Week 10
Video Transcripts

Video 1 (12:16): Principal Component Analysis

Principal component analysis is often used for something called dimensionality reduction. Here, is a slide that's going to make that a bit more intuitive. Imagine we're given data 'x_1' through 'x_n', where each point is in R^d, where we're going to assume that 'd' is very high-dimensional. However, we don't think that the information fills all of those dimensions. We think that we can take all the information contained in that high-dimensional space and map it to a much lower-dimensional space, where we don't lose much information— in this case, meaning that the vectors in the lower-dimensional space all kind of have the same relationship to each other that they had in the higher-dimensional space, and so, learning an algorithm on data from the lower-dimensional space shouldn't give worse results than what we would get from learning on higher-dimensional space. Here's an example of that, where we have high-dimensional images, so, if we view this as– each image, as a point in a very high-dimensional space, we could have—we would have these three different, these five different points, all in the same space. However, instead of having to represent it in that image space, if you notice that these five images, all really can be captured by three numbers, since all we need to do is, learn a shift and learn a rotation, two shifts and one rotation, for these three images. So, where's the center? That's a two-dimensional vector for the center of the three, and then the rotation is a third dimension. And so, really, we only need three numbers for this data set to capture everything going into it, and then we just also need the...the base pattern number three, that's going to be reappearing in each of these images. So, this is an extreme example but also, actually very relevant to many data sets. So, principal component analysis can be thought of as a way for automatically learning a mapping from this high-dimensional space to a low-dimensional space, such that, we capture all of the information from the high-dimensional space. To discuss principal component analysis, it helps to think of a toy example. So, how can we take this data set of these points in two-dimensional space so each point, for example, 'x_i' is represented by two numbers that gives this point here? And we want to take a vector 'q' of unit length and represent each of these points in terms of the vector 'q'.

So, PCA essentially does this problem. It says, take this vector 'q', make it unit length, and now represent each of these data points by dropping a line perpendicular to the line constructed by the direction of 'q'. So, this line– this arrow 'q' defines this line here. And now, we're going to take each of these points and drop it to that point there, perpendicular to 'q'. Now, we want to take each of these points and represent it by the amount that we have to stretch 'q' to get to the red dot by dropping it to that line. So, for example, we're going to take this two-dimensional vector and represent it by one number. It's the number that we have to multiply 'q' by to get it to stretch out to this red dot here, which is the closest we can...we can take 'q' and get it to the point 'x_i'. So, we've already discussed some of this, these geometric concepts in the classification of a portion of our course, where we talked about linearly separating hyperplanes and dot products. In this case, that comes– that shows itself by taking the dot

product between 'q' and each point 'x_i'. And now, if we wanted to know what does this dot product of this unit vector 'q' with 'x_i' represent, that represents the length from this origin to the red dot here. So, if we want to know what's the length from the origin to the red dot that I get by dropping this point 'x' perpendicular to this line here defined by 'q', that length is equal to 'q' transpose times 'x_i'. Therefore, if I want to take the vector 'q', and I want to stretch it, so that I hit this red point here, which is the closest I can take 'q' and get it to 'x_i', I simply want to take 'q' and multiply it by the scalar that I get from 'q' transpose 'x_i'. So, this is the amount that I have to multiply 'q' by to stretch it out to get it to hit that red point. And now, in that sense, so this is a very simple toy example that doesn't do justice to what PCA is used for, but it's, we can visualize it. In that sense, what we've done is we've taken each of these two-dimensional points, which each requires two numbers to represent it, and we've reduced each point to requiring only one number to represent it, and that's the number that I need to multiply 'q' by, to stretch it out to get as close as possible to 'x_i'. So, I've done a very trivial and not interesting example of dimensionality reduction by taking a two-dimensional vector, and reducing it, now to one dimension. And so, for each of these points, I need to keep only one number. I need to keep the value of 'q' transpose 'x_i'. And then also, somewhere, I need to keep the vector 'q' potentially, if I want to construct this— try to reconstruct this space.

