Week 9
Video Transcripts

Video 1 (10:30): Collaborative Filtering Problem

In this lecture, we're going to move to another— our next set of unsupervised models called, matrix factorization models. Specifically, in this lecture we're going to discuss the collaborative filtering problem. The motivation for this type of model is the object recommendation problem. So of course, I don't need to convince you that matching customers to products is an important practical problem that many companies would be interested in being able to do well. And, we can often make these connections using user-provided information about these products. So, for example, Netflix allows users to rate movies— there's that data. Amazon has data, where they allow their users to rate their products, and also to write reviews about them. Yelp lets users rate businesses, also write reviews about them, up—upload pictures etc. YouTube allows users to like or dislike videos and to write comments about them, among many other different examples, and even within these examples, there are other— there's other information that we possess.

So, a recommendation system can be applied to any of these problems, where we want to somehow take all of this information that we have, and now match a user to an object to make a recommendation. Specifically, we want to say, this user has not given me any feedback on this product, but I'm going to assume that's because that user doesn't know anything about the product. So, do I want to match this user together with this product? What do I think this user would think about this product? Or, what do I think this person would rate this movie? And, I want to use that now to help make predictions and— about what a user is going to like and to make recommendations to that user. There are a few strategies to doing this. So, this is a big problem with many different approaches. One general subset of approaches can be referred to as content filtering. So, this not the focus of the lecture today, but to give an overview of this important approach to it, what content filtering does is it takes known information about the products and it takes known information about the users to create profiles of the two.

So, for example, with products we have movie, we have information about the movies. We have information about the price of the product. We have descriptions that are maybe provided by the producer of the product. And so, we extract information about the movie. We know it's an action movie. We know what year it was made, etc. etc. We get a profile of descriptions about this movie. We also then do the same for each user. So, we have a demographic information. They maybe fill out a questionnaire. So, let's look at one example. The online radio, Pandora uses something called the Music Genome Project to make recommendations, among other approaches I'm sure. And so what this does— this approach, which is a content filtering approach is to allow experts to score each song in the database, based on hundreds of characteristics. So, this—we could pick a song and we could say, this song has a lot of guitar in it and has a female singer and it's four minutes long and it's, you know, it's very loud. It has all of these descriptions that we assign to it, for example, by checking off boxes. Is this

song loud? Yes is a one. No is a zero. And, we have a long list of these descriptions. And then, a user will create a profile, and many questions will be asked of this user about his or her preferences, like, what types of music do you like? What are some examples of songs that you like?

And given all of this information, each user now can have a profile created for that person. And then, based on the profile of the user and the profile of the song, they're matched together to see how similar are—is the profile of each to each other? And in this way, recommendations can be made. So it's...it's an intuitive approach. If I say I like rock music and I know that this is a rock band, then perhaps I'll like that band. So, that's a way to do it. But notice that this approach is not using any behavioral information. It's not using any of my listening behavior. It's not letting me rate particular songs or particular products, or looking at my viewing behavior. It's not taking into consideration the actual patterns of how people are using these products etc. etc. So, that leads to the collaborative filtering approach.

So, collaborative filtering is a fundamentally different approach to this type of a problem. What collaborative filtering does is it takes users' inputs, for example, users' ratings of movies or users' viewing behaviors and then, it tries to make future recommendations based on the actual data that it's given, by the user. And it's going to ignore, any sort of a priori object or user-specific information. So it's not going to care, whether a user is young or old. It's not going to care, whether a song is a rock song or a classical song. It's only going to purely base its recommendations on what users give as feedback to a particular product. So, one example of this is a neighborhood-based approach, and this is a good intuitive explanation for the difference between collaborative and content filtering. So, for example, what we could do with a neighborhood-based approach is try to do define a similarity score between me as a user and every other user in my database. And a way to do that would be, for me and another person to ask how many objects— so let's talk about movies to make it concrete. How many movies have we both rated? So, what's the overlap of our ratings, in terms of the movies that we both have watched and rated? And then, given that overlap, how much do we agree? So, if I tend to agree with another person very much, then I'm going to be very close to that person. If I tend to disagree with that—a different person very much, then that's actually also good information. I want to take that into account. So I get a proximity between me and every other user. And then, based on those proximities, I'm going to let other users recommend things to me.

So, for example, if I agree with this person very much, and this person watched a certain movie and gave it a very high score, high rating, and I have not given any feedback to that movie, then I'm going to let that person recommend, you know, that movie to me by saying because we agree so much that I most likely would give this movie a high rating and so I should watch it. So, that's the idea behind neighborhood-based approaches. So, notice the thing that's going on here is that there's no background information. The objects—the same exact approach can be used for any sort of a recommendation. We could be recommending songs. We could be recommending books or movies. All that matters is the feedback that people have given to products. And then, we're trying to pair people with other people, in order to recommend objects to each other. And—so before I discuss the main collaborative filtering algorithm, I want to...I want to go through today, I should also quickly mention that these two approaches, the content filtering, and collaborative filtering approaches are not mutually exclusive. We can often use content information and build—take that information and build it into collaborative filtering models to improve the performance. So, both techniques actually can be combined. But in this

lecture, I just want to focus on one technique. Specifically, with location-based collaborative filtering methods, what we can—what we're going to discuss is a method for embedding objects in a latent space. So, for example, we'll take movies like Braveheart, Amadeus, and Lethal Weapon, and we'll give a 'd' dimensional location here for this simple plot— it's a two-dimensional location in the space, and so, all the movies will in this example live at a point in this two-dimensional space.
And then, we're also going to give a location to each of the users in our database, in the same exact space. And then for example, this person is close to Braveheart, and so, even though this person has not rated Braveheart or I have no information about what this person thinks about Braveheart, because I have learned that Braveheart is close to this person, I'm going to assume that this person would like that movie. So, these location-based methods that we're going to discuss today make recommendations based on how...how close things are to each other in that latent embedding space that we have to now learn from the data.

Video 2 (8:16): Matrix Factorization I

And the way that we're going to learn these embeddings is by matrix factorization. To see how we can use matrix factorization to do object recommendation in the collaborative filtering setting, first we need to know, what the data structure should look like. So, what we're going to assume that we have, is a matrix of ratings. So, we're thinking of ratings of users give to objects in this case. So, imagine that we have 'N_1' users. In that case, we will have 'N_1' rows in our matrix. And imagine that we have 'N_2' objects, so they could be 'N'— they could be movies or books or what have you. We have 'N_2' objects. Each object now has a column. And so we have 'N_2' columns in our matrix. We'll call this matrix 'M' And now the ijth entry in this matrix, 'M_{ij}', is going to be the rating that the ith user gave to the jth object.
So, this element in this matrix says that user 'i' rated object 'j', and then the number will say what the rating is. So, notice that in this case, this matrix has very many missing values because if each user can only rate a tiny fraction of the products, then there's going to be a very small fraction of values that we have in this matrix. So, what matrix factorization does, is it now tries to learn a low-rank factorization of this matrix, using only the data that we've observed, and ignoring the data that we don't have, and then fill in, all of the missing values in this matrix. And so, when we fill in the missing values, those are like predictions for what a user will rate an object, and then we can use those predictions to make recommendations, because if we predict that a user is going to rate something very highly, we're going to want to recommend that thing to the user. We've already seen an example of a matrix factorization, before. We didn't go into too much depth about it, but we discussed the singular value decomposition. There we saw that any matrix 'M' could be decomposed into a product of three matrices. So, in that case, the matrix 'M' had no missing values in it. It was 'n' by 'd'. And, we could write this matrix 'M' as a product of a matrix 'U' that has orthonormal columns, so the dot product 'U' transpose 'U' is equal to the identity matrix, times a diagonal matrix 'S' that has zeros on the off-diagonal and non-negative values on the diagonal, times another orthonormal matrix 'V' transpose, meaning that 'V' transpose 'V' is equal to the identity.

