



In this video, we'll take a look at how you can use TensorFlow to implement

the collaborative filtering algorithm.

You might be used to thinking of TensorFlow as a tool for

building neural networks.

And it is.

It's a great tool for building neural networks.

And it turns out that TensorFlow can also be very hopeful for

building other types of learning algorithms as well.

Like the collaborative filtering algorithm.

One of the reasons I like using TensorFlow for talks like these is that for

many applications in order to implement gradient descent,

you need to find the derivatives of the cost function, but TensorFlow

can automatically figure out for you what are the derivatives of the cost function.

All you have to do is implement the cost function and without needing to know

any calculus, without needing to take derivatives yourself,

you can get TensorFlow with just a few lines of code

to compute that derivative term, that can be used to optimize the cost function.

Let's take a look at how all this works.

You might remember this diagram here on the right from course one.

This is exactly the diagram that we had looked at when we talked

about optimizing w.

When we were working through our first linear regression example.

And at that time we had set b=0.

And so the model was just predicting f(x)=w.x.

And we wanted to find the value of w that minimizes the cost function J.

So the way we were doing that was via a gradient descent update,

which looked like this, where w gets repeatedly updated as w minus

the learning rate alpha times the derivative term.

If you are updating b as well, this is the expression you will use.

But if you said b=0, you just forgo the second update and

you keep on performing this gradient descent update until convergence.

Sometimes computing this derivative or partial derivative term can be difficult.

And it turns out that TensorFlow can help with that.

Let's see how.

I'm going to use a very simple cost

function J=(wx-1) squared.

So wx is our simplified f w of x and

y is equal to 1.

And so this would be the cost function if we had f(x) equals

wx,y equals 1 for the one training example that we have, and

if we were not optimizing this respect to b.

So the gradient descent algorithm will repeat until convergence

this update over here.

It turns out that if you implement the cost function J over here,

TensorFlow can automatically compute for

you this derivative term and thereby get gradient descent to work.

I'll give you a high level overview of what this code does, w=tf.variable(3.0).

Takes the parameter w and initializes it to the value of 3.0.

Telling TensorFlow that w is a variable is how we tell

it that w is a parameter that we want to optimize.

I'm going to set x=1.0, y=1.0, and the learning rate alpha to be equal to 0.01.

And let's run gradient dissent for 30 iterations.

So in this code will still do for iter in range iterations, so for 30 iterations.

And this is the syntax to get TensorFlow to automatically compute the rotors

for you.

TensorFlow has a feature called a gradient tape.

And if you write this with tf our gradient tape as tape f.

This is compute f(x) as w\*x and

compute J as f(x)-y squared.

Then by telling TensorFlow how to compute to costJ, and

by doing it with the gradient taped syntax as follows,

TensorFlow will automatically record the sequence of steps.

The sequence of operations needed to compute the costJ.

And this is needed to enable automatic differentiation.

Next TensorFlow will have saved the sequence of operations in tape,

in the gradient tape.

And with this syntax, TensorFlow will automatically

compute this derivative term, which I'm going to call dJdw.

And TensorFlow knows you want to take the derivative respected w.

That w is the parameter you want to optimize because you had told it so

up here.

And because we're also specifying it down here.

So now the computer derivatives, finally you can carry out this

update by taking w and subtracting from it the learning rate

alpha times that derivative term that we just got from up above.

TensorFlow variables, tier variables requires special handling.

Which is why instead of setting w to be w minus alpha times

the derivative in the usual way, we use this assigned add function.

But when you get to the practice lab, don't worry about it.

We'll give you all the syntax you need in order to implement the collateral

filtering algorithm correctly.

So notice that with the gradient tape feature of TensorFlow,

the main work you need to do is to tell it how to compute the cost function J.

And the rest of the syntax causes TensorFlow to

automatically figure out for you what is that derivative?

And with this TensorFlow we'll start with finding the slope of this,

at 3 shown by this dash line.

Take a gradient step and update w and compute the derivative again and

update w over and over until eventually it gets to

the optimal value of w, which is at w equals 1.

So this procedure allows you to implement gradient descent without ever

having to figure out yourself how to compute this derivative term.