So, this, as simple as it seems and as potentially uninteresting, given this plot, as it seems, this is the basis for PCA and it's very important and very useful. And so, now the question actually becomes, if I only get one vector 'q', that has to be unit length in order to do this, what's the best 'q' to pick? What's the best vector? So, I picked a good one here that kind of is cutting through the data nicely. I could have picked another vector 'q' and point it in this direction and maybe that– and that would have given a completely different results. So, which is the best vector 'q' to pick for a problem like this? And so, to do this, we, again, have to set up an objective function, something— some measure of quality that we can then— a mathematical measure of quality that we can then minimize to say we've found the best vector 'q' according to this measure of quality. And so, the way that we do this, which is no surprise, because this is a squared error. So, we take the squared error term of this approximation. So, we have a vector 'x_i' and now we approximate that, as the— with the vector 'q' stretched out by the product 'q' transpose 'x_i'. So, we take 'q' transpose, multiply it by 'x_i'. That's the amount that we now stretch out 'q', and this is now our approximation in our— in the same dimensional space as 'x_i'.

And so, we approximate 'x_i' with this, and so that's all this is showing. We're taking this two-dimensional vector and approximating it with this two-dimensional vector but all of those two...all of those approximations have to fall on the same one-dimensional subspace. We squared that error and now sum the squared errors and we now try to find the minimum value of the vector 'q' to minimize the sum of squared error subject to 'q' having to be unit length. So, this is that the PCA problem in a nutshell. However, we're going to simplify it— or we're going to rewrite it in a different way... in a different equivalent way, which is how it can be done in a computer. So, we want to minimize this. We want to find a vector 'q' and so now in your mind, you can imagine that this is much higher-dimensional, 'x' could be thousands of dimensions, and 'q', therefore, is '1,000' dimensional vector that has unit length. And now, we want to find that individual, single, high-dimensional vector to approximate the vector 'x_i'. And so, again, we take that vector, we take the dot product of it with 'x_i', that's the amount that we have to stretch that vector in order to get it as close as possible to 'x_i'.

So, this is our approximation, this is our error, We sum—we squared the error, and sum it up subject to 'q' having to be unit length. But now, instead of writing it this way, let's expand this product out. So, we multiply this whole thing out. So, remember this is a vector and if we want to know the squared magnitude of that vector, we can simply take the vector, transpose, multiplied by itself. And so, when we do that, we get an 'x' transpose 'x', which is this term here, and when we multiply this thing by itself, what we get is a 'q'— we get some 'q's' that cancel out with each other. And when we multiply these things together, we also get some things to cancel out with each other. So, that the cross term of this, as well as this term with itself, gives this product, right here, so this is an equivalent way to write it. And so, we've simplify — I've simplified this by taking the sum, bringing it in here. And now we want to find the 'arg min' of this term minus this term. And so, now instead of summing the outer products of each of these points 'x_i', let's view this as a matrix product for simplicity. So, in the first half of this course, we viewed each of these 'x_i's' as being put on a row of a matrix 'X'. Now we're going to switch that and we're going to say, "Take each vector 'x_i' and put it in a column of a matrix 'X'." So, the matrix capital 'X' is constructed by taking each of our data points, 'x_i', and putting it along a column, and so, it's 'd' by 'n'. And now, this product is equal to 'x' transpose— 'x' times 'x' transpose, and that's kind of sense.
So now, we want to minimize this term, which is exactly the same as this term, but written in a different way and also using matrix products here. However, if you notice, this term doesn't include 'q' at all, so we can simply throw it away. And if we want to minimize a negative term, we also want to maximize the negative of that term. So, we want to maximize this term in 'q'. And so, an equivalent of mathematically identical way to represent this problem is finding the vector 'q', that maximizes this quadratic term subject to the magnitude of the 'q' equaling one. So, we want to take 'q' transpose times the matrix that we get by taking this matrix product times 'q' and we want to now find the vector 'q' that maximizes that product. This vector 'q' has a name, it's called the first eigenvector, of the matrix 'XX' transpose. Notice that this thing is a 'd' by 'd' and symmetric. And the value of this product evaluated at the optimal 'q' is called the first eigenvalue. So, if we take 'q' transpose times this matrix times 'q', the value of the maximum, that's the maximum value we can get by that type of a product subject to this constraint, call that Lambda, that's the first eigenvalue.