So, we can always do this. What we didn't really discuss in detail is that when this— if 'M' is a low-rank matrix, what that means is that these— this decomposition, for example, if 'M' has—is rank 'r', this decomposition will have only 'r' columns. So, if we can write 'M' as a product of low-rank matrices, then we don't have an 'n' by 'n' matrix here. We have an 'n' by 'r' matrix where 'r' is less than 'n'. So, when the rank 'r' is smaller than the minimum of the number of rows or columns, when it's much smaller, then in a sense what that's saying is that the matrix has fewer degrees of freedom. So, a low-rank matrix factorization can restrict the degrees of freedom of how many different types of things we can model in the matrix 'M'. So, this is going to be what we exploit— this idea of a low-rank matrix factorization in the collaborative filtering problem. So specifically, what we want to do is we want to say that our ratings matrix, so this is from before where we have 'N_1' users and 'N_2' objects and the ijth entry in that matrix, says what user 'i' rated object 'j'. So, this matrix which now has a lot of missing values, is approximately equal to a low-rank matrix, a matrix product with rank 'd'. What that means is that we can take this matrix and represent it as a product of a matrix that is 'N_1' by 'd', times a matrix that is 'd' by 'N_2', and now 'd' is going to be much much less than 'N_1' and 'N_2'.

So, if you imagine that we have hundreds of thousands of users and tens of thousands of objects, this will be a huge matrix. But 'd' is going to be something much smaller. So, it will be something on the order of '10', that will say how many degrees of freedom we really have in this matrix. And intuitively, you could almost think of it as saying, how many things are really taken into consideration when giving a movie a rating. If 'd' is equal to '10', then that's like saying there are '10'—'10' underlying things that factor into a particular person's rating of a particular object. But, we aren't going to define the meaning of what those are. It's a latent sort of thing. Okay! So, we're going to assume this sort of a factorization, and we're going to call this matrix 'U' and this matrix 'V'. And now, notice that the ijth entry in this matrix can be viewed as the ith row of the left matrix, times the ith— jth column of the right matrix.

So, this is the rules that you have with matrix products that if you look at the entry 'ij' here, that's approximately... approximately going to be equal to the ith row of this matrix, times the ith— jth column of this right matrix. And so, now our goal is to learn these two matrices. And, notice that if we restrict our view to the ith user, then what we're doing is restricting our view to the ith row of this matrix. And if we want to know what are the ratings that the ith user gives to all of the objects, we're going to take the ith row of this left matrix, call that 'Ui', so this is a 'd' dimensional vector that's on the ith row of the left matrix, and then we take a dot product of this ith vector with any particular column in the right matrix, in order to predict what the ith user will rate that product. And similarly, if we want to focus on a particular product like the jth product, and we want to look across users in this way, then that—in that case what we're doing is we're taking the jth column of the right matrix, call that 'V_j', that's a also a 'd' dimensional vector. We can view this like the location of the 'g'—the jth object. And then we take the dot product of that vector with any particular row, according to which user that we want to query. And so, these are the locations that we're going to give. The ith user has a location in dimension 'r', in R^d that's equal to the vector 'U_i', which is the ith row of this matrix. And the jth object has a location in R^d, which is equal to the jth column of this right matrix.

Video 9-3 (8:06): Matrix Factorization II

Now the question is why does low-rank work, what do we gain by being low-rank. I'm just going to go through some of the intuitions here, about why we would want to do a low-rank matrix factorization. To see this, imagine that we have two movies, we have the movie Caddyshack, and we have the movie Animal House. And so the Caddyshack—the movie Caddyshack corresponds to this column, so this is the column for Caddyshack, this is the column for the movie Animal House, and every other column corresponds to some other movie. What this column now is going to have is all of the ratings that any user has given to Caddyshack. So, if there are 'N_1' users, a subset of those users have watched and rated Caddyshack and their ratings are going to be in particular rows of this column vector. And we have the same exact thing with Animal House, we have 'N_1' users, a fraction of them have seen this movie, and rated it, and their ratings also fall along the corresponding rows of this column. And we can imagine that some of those rows are going to overlap, the same user will have watched and rated both movies, but also some users watched one movie and rated it while not watching the other and vice versa.
Now, if you know about these movies, you know that they're very similar, they're from the same period of time, they're both the same type of comedy movie. And you can imagine, although it's not necessarily always the case, you can imagine that if somebody really likes the movie Caddyshack, they're also going to really like the movie Animal House. So I tried to pick an example that everybody would agree with. And therefore, if a person who rates Caddyshack highly, they're also most likely going to rate Animal House highly. On the flipside, if a person hates one of the movies, they're also very likely going to hate the other movie. And so, we imagine the ratings along these two columns are very highly correlated. They look very similar to each other, that's what we imagine. And if, you don't like this example you could imagine a different example of two other movies that have high correlation in their ratings.
And so what low-rank factorizations do is learn, and they strictly enforce this type of a correlation. So, if we look at the column for Caddyshack, and Animal House, and we have a rank defactorization, what we're saying is that this column is equal to this 'd' dimensional vector, times this matrix, whereas this column is approximately equal to this 'd' dimensional vector, times this matrix. And so, in that sense, because we are restricting all of the objects and users to live in this 'd' dimensional space, we're restricting what types of matrices that we can learn and we're forcing there to be correlations. And so because, we're only going to have a very tiny fraction of the ratings in this matrix, by enforcing this type of a low-correlation—low-rank, high-correlation factorization, we're going to be able to borrow information across movies in order to learn these locations. So, we're going to be able to borrow information across the users that rated Caddyshack, in order to help predict what those users would rate for Animal House, and vice versa. And we're going to actually take into consider all of the correlations among all of the movies in making these predictions. The technique we're going to discuss in this lecture is called probabilistic matrix factorization. This is one particular model for learning a low-rank factorization in the missing data problem. I want to add a few bits of notation, for this model. So, again here's our data matrix. We have 'N_1' users, 'N_2' objects. The ij^{th} entry of this matrix is the rating of user 'i' for object 'j'. We'll call this matrix 'M'. And now again, this matrix has very many missing values. So, what we're going to do is, let the set Omega contain the pairs of all of the ratings that we have.

So the pair 'ij' is going to be in the set Omega, if 'M_{ij}' is measured. So this set, now has all of the measured index pairs in this—from this matrix. And let's, let '$Omega_{ui}$' be the index of the set of all objects rated by user 'i'. And let's, let '$Omega_{vj}$' be the index set of all the users who rated object 'j'. So this is just for notation, so we can have a cleaner way of writing this later. What probabilistic matrix factorization does, what this particular matrix does is assume a generative model, which is why it's called probabilistic. It assumes a generative model for the data that goes, like this. So, for each of the 'N_1' users, we assume that their location in R^d is going to be generated from a zero mean Gaussian with some covariance. And for each of the 'N_2' objects, we're also going to assume that their location is generated $_{iid}$ from the same Gaussian. So, this is just a simple case of what types of priors we can use. So, these are the priors that we define on the respective locations of all the users and all of the objects in R^d. And then given these locations, what we want to generate, the rating, the value for 'M*ij*' for each pair in our measured set. What the model assumes, is that this rating is a Gaussian random variable, with mean equal to the dot product between the user location and the object location that we're generating the rating for, and some variance. So, this is the data level distribution, and this is the prior on all of the model variables. So, there are a couple of comments about this model. First, notice that since 'M_{ij}' is going to be assumed to be a rating matrix, in cases where it is a rating matrix, it's clear that this Gaussian assumption is wrong, because the ratings are discrete, they take values one, two, three, four, five. Whereas this Gaussian is a distribution on continuous-valued, random variables that can take any value.

So, this is a bad model definition, in the sense of, being defined on the support of our data. However in practice it's going to work very well, and it's also been refined and developed. I just wanted to present the original model to you here. In practice, even though we make a bad assumption about the distribution on our data, it's going to work very well because even though the data can only take one of say five or '10' values, they're still ordinal, meaning that order matters, one, two, you know the relationship of the rating one to two and two to three in a sense means the same, one is—one and two are the same distance as two to three, and one, and three is twice the distance, so intuitively that makes sense. And so this Gaussian is going to be able to still model that relationship correctly.

Video 4 (8:24): Model Inference

The next step now is to infer the model. As usual, the story goes that we propose a generative model, a probabilistic model on our data. That was the probabilistic matrix factorization model of the previous slide. Now, the problem is that we have some data. We want to do the inverse problem of inferring the model parameters from the data. So, this might lead to a question, in the context of what we've been discussing in the previous few lectures, we have a matrix 'M', and it has many missing values in the matrix. And we want to predict what the missing values are. Two lectures ago, we discussed exactly that problem of predicting...of doing inference when we have missing values, and we saw there that we needed to do an EM, or EM was going to help us out. So, now the question is do we need some sort of an EM algorithm in this scenario? And so, let's think through this problem. Let's let 'M_o' be the observed part of 'M'. And 'M' subscript 'm' be the missing part, so using the similar notation as two lectures ago.