This is a very powerful feature of TensorFlow called Auto Diff.

And some other machine learning packages like pytorch also support Auto Diff.

Sometimes you hear people call this Auto Grad.

The technically correct term is Auto Diff, and

Auto Grad is actually the name of the specific software package for

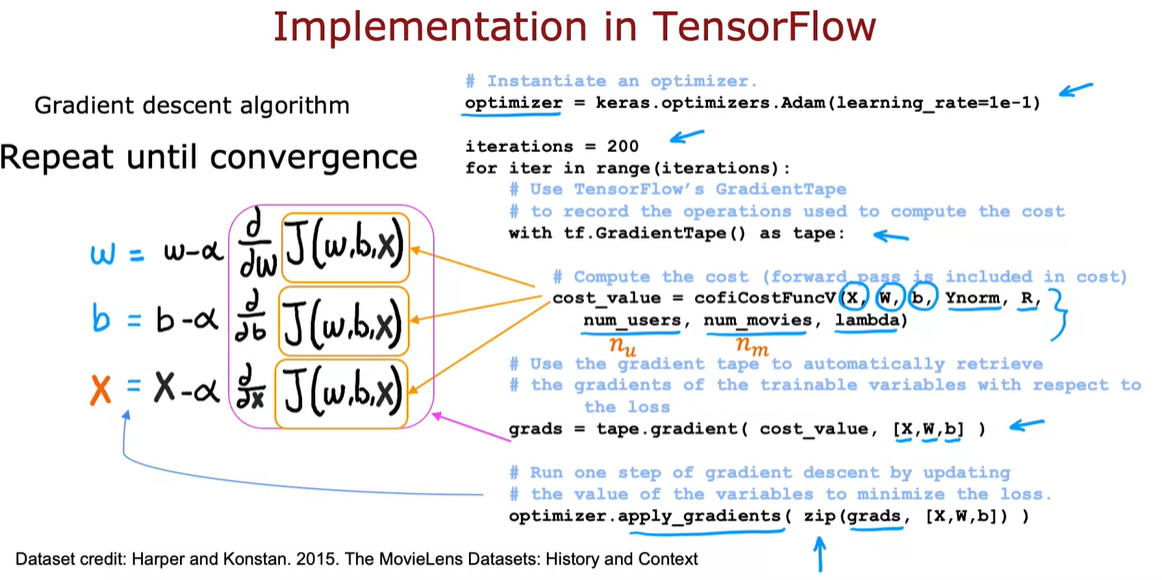
doing automatic differentiation, for taking derivatives automatically.

But sometimes if you hear someone refer to Auto Grad, they're just referring to this

same concept of automatically taking derivatives.

So let's take this and look at how you can implement to collaborative

filtering algorithm using Auto Diff.



And in fact, once you can compute derivatives automatically,

you're not limited to just gradient descent.

You can also use a more powerful optimization algorithm,

like the adam optimization algorithm.

In order to implement the collaborative filtering algorithm TensorFlow,

this is the syntax you can use.

Let's starts with specifying that the optimizer is keras

optimizers adam with learning rate specified here.

And then for say, 200 iterations,

here's the syntax as before with tf gradient tape, s tape,

you need to provide code to compute the value of the cost function J.

So recall that in collaborative filtering,

the cost function J takes is input parameters x, w, and

b as well as the ratings mean normalized.

So that's why I'm writing y norm, r(i,j) specifying which values have a rating,

number of users or nu in our notation, number of movies or nm in our notation or

just num as well as the regularization parameter lambda.

And if you can implement this cost function J,

then this syntax will cause TensorFlow to figure out the derivatives for you.

Then this syntax will cause TensorFlow to record the sequence of operations used to

compute the cost.

And then by asking it to give you grads equals tape.gradient,

this will give you the derivative of the cost function with respect to x, w, and b.

And finally with the optimizer that we had specified up on top,

as the adam optimizer.

You can use the optimizer with the gradients that we just computed.

And does it function in python is just a function that rearranges the numbers into

an appropriate ordering for the applied gradients function.

If you are using gradient descent for collateral filtering,

recall that the cost function J would be a function of w, b as well as x.