Video 2 (08:43): PCA: General I

Okay! So that's for finding— that's for how we would represent every data point in— with one vector, meaning we take the original data and we map it down to one-dimensional space, so we can represent every data point now with one number. It's the amount that we have to multiply 'q' by to get it as close to the original data as possible. However, PCA, in general, doesn't just consider one eigenvector and one eigenvalue, meaning that it doesn't map the data to one-dimensional space. It often will consider 'K' eigenvectors, where 'K' is arbitrary. So, we want to find 'k' vectors in order to do this and so now the problem becomes this— like this, where we want to approximate 'x_i' by a linear combination of 'k' different vectors, where each vector has unit length, so, the dot product of any of those vectors with itself, so when 'k' equals 'k' prime is equal to one. And also, those vectors have to be orthogonal to each other.

So, 'k' acute 'k' times 'q_k' prime has to equal zero, if 'k' does not equal 'k' prime. So, we have these two constraints, that we now get 'k' vectors, 'q_1' through 'q_k', and they're all unit length and they're all orthogonal to each other, meaning that their dot products equal zero, across different vectors. And so, again, if we want to represent our approximation of 'x_i' as a linear combination of these 'k' different vectors, we take each 'q'– each eigenvector, 'q_k', and we stretch it by the dot product of the point 'x_i' with the eigenvector. So, this is the amount— this dot product, again, is the amount that we have to stretch 'q_k' by to get that individual vector as close to 'x_i' as possible. And now because each of these 'q's' are orthogonal to each other, we treat it as 'k' different approximations and then sum each of those up. So, we're trying to approximate the data with a 'k' dimensional vector, which is the 'k' dimensions that we get from the 'k' different dot products here. Okay.

So we want to minimize this now over 'q', over capital 'K' different vectors for 'q'. And again, we follow exactly the same procedure and multiply this all out and because of this constraint on the vectors, we get some nice cancellations that give us this problem. So, we want to now minimize this term, which is exactly as before, minus and now notice that we have a sum over 'k' different vectors this time. Before, we had one vector, there was no sum. Now we sum the 'k' different vectors and we have, again, exactly the same matrix product here as before and exactly the same vector-matrix product here but now we have 'k' then that we sum up. And so, if we want to now minimize this, we can, again, of course maximize this over the 'k' vectors 'q' subject to the these two constraints. And now these vectors, these 'k' different vectors give us a 'k' dimensional representation of our data. So, now we've taken the original data, and reduced the number of dimensions to 'k', where the projection that we get of the data from the higher dimension, is simply represented by taking the dot products of the data with each of these vectors 'q'.

So, again, 'x' can be a very high-dimensional vector. We learn 'k' of these eigenvectors. We then take the dot product of 'x' with each of those 'k' eigenvectors. And those 'k' dot products give us the 'K' dimensional representation of 'x'. So, now we've got 'x' in 'd' dimension. We represent it in 'K' dimensions in this way. So, this is our new coordinate representation for any vector 'x'. And, of course, if we want to know how this projection relates to the original value, what we're saying is that our approximation of 'x' is the dot product of 'x' with the k^{th} eigenvector, which gives us the amount we have to stretch that eigenvector to approximate it, sum over all of the eigenvectors. So, in practice, when we do this, what we'll often do is we'll take this matrix that we get 'x', take its dot product, and then call some function in the coding language we use in order to find the 'k' largest eigenvectors, as well as their associated eigenvalues, which is the value we get– the k^{th} eigenvalue will be the value of this product for 'q_k'.