Then, what we want to do is we want to maximize the likelihood in a sense of the observations that we've seen given, 'U' and 'V'. So I'm— we also have a prior here that I'm ignoring for a moment, and that's equal to the integral of the—over the missing values of the complete matrix, given 'U' and 'V'. So, the question now is whether we need to use an EM algorithm, where we introduce this missing data to the model. To answer this question, let's recall that EM is simply a tool for maximizing a marginal likelihood, where there's no missing information. Therefore, we only need it when we have the situation that this is hard to maximize, but this is somehow easy to maximize, and also the posterior is known. So, if we can maximize this term without any problems, then there's no need and really no point in doing EM. If we can get close form updates, for example for this— for doing maximum likelihood, or in our case, it's going to be Maximum A Posteriori over 'U' and 'V' of this likelihood, then EM doesn't buy us anything. So, we have to look at what does this likelihood look like? So, to test how hard it's going to be to maximize the joint likelihood of the observations of the user locations and the object locations over 'U' and 'V', what we have to do is do the following three steps. We have to write out the joint likelihood. We then need to take its natural logarithm. And then, we need to take derivatives with respect to all the things that we want to update. So, we want to update 'u_i' and we want to update 'v_j', the user location for user 'i' and the object location for object 'j' over all the users and all the objects. We want to take these derivatives and see if we can solve them. And if we can, then we can do that without having to introduce EM. For the probabilistic matrix factorization model, we notice that what we can do is write this joint likelihood in this way. We can write a product over all of the observed ratings that we have in our data set, of the rating that user 'i' gives to object 'j', given the user location and the object location for that rating, times the priors on those locations. So, remember that the user locations were all v_{iid} from a distribution, as were the object locations, and so the prior factorizes into this product over the individual priors. And so by definition, we can write out each of these. This is a Gaussian and these are all also Gaussian. So, we take this likelihood, we take its log, and we get the objective now that we're trying to maximize. So, what we want to do when we do MAP inference, Maximum A Posteriori, is to find the values for the matrix 'U' and 'V' that maximizes the log of the joint likelihood. And in this case, we get this form, where we sum over all of the measurements in our matrix 'M', the log of the likelihood of each individual measurement, given the locations for the user and object, plus the sum over all of the logs of the priors. So, we'll plug in what these distributions are. Call this objective 'L'. And what we find is that 'L' is equal to this. So we have negative, a sum of a squared error term, minus these additional regularizations. So, this is the log of the joint likelihood, when we plug in, and these square terms all come, because these are Gaussian distributions. Remember this is a Gaussian distribution, where we assume that this was generated from Gaussian with this mean, and so that's how this form comes about. And these are all independent Gaussian priors with zero mean. And, so that's where these terms come from, 'ui' has zero mean, 'v_j' has zero mean prior. Now, we want to see whether we can actually update these parameters. Whether we can update the vector 'u_i' and 'v_j' by taking the derivative and setting to zero.

So, we take the derivative with respect to 'u_i', and we find that what we get is a sum over all objects that user 'i' has rated. So, remember this notation. We sum over all of the objects that are in the index set of the user 'i's' ratings of this term, minus this additional prior on user 'i'. Similarly for object 'j', we sum over all of the users who have rated that object of the derivative of the first term with respect to 'v_j'. So,

we take the derivative of this term with respect to 'v_j'. We end up with this term. We take the derivative with respect to the prior. We end up with this term, and we set them all to zero, and we see whether we can solve. And it turns out that for this problem, because of the specific model structure that we assumed, we can solve all of these things. And so, the user 'i's' location is—can be updated using this equation, and object 'j's' location can be updated in this—using this equation. So, notice that what we have here is an outer product for user 'i'. We have a sum of the outer product of all of the locations of the objects that user 'i' has rated plus this additional term. We take that matrix and invert it, and we multiply it by the vector we get by taking the location of each object that user 'i' has rated and multiplying it by the rating, and summing those up. And that's the...thats's the update for 'u_i' that maximizes the objective. However, notice that if we wanted to solve for all 'u_i's' and all 'j'—'v 's' simultaneously, we would not be able to do that. Because the update for 'u_i' involves 'v', and the update for 'v_j' involves 'u'. So, we can't say that this value for 'u_i', or this value for 'v_j' is the optimal one, because it depends on the other parameter settings that we have. So, what we're going to end up having is a coordinate ascent algorithm, like we've been discussing for the previous few lectures.

Video 5 (8:34): Probabilistic Matrix Factorization I

So here's the coordinate ascent algorithm for doing MAP inference for the probabilistic matrix factorization model. What we have as an input is an incomplete ratings matrix 'M', where the measurements in 'M' are indexed. The locations of the measurements are indexed by Omega. And we also input a rank 'd' that we want to learn. So, we input the dimensionality of this latent space that we want to embed all of these users and objects into. The output is going to be 'N_1' user locations where each user is located in R^d, and 'N_2' object locations where each object is also located in R^d, so the same space. First, we're going to initialize each of the object locations. So, we could have also initialized the user locations first, and then flipped the algorithm. I've chosen in this slide to first initialize the object locations.

For example, we could just generate them randomly, then for each iteration we follow the two coordinate ascent steps. First, within a particular iteration we update each of the 'N_1' user locations by calculating-by maximizing the objective over each user location, given the current values of all of the objects, all of the locations of all of the objects. So, we solve this equation plugging in the current locations of the objects and using the data here. Then the second half of the same iteration will be to update the locations for all of the objects. And so for each value 'j', we update the location for object 'j' solving this equation, where we use the relevant locations of the users. We use all of the users who have rated that object and also all of their ratings here. So, we iterate back and forth between these two steps, and eventually, the algorithm will converge. We can assess this convergence by calculating the log of the joint likelihood. That's the function we're trying to maximize, so we can evaluate that function after each iteration to see if it's converged or not. And then, when it's converged, if we want to make a prediction for particular user 'i' and a particular object 'j' that we don't have in the matrix 'M', we'll predict the corresponding element in 'M' to be the dot product between user 'i's' location and object 'j's' location, and then we can round it to the closest rating option. And that's how we're going to be able to

make a prediction for any user and any object that we don't know a rating for. We can pick out the user's location. We can pick out the object's location. And we just take their dot to the nearest legitimate possible rating. And that gives us our rating, our prediction of their rating.
So, we get some sort of embedding like this. This is a simplified example, where we have the data living in R^2. In practice, the data would most likely be in a higher-dimensional space like R^{10} or something like that. And for every movie, we have a location and also for every user we have a location, and so we can use this to visualize or to understand what's going on. For example, we can take a particular movie. This plot is not necessarily the best one to consider, but we can take a particular movie and we can say what are the '10' closest movies to this movie, and what we should see is that that list makes sense. So, if you picked Star Wars, most likely the closest movie to Star Wars will be Empire Strikes Back, something like that. Now, if we return to the original example that I showed, it's easy to understand why this latent space embedding of users, and objects should end up being interpretable, where proximities between things will be meaningful. For example, if we look at the movies Caddyshack and Animal House, and we agree a priori that they're very similar and users should rate them similarly whether they like them or not, we then have that this column is the column of the ratings that were given to the movie Caddyshack, and this is the column of ratings given to the movie Animal House across all of the users. So again, we only have some of the measurements in here and here. We've run the probabilistic matrix factorization algorithm to take this missing data matrix. This matrix with many missing values and factorize it into a product of this matrix, times this matrix, where the left matrix is 'N'-'N_1' by 'd' and the i^{th} row corresponds to the location for the i^{th} user, and the right matrix is 'd' by 'N_2', where for example, this column corresponds to the movie Caddyshack.
So the i^{th}-the j^{th} column here, corresponds to the j^{th} column here, and this column corresponds to the location for Animal House. So, if I want to know what this column's equal to, it's about equal to this, times this, and similarly this column's approximately, this matrix, times this column. Now, if we assume that these two movies should have similar rating patterns, so these two vectors should be very close to each other in a very high 'N_1' dimensional space. What we're saying is that this vector is equal to this matrix, times this small vector. And this vector is equal to the same exact matrix, times this vector. So, if these two columns look alike, if they are very close to each other, then that means that these two columns also have to be very close to each other. Because if the values in these two columns differ significantly, then the product of this vector with this matrix will be very different from the product of this vector with this matrix. And so, if these two columns are highly correlated, in the matrix factorization language, that means that these two much smaller vectors are very close- should be very close to each other. And whatever error there is, is either due to the Gaussian likelihood, which absorbs a lot of the error, or...or corresponds to the slight variation in the values of these two vectors. And so we can intuitively make this argument for any pair of movies and also for any two users, if I picked two users and they have very similar ratings then their corresponding rows of ratings will be equal to the corresponding rows of these vectors, times the entire matrix here on the right. And again, those two users should be located very close to each other in that space.
Therefore, if I want to for example, pick two users and I have one user viewed Caddyshack and loved it but didn't view Animal House, and then another user viewed both movies and liked both of them a lot, and they also agree on many other movies, then those two users are going to be very similar to each

other in this matrix. And then when you look at their dot products, that fact translates to the user who didn't rate one of the movies giving a similar rating to the movie, as the other user who did rate the movies. So, that's where this idea of collaborative filtering comes in that users can, kind of, influence each other's ratings and give information about what other people will rate based on what they liked.