And if you are applying gradient descent,

you take the partial derivative respect the w.

And then update w as follows.

And you would also take the partial derivative of this respect to b.

And update b as follows.

And similarly update the features x as follows.

And you repeat until convergence.

But as I mentioned earlier with TensorFlow and

Auto Diff you're not limited to just gradient descent.

You can also use a more powerful optimization algorithm like the adam

optimizer.

The data set you use in the practice lab is a real data set comprising

actual movies rated by actual people.

This is the movie lens dataset and it's due to Harper and Konstan.

And I hope you enjoy running this algorithm on a real data set of movies,

and ratings and see for yourself the results that this algorithm can get.

So that's it.

That's how you can implement the collaborative filtering algorithm in

TensorFlow.

If you're wondering why do we have to do it this way?

Why couldn't we use a dense layer and then model compiler and model fit?

The reason we couldn't use that old recipe is, the collateral filtering algorithm and

cost function, it doesn't neatly fit into the dense layer or

the other standard neural network layer types of TensorFlow.

That's why we had to implement it this other way where we would

implement the cost function ourselves.

But then use TensorFlow's tools for automatic differentiation,

also called Auto Diff.

And use TensorFlow's implementation of the adam optimization algorithm

to let it do a lot of the work for us of optimizing the cost function.

If the model you have is a sequence of dense neural network layers or

other types of layers supported by TensorFlow, and

the old implementation recipe of model compound model fit works.

But even when it isn't, these tools TensoFlow give you a very effective way to

implement other learning algorithms as well.

And so I hope you enjoy playing more with the collateral filtering exercise in

this week's practice lab.

And looks like there's a lot of code and lots of syntax, don't worry about it.

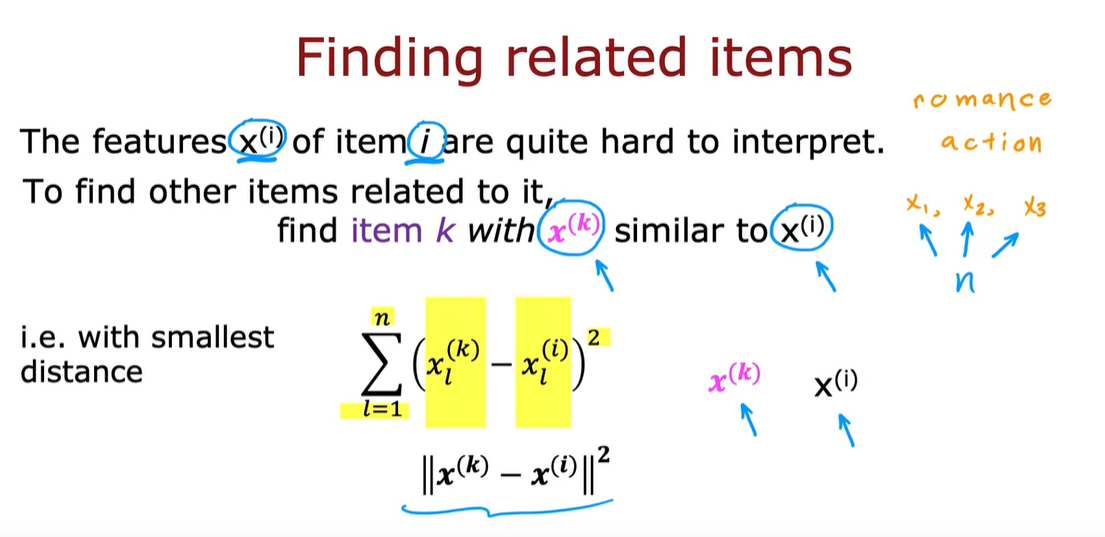
Make sure you have what you need to complete that exercise successfully.

And in the next video, I'd like to also move on to discuss more of the nuances of

collateral filtering and specifically the question of how do you find related items,

given one movie, whether other movies similar to this one.

# Video Finding Related Items



If you come to an online shopping website

and you're looking at a specific item,

say maybe a specific book,

the website may show you things like,

"Here are some other books similar to this

one" or if you're browsing a specific movie,

it may say, "Here are

some other movies similar to this one."