Video 6 (5:40): Probabilistic Matrix Factorization II

To see the relationship between matrix factorization and ridge regression, let's focus on the update for one specific movie. So, in this case, we have the ratings matrix, that's a incomplete matrix with a lot of missing values. We focus on one particular object, say the j^{th} object, which corresponds to this column. If we only have these particular ratings for that object that means we have points here, at—where you see a black dot. Those are the points where we have ratings, and every other row has no data there. Then, we can view the ratings that we have in these dimensions as being approximately equal to the corresponding rows of this matrix. So, this row corresponds to this row, times this vector in the j^{th} column. And now we can take this matrix factorization problem, restrict it to the j^{th} object, and view this as a sub-problem, like this, where we take the ratings that we have, make a smaller vector. So, I've taken these values and put them in a smaller vector, and we're saying that those values are approximately equal to the corresponding rows of this matrix, which I now take out and put into this full matrix here, times this column, which I put here.

And now when I update the location for object 'j', what I want to do is actually solve a miniature problem that looks like this. And so, let's remember in the ridge regression setting, what we did. What we did there was we minimized the sum of the squared errors of this approximation to this data—vector. In that sense, you could also think of it like minimizing the sum of squared errors of the corresponding value 'M_{ij}' in here, minus the corresponding vector 'u_i', which is the row here, times the vector 'v_j'. We summed those up, and then we added an 'L''2' penalty in the ridge regression problem to the magnitude of 'v_j'. If we view this little miniature problem as a ridge regression problem, what we'll find is that we get this sort of an update as well. So, this was actually the ridge regression update. This matrix—this update fo the vector 'v_j' corresponds to a ridge regression update. The equation is the same, where for any particular movie—user 'i' that rated object 'j', we're using the user's location as the covariate vector for that rating. And then, 'M_{ij}' corresponds to the output that we want to now predict, for that particular rating. So, in ridge regression, remember we actually had these outputs and we had the inputs as well. We wanted to predict an output, based on the input covariate vector.

In this case, those corresponding covariates are themselves unknown and we want to learn them. So, we can view the matrix factorization as a sequence of 'N_1' plus 'N_2' different ridge regression problems that we solve, iteratively. We can also think of this as a least squares problem, if we want to remove the Gaussian priors, in which case if we view this as a least squares problem, we have this update. All we do is remove this additional regularization term down here and we get this sort of an update, and so that would correspond to doing maximum likelihood for this problem. So, this sequence of updates for each column in each row corresponds to maximizing the likelihood of the observed values, given 'U' and given 'V' instead of maximizing the joint likelihood. However, notice that in this case if we did not put priors on

'U' and 'V', and simply to maximum likelihood, in order to invert this matrix we would require that every single user has rated at least 'd' different objects, and similarly we would require that every single object has been rated by at least 'd' different users, in order for us to guarantee that these inverse—this inverse can be done. So, for this problem, it's most likely that that's not the case, that we have if—that every user has rated more than 'd' objects and every object has been rated more than 'd' times, especially if a user is new or an object is new. And so we can kind of see here, where the Bayesian approach is actually necessary for this model. Previously, it wasn't necessary—it wasn't required in order to make progress.

For this type of a model, we need the additional regularization represented by this additional matrix here, which corresponds to a Gaussian prior in order to make sure that we can actually invert this matrix for every single user, and therefore that the algorithm won't crash because we have a non-invertible matrix.

Video 7 (5:25): Topic Modeling

The motivation for models like, Latent Dirichlet Allocation is to model text. So in these cases, we have different documents coming into us that have—each have different words in it. And we want to take these documents and organize them, so that we group documents that discuss the similar things into similar clusters for example, or we would want to summarize or annotate these documents, learn what's the information underlying them, what are the themes that these different documents discuss for a variety of different tasks, either information retrieval, recommendation, prediction etc. So, for example, the idea behind topic modeling is that different documents contain words in them that belong to different thematic concepts. So, for example, if we looked at a document called, A Digital Shift on Health Data Swells Profits in an Industry, we have a document that discusses how the government is thinking about taking documents, health records that are written on paper and digitizing them.

And so, we have words that discuss a variety of different topics. We have words like government, president, law, legislation and bill, and lobbying that discuss—that are all kind of revolving around a political theme. We also have words that have to do with medicine like, doctors, healthcare, medical, and hospitals that revolves around the fact that this has to do with digitizing health records. Because we're digitizing them, we have things like electronic systems and digital, which have to do with technology. And finally, because it's a lucrative, potentially lucrative business, we have words like business, economic, and companies that have to do with business. So, there's one document expressing many different themes, all of them coherently put together to describe what is going on in this document. So, the goal of a topic model, is to model all of this information. And what we assume, starting out, is that we have a set of probability distributions on a fixed vocabulary, and each of these probability distributions have all of their probability or almost all of their probability on a subset of the words that all somehow belong together and coherently express a theme.

So here's a...a toy example that I created, where we might have these five underlying topics, where the first topic has its four most probable words as health, medical, disease, and hospital. And, the second topic might have its most—four most probable words team, basketball, points, and score, and so on.

And if you look at any of these five lists, every single word in the vocabulary appears on it. So, the word 'law', appears in this sports-related list, except its probability is so small that when we order them based on probability, it appears very low in this list. So, this is the idea were each of these is a probability distribution on the exact same set of words. But, if we view that probability distribution as an ordered list of words based on how probable they are, we'll see that the most probable words that come to the top, all relate to the same theme. So, we have the—so we learn five different topics, where each topic captures one theme underlying the data set. So, this is a global thing that interacts with every document. Then for one particular document, for example the one we previously saw, we have two additional things that go into making a topic model. We have first, a vector of topic proportions. So, for example, if we have five topics then, this will be a five-dimensional probability distribution on those five topics, where in this case, you can see that the four topics that appear in this document have significant probability, and the sports topic has essentially no probability for this document. So, that's the first document-level variable. And then, we also have for each word that appears in a document, an assignment of which of the topics it belongs to.
So, for example for this word 'government', we color-code it yellow because it belongs to the government topic, which is the yellow topic. And we have this color coding for every single word that appears in the document saying that each word has to pick a topic that it's going to come from in this document, and it's going to be colored according to that topic. But of course, we don't have any of this information, all we have are the words contained in the documents and so we need to learn it. And that's the inference goal of a model like LDA, which is a topic—one of the most famous topic models.

Video 8 (13:20): Latent Dirichlet Allocation I

So my goal in this lecture is, to present, the generative process for LDA the way of generating documents, according to the model LDA and then, move onto non-negative matrix factorization, which is a more general technique for factorizing matrices, and discuss two algorithms for doing that, that are relevant to the same problems that LDA addresses, but are not equivalent to the algorithm that is used for LDA generally. So, there are two essential ingredients to LDA, and essentially any topic model. Again, there's a collection of distributions on words, which we call topics. And then, we also have a distribution, on topics for each document. So, here I'm just giving another view of what I previously discussed, where, if you imagine that we have these words constituting our vocabulary, so this is a toy example. In general, our vocabulary that we're going to use to model these documents will be thousands or tens of thousands of words. But now, we're just focusing on these, and saying this is the vocabulary, that exists for this problem. Then, if we show three different topics, for example, if we have up here, seven topics, and we want to pick one, for example, the politics topic, and show the distribution on the words and the vocabulary according to this topic. If we call this, the third topic, so we have a vector, '$Beta_3$', which is a probability distribution, on this set of words.
Then, the high probable words would be things like, vote, politics, power, and senate, etc. If we then focused on a different topic, for example, the sports topic that will be represented by probability distribution on exactly the same words, except its high probability words will be things like, ball, score,

and seasons, etc. And also with an additional topic, for example, a logic topic. And so each of these topics, and we could have many of them, is going to be, characterized by probability distribution, on the same exact set of vocabulary words. But, they're going to have different probability distributions. Then what we have, is for a particular document, we have a distribution on these topics. So, for example, if one person sits down and writes a topic, what the model hypothesizes, is that, that person will pick a probability distribution on these topics. So, we'll call that for the first document, call that 'Theta$_1$', and perhaps the highly probable words, the highly probable topics, will be childhood and economics. And so, most of the words picked for this document are going to come from these two probability distributions. And another person might want to write about something else. And a third person about a different topic, a different subject, which will be related through using words coming from these four different topics, according to the proportion of this— of the distribution on those topics. So, the way that we represent the model, like LDA, which is a Bayesian model, is the same as the way we represent any other Bayesian model, through a generative process. So, we hypothesize a generative process, for the data that we see. And then, have to derive an inference algorithm for doing the inverse problem of learning the actual parameters, for a posterior distribution on those parameters of the model that could explain the data that we saw. So, for LDA, we have this generative process. We assume that, we have, 'K' different topics underlying our data set, which means we have 'k' different probability distributions, on the same set of words, and each of these distributions should capture a theme, by putting its probability mass on a coherent subset of the words. The prior model assumes that each of those topics is generated in—independently and identically distributed as a Dirichlet distribution. So, this is the prior distribution that's placed on each, of the 'k' topics. They're all generated, each topic is generated once by drawing from this distribution, and then fixed for all time. Then for each document, so each individual document, we need to decide, for that document, how it's going to use the topics, that are available to it.
So, for the d_{th} document that is done, by generating a 'k' dimensional probability distribution. So, this is a distribution on the 'k' different topics up here. So, this is 'k' dimensional. It's a probability distribution, that's generated *iid* from this Dirichlet distribution. And, we do that once for each of the capital 'D' documents in our data set. Now that we have for the d^{th} document a distribution on the different themes that Beta captures, we have to generate the words, that appear in that document. So, that's done according to this third step, for the n^{th} word, in the d^{th} document. So this is—we're focusing on the d^{th} document and looking at the n^{th} generated word.We first have to decide, which topic that word is going to come from. And that's done by generating a topic indicator, call it 'c_{dn}', from a discrete distribution, using the distribution on topics, for the d^{th} document. So 'c_{dn}', will pick out one of the capital 'K', topics available to it, where the probability of picking a particular topic is encoded in this distribution vector, 'Theta*d*'. Once it's picked out, its topic, the actual word is finally then generated, and so this is the data level, finally, that we arrive at, where, we actually see the data, the d^{th} word, the n^{th} word, in the d^{th} document, which is generated, also from a discrete distribution, where it uses, the topic picked out by '$c_{dn'}$. So 'c', subscript 'dn' is an index. It's a number between one and 'K', that picks out the index of the topic to use to generate the data. And so, that's why, we're subscripting with 'C_{dn}', here, because this is going to now pick out the correct index, for that word, in that document. So, a few things to notice, first, we don't know what any of these are, exceptfor the data 'x'. Wedon't know what 'c' is for each word and each document.

So, there are, many of these indicators, we have to learn. We don't know what the distributions on topics are, and we also don't know what the topics themselves are. So, there's many, there are many different things we need to learn with this model. The other thing to notice is that, this is what's called a bag-of-words model. So we're not modeling with LDA, we're not modeling language, as it's, as it appears in order. We're simply taking, all of the words, that appear in the document, and essentially throwing them in a bag. And then jumbling them up, and modeling the entire set of words, without any reference to order. So, notice that there's n order, taken into consideration, in this generative process. What this means is that, if you actually tried to, generate a document from this model, you would get nonsense. It's not going to make any linguistic sense. But it's going, but it is good enough in practice, to pick out the underlying themes in the document. The underlying topics, and also how that document uses those topics. So, it's able to get the macroscopic view, of what is the information, that appears in this set of documents. But the modeling power, is not fine enough, to actually model language as it's used in the documents. So, of course, there are many of those models, that have been developed, to take of those flaws, or potential flaws. But we aren't going to discuss those in this course. Another question is what is going on with this Dirichlet distribution? So, we haven't really discussed the Dirichlet distribution in this course. What a Dirichlet distribution is, is a probability distribution on vectors, that are non-negative and sum to one. So, if we let, 'Beta_k' be a random variable generated, from a Dirichlet distribution, what that means is, that 'Beta_k', is a vector of non-negative numbers that sum to one. So, we can view this as being a discrete probability distribution.

So, in that sense, the Dirichlet distribution, is a probability distribution on discrete distributions, when we need to put a prior, on a 'k', or a 'v' dimensional probability distribution, the Dirichlet, is a natural choice. However, the Dirichlet itself is a continuous probability distribution. So, there are some of these things, that can be a bit confusing that you need to think through, about what all the moving parts are. Dirichlet distribution is a continuous distribution on what can be viewed as a... a discrete probability distribution. The other question is, how the parameter of the Dirichlet distribution— what assumptions different values of this parameter are enforcing on the distribution. And so that's what, I'm showing in this, in the following few slides. For example, if we assume, that the parameter, to the Dirichlet is shared, so, even though we have an index, we have a 'v' dimensional probability distribution in Beta. And so we need a 'v' dimensional parameter vector in Gamma. If we assume that, all 'v' values, in that parameter vector are the same, we can then look at what different draws, from this distribution, will look like. So, down here, what I'm showing, is a case where, Beta is a '10' dimensional vector. And so, I have '10' different values in the 'x' dimension here. And Gamma, is a vector of also '10' dimensions, with the same value, in each element. So I'm going to just say, what that value is here. So Gamma, is a vector of '10' ones, in this case. And then, I generate '10' different random vectors, from this distribution, and I will get something that looks like this. So, these are '10' different random variables, all generated from a Dirichlet distribution, with parameter Gamma equal to one. And so you can see that, I get '10' different, '10' different discrete probability distributions. They're all non-negative, and they'll all sum to one. And they look something like this. Now as I let Gamma increase, you notice, from '10' to '100', you notice that, the random variables, that I generate start to look more and more uniform.

And so that's what the parameter does, it enforces a more and more uniform distribution, as this parameter goes to infinity. And then, if I go back in the other direction, and let it drop from one down to

zero, I get something that looks like this. I get more and more discrete distributions. So, from one to '0.1' to '.01' I'm sorry, not more discrete distributions. I get more, I get distributions that are, put most of its probability mass on a smaller and smaller subset of these dimensions. So, in practice with Dirichlet, with latent Dirichlet allocation, these parameters will be set to something less than one, which enforces our apriori belief, that each topic should only put its probability mass on a small subset of, the total number of words available to it. So, for example, the sports topic would put all of its mass on the sports-related words, which should be a small fraction of the total number of words in the vocabulary. And similarly, for the... the prior distribution on, the topic proportions, is Dirichlet as well. And so, if we, let Alpha be, something that's less than one, that enforces our apriori belief that, each document only uses a subset of the different topics available to it. So, there are '30' different things a document could talk about, represented by '30' different topics, represented by '30' different probability vectors Beta, we might set the prior Alpha, such that, only five or on expectation only five, for example, would be manifested in any individual document. But of course because it's a random vector, we can have more or less than that number