How do the websites do that?,

so that when you're looking at one item,

it gives you other similar or related items to consider.

It turns out the collaborative

filtering algorithm that we've been talking

about gives you a nice way to

find related items. Let's take a look.

As part of the collaborative filtering we've discussed,

you learned features x^(i) for every item i,

for every movie i or other type of

item they're recommending to users.

Whereas early this week,

I had used a hypothetical example of the features

representing how much a movie

is a romance movie versus an action movie.

In practice, when you use this algorithm to

learn the features x^(i) automatically,

looking at the individual features x\_1,

x\_2, x\_3,

you find them to be quite hard to interpret.

Is quite hard to learn features and say,

x\_1 is an action movie

and x\_2 is as a foreign film and so on.

But nonetheless, these learned features,

collectively x\_1, x\_2, x\_3,

other many features,

and you have collectively these features

do convey something about what that movie is like.

It turns out that given features x^(i) of item i,

if you want to find other items,

say other movies related to movie i,

then what you can do is try to find the item k with

features x^(k) that is similar to x^(i).

In particular, given a feature vector x^(k),

the way we determine what are known as

similar to the feature x^(i) is

as follows: is the sum from l equals 1 through n with

n features of x^(k)\_l minus x^(i)\_l square.

This turns out to be the squared distance between

x^(k) and x^(i) and in math,

this squared distance between these two vectors,

x^(k) and x^(i),

is sometimes written as follows as well.

If you find not just the one movie with

the smallest distance between

x^(k) and x^(i) but find say,

the five or 10 items with

the most similar feature vectors,

then you end up finding

five or 10 related items to the item x^(i).

If you're building a website and want to help users find

related products to

a specific product they are looking at,

this would be a nice way to do so because

the features x^(i) give a sense of what item i is about,

other items x^(k) with

similar features will turn out to be similar to item i.

It turns out later this week,

this idea of finding

related items will be a small building blocks that we'll

use to get to

an even more powerful recommended system as well.



Before wrapping up this section,

I want to mention

a few limitations of collaborative filtering.

In collaborative filtering, you

have a set of items and so

the users and the users have rated some subset of items.

One of this weaknesses is that is

not very good at the cold start problem.

For example, if there's a new item in your catalog,

say someone's just published a new movie

and hardly anyone has rated that movie yet,

how do you rank the new item

if very few users have rated it before?

Similarly, for new users

that have rated only a few items,

how can we make sure we show them something reasonable?

We could see in an earlier video,

how mean normalization can help

with this and it does help a lot.

But perhaps even better ways to

show users that rated very few items,

things that are likely to interest them.

This is called the cold start problem,

because when you have a new item,

there are few users have rated,

or we have a new user that's rated very few items,

the results of collaborative filtering for that item

or for that user may not be very accurate.

The second limitation of

collaborative filtering is it doesn't give you

a natural way to use side information

or additional information about items or users.

For example, for a given movie in your catalog,

you might know what is the genre of the movie,

who had a movie stars,

whether it is a studio,

what is the budget, and so on.

You may have a lot of features about a given movie.

For a single user,

you may know something about their demographics,

such as their age, gender, location.

They express preferences, such as if they tell

you they like certain movies genres

but not other movies genres,

or it turns out if you know the user's IP address,

that can tell you a lot about a user's location,

and knowing the user's location might also help

you guess what might the user be interested in,

or if you know whether the user is accessing

your site on a mobile or on a desktop,

or if you know what web browser they're using.

It turns out all of these are little cues you can get.

They can be surprisingly

correlated with the preferences of a user.

It turns out by the way, that it is known that users,

that use the Chrome versus Firefox versus

Safari versus the Microsoft Edge browser,

they actually behave in very different ways.

Even knowing the user web browser can give you

a hint when you have collected

enough data of what this particular user may like.

Even though collaborative filtering,

we have multiple users give

you ratings of multiple items,

is a very powerful set of algorithms,

it also has some limitations.

In the next video,

let's go on to

develop content-based filtering algorithms,

which can address a lot of these limitations.

Content-based filtering algorithms are

a steady art technique used in

many commercial applications today.

Let's go take a look at how they work.