Video 9 (7:29): Latent Dirichlet Allocation II

And so we won't be able to discuss, the actual inference algorithm in this course, for latent Dirichlet allocation, because it uses something called, typically uses something called, variational inference, which is not an inference technique, that we'll be discussing. I do want to show an example, of what LDA does learn on real data. In this case, a set of a couple of million, a few million documents from the New York Times, over a '20' year period, what I'm showing here is those, those two million documents, were put into the LDA, we applied LDA to those documents, we learned the topics, and I'm showing '10' of those topics here. There were more in the model, but these are '10' of the topics that were learned. So, each of these lists corresponds to one vector, Beta, over, I believe, '8,000' word vocabulary, So, each of these lists has '8,000' words in it, in principle. But we're only showing, the '10' most probable words according to any individual Beta. So, if we called this '$Beta_1$', it's an '8,000' dimensional probability vector, on the '8,000' vocabulary words, And these are the '10' most probable words, according to that vocabulary, that vector '$Beta_1$'. And similar, if we call this, '$Beta_2$', '$Beta_3$', and '$Beta_4$' and so on.
Each of these represents one probability distribution on the same exact set of words. But the '10' most probable words changes, based on which vector we look at, and you can do a quick scan to realize that each of these distributions is capturing a coherent theme or topic that could appear in these documents. And, this is not something that's encoded in the model at all. This is something that's picked out, in the posterior distribution, because that information's contained in the data. And similarly, for every single document in the data set, we learned a probability distribution on these topics. So, there would be more topics, than just this. But for a particular document, we could have, a probability of this topic appearing in that document. And this one etc. And that, and that way we can compactly represent each document by a probability vector on the 'K' different topics in the model, which says these are the themes, that appear in this document, and also, these are the themes, that don't appear in this document, according

to that probability vector. I'm starting with LDA because, it's a very popular, and useful model for, machine learning, and with documents, and also other discrete group data.
So, I want to cover it. But I'm covering it now, because, LDA is very closely related to matrix factorization. And so, the way that we can see this is by asking for a particular document, what is the probability, that the n^{th} word in the d^{th} document is equal to the index 'i'. So, we have a vocabulary list and whichever word is indexed by 'i', that's what we're querying here, given only the set of topics and given only the distribution on those topics for the d^{th} document. And so, notice that there's no indicator here of which of the topics this word came from, 'c' is gone. In other words, this is the marginal distribution, where we've integrated out 'c'. And so we'll write that here and see what, if we can calculate that.So, here is the, the same probability we're asking for. And we write that as a sum over the joint distribution of the n^{th} word in the d^{th} document being equal to 'i'. And the assignment of the n^{th} word in the d^{th} document, to cluster or topic 'k'. And now we have a joint distribution, over this word index, and also this topic assignment for that word. And now we sum out the topic assignment. And so, of course, we can, that's integral or that sum is equal to the marginal distribution we're interested in. However, if we keep it this way, and then we break this down, this joint distribution over these two things, into a conditional distribution of 'x' given 'c', and given Beta and of course because we have 'c', we don't need to know Theta anymore. So 'x' is conditionally independent of Theta given 'c', that's why 'c', Theta doesn't appear here, then times the prior on the topic assignment, given the vector Theta. So I've broken this joint distribution, into this likelihood times this prior on 'c'. Then, we can read off what each of these are, given Beta and given Theta. The probability of a word, of the word, 'i' given that, that word comes from topic 'k', is just the equal to, the i^{th} dimension of topic vector, '$Beta_k$'. So, we can read off the probability from Beta, given these two bits of information. And also, the prior probability of any word in the d^{th} document coming from topic 'k' is just equal to the k^{th} dimension of the d^{th} probability vector, '$Theta_d$'. So, we have '$Theta_{dk}$'.
And so, we can replace these probabilities, with these two values here. And now finally, let's let capital 'B', be the matrix that we get, it's 'v' for 'v' vocabulary words by 'K' for 'K' topics. So, it's 'v' by 'k' matrix that we get by taking each of these topics, and putting them along the column. One column is one topic. And let's also construct this matrix Theta, capital Theta which is the 'K' by 'D' matrix that we get, by taking each distribution on topics and putting them along a column. So, '$Theta_d$' appears a long the d^{th} column, in the d^{th} column of this matrix then, we can also read off this probability by just taking this matrix product 'B' times Theta and getting all the probabilities for all words in all documents and then, reading off the id^{th} entry, which is the probability of seeing word 'i' in document 'd'. And so, we have this matrix product and then, we read off one entry. And so, notice that in a sense, topic modeling can be thought of as a non-negative matrix factorization. We want to learn a factorization of these non-negative matrix, of this matrix of non-negative probabilities, and we represent that, as equal to a product of two matrices, that also have non-negative values in it. So, 'B' and Theta are both matrices in this case, and both have non-negative values in it. So, we can view this, we can think of this, as a non-negative matrix factorization problem.

Video 10 (7:31): Nonnegative Matrix Factorization

So we've already described LDA and, and made the connection to non-negative matrix factorization. In the remainder of this lecture, I'm going to also discuss two other methods, for doing non-negative matrix factorization. And we'll discuss the actual algorithms as well for, for learning the parameters, given some data. So, both of these methods are— also have the name non-negative matrix factorization so it's a bit confusing, because non-negative matrix factorization is, is a technique, a general technique, and it's also, the name of a set of two different algorithms, which are called, for short, NMF. So, when we say non-negative matrix factorization, in general we're going to mean taking a non-negative matrix, and factorizing it into a product, of two non-negative matrices. So, the general problem, of doing that matrix factorization when we say, NMF, which is short for non-negative matrix Factorization, we're going to be thinking, in general, about the two specific algorithms that we're going to discuss in the rest of this class. The intuitions for NMF is similar to the probabilistic matrix factorization problem, PMF, that we discussed in the last lecture. So, we're going to—we can represent them in the same way. In this case we have an 'N', an 'N_1' by 'N_2' matrix, call that 'X'. So, this is our matrix of data where, the i^{th}, j^{th} entry called that 'X_{ij}', is a non-negative number.

And we're going to assume, for this lecture, that we have all values in this matrix. So, that's important. Last time we have many, many missing values in this matrix. This time, we're assuming that we have, every value in this matrix. But many, for example could be equal to zero. But there are no missing measurements here. And we're going to approximate this, as a product of, these two matrices, one called, 'W' and one called, 'H', where, if we make you rank 'k' approximation, that means that this is 'N_1' by 'k', which is the rank, and this is 'k' by 'N_2', and all of the values, in these two matrices, are also non-negative. So, that's the non-negative matrix factorization problem. And our goal now, is to learn this matrix factorization, that we're going to try to learn the values of these two matrices, to have their product approximate the measured data matrix. So, notice that, last time also we, the way that we wrote this, mathematically, was to say that, each row here, and each column was one vector. And then, we represented one particular entry as a dot product of two vectors. In this case, we're not going to write that. So, it's a purely a notational thing. We're going to say that, 'Xij', is equal to a sum over 'k', of 'W_{ik}' times 'H_{kj}' So we take, 'W_{ik}' for each value of 'k', we dot multiply it with 'H_{kj}', which is each value of, of this column vector, and then we sum it up, and that represents what our approximation of this element is in the data matrix. There are a few different problems, that we could consider, using non-negative matrix factorization for, we've already discussed topic modeling. In this case, we have our text data, where the data matrix contains word term frequencies, what that means is, the value 'X_{ij}', so, we have the data matrix 'X'. The value of the i^{th} row and the j^{th} column, is a count of the number of times, that the i^{th} word, word 'i' appears in document 'j'. So, in this case, we take this data matrix, and we view every column as a different document, and every row as a different vocabulary word in our vocabulary. And then, the element 'Xij', says how many times did word 'i', appear in document 'j'. So, that's the term frequency matrix, and then, we factorize that. Another instance of a data modeling problem, that uses non-negative matrix factorization is, image factorization. In this case, what we do, is we take an, 'N' by 'M' image, so, all the images have to be exactly the same size. We vectorize it, by taking the two-

dimensional values and putting them all, using the same exact method for each image. We put them all into one long vector of size 'N' times 'M', and then, we put each image along one column of the data matrix 'X'. So, for example, if we wanted to factorize images of faces, we would take each face, we would take the two-dimensional, image matrix of a face, put it, into a vector, one vector, and put that along the column of the data matrix, and then factorize that matrix. Other discrete group data, work just as well. This is a general technique, what is done is we take sets of features, and we quantize them using K-means.

And then, 'Xij' is going to count, how many times cluster 'i' is used by data in group 'j'. So, for example, we could have many, many different songs. Each song, in this case would be, like, one document. For each song, which is one long audio signal, we extract a set of features, that are 'd' dimensional. Each feature is 'd' dimensional, and imagine that, we have 'n' of these individual features, that are spaced out in time across the song. So, this could be, for example, something like the spectral information in the signal, where, 'd' would be the frequency content, 'n' would index a time slice, and we have a bunch of these features, extracted across the song, at different time points, saying this is the spectral, the frequency content, which will capture things like guitars, and voices, and so on. We then quantize all of those features for all the songs together, at the same time using. K-means clustering, where the 'k' now corresponds to, like, the vocabulary, the size of the vocabulary. So we set 'k'. Then, every song is now reduced from a set of 'd' dimensional features, to a sequence of indexes of, which cluster, each of those features was assigned to. And then, we histogram that put that along a column, of the matrix 'X', and factorize it. And that way, we can, we can say how songs use the codebook, you know, how they cluster across the codebook. And so similar songs should use this, use the codes from K-means in a similar way, and so on.

Video 11 (17:25): Two Objective Functions

Now let's go back to, being abstract about it, and discuss the two specific objective functions, that the NMF algorithm, so this is the specific instance of non-negative matrix factorization that we're discussing, the NMF algorithm. What are the two objective functions that, this algorithm can minimize. So, really it's a set of two algorithms for two different objective functions, and we'll discuss both of them separately. So, first, there's one choice, which is to try to minimize the sum of squared errors. So, if we approximate, 'X' as a product of the non-negative matrix 'W' and the non-negative matrix 'H', then we take 'X' and subtract our approximation of it. This is the error of our approximation, and then, what this notation indicates here is, we take each element, because this is a matrix of errors, we take each element, square it, and then sum it up, entirely. So, sometimes this is written with an 'F' down here, and it's referred to as the, Frobenius Norm.

So, it's just like this squared magnitude of a vector, except now, it's for the entire matrix. And that's written more clearly, or more explicitly, but less compactly, this way, where we take the ij^{th} element, we subtract our approximation of it, of the ij^{th} element, square it, that's the squared error of our approximation, and then sum up, all of the data. So, this is one objective that we're going to try to minimize, using the NMF algorithm. So, this will be one algorithm, of the two NMF algorithms. And then,

a different objective we could try to minimize, which I'll make more interpretable, in a later slide, is something called the divergence objective. And so, this is also like a distance measure, between the matrix 'X' and its approximation 'WH', and it's written this way. So, this will become much more clear on a later slide, but let's just look at what we have. We sum over all of the elements in the matrix, so we sum over 'i' and 'j' of the matrix—data matrix, the ij^{th} element to the data matrix times a log of the ij^{th} element, of our approximation minus, simply the ij^{th} element of our approximation. So, there's no data interacting with this term here, but there is data interacting with this term here, and we sum them up, we take the negative, and now our goal is to minimize this. That will be a different algorithm, a different NMF algorithm. The nice thing about NMF, in general, is that it's a very fast algorithm. So, these are going to be two very fast algorithms for learning the matrix 'W', and the matrix 'H', using something called multiplicative updates. I'm just going to present the algorithms. I'm not going to re-derive them, from scratch in this class. The original paper is found here, where they give everything, that's necessary, in a short seven pages. However, what their proof does, in showing that their algorithm does minimize these two objectives, is something very similar to the EM algorithm.

So, remember, that when we want to minimize some function 'F', so we're in abstract land here, if we want to minimize some function 'F' in the parameter, 'h' that can be put into it, one way to do this is to generate a sequence of parameters, 'h', such that, the evaluation of the objective is, monotonically decreasing in that sequence. So NMF does this, by doing something called introducing an auxiliary function. So, here we have 'F'. It introduces some auxiliary function 'G', that function looks like this. So, we want to minimize something that looks like this. We introduce at each iteration. So, at iteration 't', a function that approximates it, is equal to the function at, iteration h^t and is convex, and also bounded below, by the original function. So we get something that, intuitively looks like this. And then we, instead of minimizing, in iteration 't', this function, we minimize this red function in iteration 't'. And so we can guarantee, that we get a better value. Actually, if you look at EM, it's something very, very similar. So, this function is like the function 'L', if you recall from that lecture, that gap is like the Kullback-Leibler divergence, and then, this function is like the marginal likelihood. So, something very similar is being done here, but all of those details, are in this paper. For this lecture, we're simply going to go through the very, very simple algorithm, that results from that proof.

First, we'll discuss the algorithm for minimizing the squared—the sum of squared errors. So, again, our goal is, to minimize this sum of squared errors, with respect to the matrix 'W', and 'H', subject to, all of the values in 'W' and 'H' have to be non-negative. So, this is what we're trying to do, the optimization problem. The algorithm for doing this looks like this. We random initialize 'H' and 'W' with non-negative values, and then, we iterate the following, first, for 'H', then, for 'W', the order actually doesn't matter, but we'll show it for that ordering. We first, for every element in 'H', so the kj^{th} element for all 'k' and all 'j', we update the kj^{th} element, by taking it's old value and multiplying it by this gain. So, this is why, it's called a multiplicative update, because every update, to each value in 'H' and 'W' is found by, taking the old value and multiplying it by something. And notice that, this something is a function of the data matrix 'X' and the most recent value of 'W', and we just pick out, as well as the most recent value for 'H', and we just pick out the correct element in those products. So, this is something that we evaluate. We pick out the correct elements, multiply them together to get the update. Similarly, for the update of 'W', we take some matrix products, pick out the correct elements, multiply it by the original value to get the

new value. And what the algorithm, what the proof shows, is that by iteratively updating 'H' and 'W', in this way, we're continually finding new values for 'H' and 'W', that are monotonically decreasing this objective function. And so, we can evaluate this objective function here, by plugging in the values for 'W' and 'H', as a function of iteration, and then seeing how the relative improvement that we get, and then terminating when that improvement is small. So, here's a more kind of visually helpful, intuitive way of making these updates.

So, in practice, we can actually update the entire matrices, all at the same time, in this way. So, let's use this color coding, where, 'X' is the data matrix is yellow, red is 'W', and blue is 'H', and we're making this approximation. What we do then is we take the original matrix 'H', we dot multiply it. So, this is, 'MATLAB', speak for saying take this matrix and this matrix, and do element-wise multiplication. So, we do dot multiplication of the original matrix 'H', the old value of the matrix 'H', with this product of, 'W' transpose times 'X'. So, we take dot multiplication, element-wise multiplication of this. Then, we take this matrix multiplication, and do element-wise division of this result. So we do, element-wise multiplication and then element-wise division of that. This entire thing now, is the new matrix 'H'. So, this requires, one line of code. Then on the second line of code, we do this. We take the old matrix 'W', we do element-wise multiplication, of the data matrix 'X', times 'H' transpose, and then, that quantity element-wise division by 'WH', 'H' transpose, by this matrix product. So, this is, the second line in the code. If we want to give this a probabilistic interpretation, it's not necessary, or even useful, necessarily for this specific objective. We can, of course, view this, as saying that, were doing maximum likelihood for a model where, the elements of 'X' are Gausssian, with this mean, and this variance, where the variance in this case doesn't matter, subject to the parameter constraint that 'W' and 'H' have to be non negative.

So, this is kind of like, what we're doing, because we remember that, squared error penalization is like trying to maximize the log of the Gaussian penalty. Now, let's look at the algorithm, for minimizing the divergence penalty. So, let's look at the multiplicative updates, that result from the NMF derivation for this objective. So, again, our goal is to minimize an objective, in this case, it looks like this, subject to the non-negativity constraints, on the values of 'W' and 'H'. The algorithm, again, randomly initializes 'H' and 'W', with non-negative values, and then iterates with the following updates. These look a little bit more complicated, but again, they can be done in a couple of lines of code, where, we take the old value in 'H', we calculate it's gain and then multiply the two together to get the new value of 'H'. And similarly, we do that for each value in 'H', and then we move to 'W', and for each value in 'W', we take the old value and multiply it, by some function, of what we have, where, again, this 'W' here on this side, and this 'W' here is the old value, and then we get the new value. We iterate between these updates. We can evaluate this objective function. It will be monotonically decreasing, and we terminate the algorithm, when the improvement is relatively small. So, it helps to visualize this one perhaps even more, so, let's do, let's visualize it, in this way. We'll call the purple matrix by definition to be the data matrix, 'X', element-wise, divided by its approximation.

So 'WH', remember is the and so we take the data matrix 'X' and do element-wise division with its approximation. We'll call that matrix purple. Then to get the update of the 'H' matrix, we take the old 'H' matrix and we do element-wise, multiplication with the matrix 'W', where we normalize the columns of this matrix 'W'. So, this is 'W' transposed. We normalize the rows of 'W' transpose so that they sum to

one, and then we do, multiplication, matrix multiplication, of this matrix, with the purple matrix, and then do element-wise multiplication. Similarly, for the updating, the matrix 'W', we take the matrix 'H', we transpose it, and then, normalize the columns of the transpose or equivalently, what we would do, is normalize the rows of the original matrix so that they sum to one, and then, do matrix multiplication, of this purple matrix that we get this way, with the normalized matrix, and then, element-wisemultiplication of the result. So, something to notice here, that's a—that could end up making your algorithm not work is that this purple matrix, when we update the blue, when we update 'H', we need to then update this matrix here because 'H' has changed, and therefore this element-wise division has changed. So, the algorithm would look something like, iterating between, updating this matrix purple, updating blue, going back to update this matrix, and then updating red. So, perhaps it can be done in about four lines or so, of code. The divergence penalty is a bit less clear, it's harder to interpret, at face value, than the squared error penalty. However, the probabilistic interpretation of that divergence penalty, that we just discussed is perhaps more satisfying than the interpretation of the squared error penalty.

So, I have a slide here, where I'm going to discuss, what is the equivalent probabilistic model, that we have, that would lead to the same exact update rule, as we have with the divergence penalty with this algorithm. So, imagine, that we have this generative process. We say that the ij^{th} element, in our data matrix, is a Poisson random variable, with parameter equal to, the ij^{th} element, of the matrix product, of 'W' and 'H'. So we take 'W' and 'H', we multiply them together, we look at the ij^{th} element, for the Poisson of the data 'X_{ij}'. So, remember a Poisson, is a discrete distribution on the non-negative integer, so, a random variable 'X' that's Poisson distributed, can take values from one—zero, one, two, up through, infinity, but all integer values. And Lambda is the parameter, and what we can show is, that the expectation of 'X' is equal to Lambda. So, in a sense, this product, this matrix factorization we're doing, is like learning the expected value for the data matrix, under a Poisson generating assumption, likelihood assumption. So, if we make this assumption, this modelling assumption, and then, we want to do maximum likelihood for 'W' and 'H', under a constraint that they have to be non-negative, what we'd get is something like this. What we first have, that the likelihood of the entire data matrix 'X', given 'W' and 'H', because we make an independence assumption here in this thing, so I'm making an independence assumption, it turns into, the product over every element in our data matrix 'X', so we have a product over all the rows and all the columns, 'ij', of the likelihood of each element of 'X', given 'W' and 'H', which can also be written as a product, over Poisson random variables, where, we're evaluating the distribution at point 'X_{ij}', using parameter equal to the ij^{th} element of the matrix product 'WH'. So, we have this product, and now we want to do maximum likelihood, for this data likelihood. So, remember that, what we can do is take the log of this likelihood, and maximize that instead, and when we do this, what we find is that we get this term here.

So, the log of the Poisson is equal to this term, where we throw in a constant here to take care of, for example, this term. We have the sum over the logs of the individual Poissons. So, this is the log, this term here is the log of this likelihood—we want to maximize over 'W' and 'H'— and then, if we add some constant to it, we can show that, that is equal to this term here, which is equal to the negative of the divergence penalty, that we're trying to minimize. So, we want to minimize the divergence penalty. That's equivalent to maximizing the negative of the divergence, which equals this term, which is

equivalent to maximizing the likelihood of this Poisson model. So, this makes a bit more sense. For example, if we're trying to factorize matrices with counts in them, like term frequency matrices to do topic modeling, the data matrix has integer-valued counts, and a Poisson, is a natural model for that, because the counts could be zero through—zero one, two, up to, there's no limit to how many counts there could be. However, it's very low probability to have high counts, which can be captured by this distribution. And so we're doing maximum likelihood for a model that fits the type of data that we're seeing. So, in a sense, this penalty makes more interpretive sense.

Video 12 (8:25): NMF and Topic Modeling

So the way that we can do topic modelling, with NMF, using this divergence penalty, is we can first, form the term frequency, matrix 'X' where, we let 'X_{ij}' equal the number of times, that word 'i' appears in document 'j'. So, where, 'i', is just some arbitrary indexing of our vocabulary, 'j' is an arbitrary indexing of our document, and now we count how many times these things occur, and put them in the matrix 'X'. We then run, NMF, to learn this factorization, 'W' and 'H', using the divergence penalty, and then if we wish, as an added step, what we can do, is let 'a_k' equal the sum, of the k^{th} column, of the matrix 'W'. So, we take the matrix 'W', we sum all of it's columns and let that be 'a_k', for each 'k', each column. We then divide the k^{th} column by 'a_k' and multiply the k^{th} row of 'H' by 'a_k'. So, if we do this, all we've done is scaled our data, but we haven't changed the multiplicate the product of our data 'WH'. So also, if we think, in terms of maximum likelihood, and we think in terms of divergence minimization, notice that both of those penalty terms, don't penalize the actual values of 'W' and 'H'. All they do is look at the resulting values in this matrix product. So, by doing this, this additional step, we have not changed, the maximum likelihood solution that we found or the minimum of this divergence, because they only depend on the product. But now we can interpret the columns of 'W', as probability distributions on words in a vocabulary. And so we can interpret the columns of 'W' as topics.
And therefore, if we factorize the data matrix consisting of counts of numbers, of occurrences, of words in documents, we can view the ordering of the highly probable columns of each column of 'W', rows of each column of 'W', as a topic. And we'll see that we actually learn meaningful topics, this way. And so this model can be used for topic modelling in that sense. In the next three slides, I also want to show three different algorithms, one of them being NMF, and what they learn on the same exact data. So, this data consists of many images of faces, that look similar to this one right here, where each one is taken, and it's a gray scale image. It's put into a vector, and then we put each of the images along a column of a matrix, 'X'. So, each column of 'X' consists of, different images of faces, and each row corresponds to a pixel in the original image. And then we factorize this. And so we factorize this learning seven times seven, so about '49' different clusters or topics, if you will, something like this, and then we represent each face in how it uses those '49' different clusters. So, I say clusters, because the first example we look at is K-means, also called vector quantization, and so these correspond to say the '49' different centroids, that are learned by clustering this data, and then we represent a particular image, by which cluster it belongs to.

So, if we want to think of this as a matrix factorization, we can think of this data as being represented by taking this factor, which all are shown as images, just to give interpretability, but this would be one column of 'W', a second column of 'W', and a third column, etc., of 'W', and so we take this factor, and multiply it by the number in this element here. So, this is a seven by seven matrix. This element is zero times this. And we take all of these, and we have zero times everything, except for here, where we have a one in the black square, which we multiply against this image here, and you can notice that this image, is equal to, this image. So, that's like hard clustering. We say that, we have to represent each data point by centroid of the cluster it belongs to, and in that case, what we can say, if we want to view that as a matrix factorization, is that every data point can be—has to be a matrix, of factor loadings, like this matrix, times a vector with all zeros except for a one according to the cluster assignment. And we're representing all of those vectors and matrices, in this way just for visualization, and so with K-means clustering, we have a hard clustering assignment, so that means, all zeros except for one according to the cluster it belongs to. Now let's look at SVD. If we took exactly the same data matrix, of columns of faces and we did SVD, and we looked at the singular vectors, the left singular vector is shown as an image. We would get something like this. So, remember the singular vectors all have to be, within normal.

So, we get something like this, where this is the mean of all the faces that's shown as the first one, and then these are the singular values of everything else. And so we get a lot of negative numbers, and positive numbers represented by red and grey, and they all have to be orthonormal. So, if we took this image here, and we multiplied it by this image, and then summed up, it would equal zero. And if we took this image, and multiplied it by itself, and summed up, it would equal one. And so that enforces, that we learn this type of a factorization, that orthonormality constraint, and then, we learned a non-sparse representation of this image and this...these factors. So, we take this color represents intensity, or value, we take this, times this image, plus this element here, times this image, plus this scalar here, times this image, and so on, we multiply them all, we add them up, and we get this face. So finally, these are just to give comparisons to what NMF learns. And this is the, one of the key features of non-negative matrix factorization. If we looked at, what the different columns of 'W' look like, on the same faces data set, represent those columns of 'W' as images in the same original space, we get something like this. And so notice that each of these images picks out a part, of the face. And now, we represent this face as a weighted sum, of different parts, of these faces, that we learned from the data set.

And so in this case, NMF will often be described as learning a parts-based representation. Because for faces, we take each face, we take all the faces, and we learn different parts of those faces, and represent each face, as some linear combination of those parts. If you want to think of topic modelling, we take all of the documents, we learn the different parts of the documents, in this case, those parts would be topics or themes, coherent themes, and then, we would represent each document, which will be a column of the matrix 'X', as a linear combination of those different parts or themes, and so parts-based, when we talk about images, we're thinking about parts-based learning. When we talk about documents, we're thinking about topic-based learning. Really, they all are talking about the same thing, if you just think about, non-negative matrix factorization. If taking the data, and finding the different aspects of the data, that underlie it, and then representing each data point as a sum of those aspects. And we get this, the reason that we can learn something like this with NMF, while we don't learn it with SVD, is that we

have nonnegativity constraints. So it's the nonnegativity constraints, that enforces this parts-based learning.