Another way that we can write this eigenvalue problem is as follows. So, we can equivalently, mathematically equivalently say that we're trying to find a vector 'q' and a scalar at Lambda, such that 'q' times this matrix product is equal to Lambda times the same vector. So, what we're saying is that this matrix here, multiplied by 'q' doesn't change its direction, it only stretches it by a value Lambda. Also, since this matrix product is a positive semi-definite matrix, we know that there are 'r' pairs of these things, where 'r' as the minimum of the dimensionality in the data— number of data points, where we have, all of these values are positive. This result—these eigenvalues being positive is a result of this being a positive semi-definite matrix, and where we can get 'r' these different eigenvectors where each

one is unit length and orthogonal to each other. Okay, so this is a something that we can say in advance about this matrix, why is it positive semi-definite though? So, it's very easy to show actually using the SVD.

We've seen how any matrix 'x' can be broken down as a product of the 'U', 'S', and 'V' transpose, where 'U' and 'V' transpose are orthogonal. So, the columns of 'U' are orthonormal and the rows of 'V' transposed are orthonormal. And 'S' is a matrix that has non-negative diagonal values. And so, now if we take 'X' times 'X' transpose and we write it in terms of its singular value decomposition, what we have is that this product, this matrix product is equal to 'U' times 'S' squared times 'U' transpose. And so, in that sense, we can find the eigenvectors and the eigenvalues of this matrix product by just taking the SVD of the original matrix.

We know that the eigenvectors, so this matrix 'Q' has at the eigenvectors along its column, is equal to the matrix 'U' that we get from this eigen decomposition. And also, the i^{th} eigenvalue is equal to the i^{th} diagonal element of the matrix as squared. So, we take the—we get the singular values, we look at the i^{th} one, square it, that's equal to the i^{th} eigenvalue. And, of course, because it's a squared value, it's got to be greater than zero. Also, as a preprocessing step, what is typically done is that the rows of 'X' have their mean subtracted off. So, we center the data so that all of the dimensions in our data set have mean zero and that's so that we get something like this, where we want to– we want the data to be nicely centered along the original axes, so that we can represent this data in its first principal component and its second principal component. So, that's a fact– that comes from the fact of these principal components having to pass through the origin. If you imagine this data set shifted very far away from the origin, it would be much harder to find such a nice representation, if this were rotated a little bit.

Video 3 (05:24): PCA: General II

Here's another example where we have data in R^3 and we want to project it to R^2 using the eigen decomposition. And I've color coded the data here to show how the mapping looks. So, in the left's hands part of the slide, we have data in R^3. So, this—each of these points corresponds to the three-dimensional coordinates of the original data set. When we try to find an eigen decomposition to project it into two dimensions, it's like finding a plane that slices through the data, with as little error as possible. So, we take a plane, we drop a line to the hyperplane perpendicular to it, that's our error, and we sum that squared error, that gives us— that's the thing that we're trying to minimize. And so, we get something like this, if we took this three-dimensional data and represented it in two dimensions, we would get something like this. And so, we capture as much of the information as possible within these two dimensions. Notice that if we had just simply taken two of the original dimensions, and thrown away the third, we couldn't have gotten as nice of a representation of the data set.

So, it's like trying to shift and rotate the data around so that we can just throw away some of the dimensions but keep the remaining— but, where we've aligned the information as much as possible along the dimensions that we're going to keep. And here's one more example. In this example, we have '16' by '16' images of handwritten digits, threes. And so, the data is now '256' dimensional. However, we

don't think that '256' numbers are necessarily required to keep all of the information, all the important information about each of these threes. And so, we project them into a lower dimension using the eigendecomposition, the PCA, and so, here's what we get from the data set. Here's first the mean that we subtract off. So, we take this number and subtract every— subtract this vector or image from every image of a three. And then here's the first eigenvector that we get. So, we learn—we take each of these threes, we make them into a '256' dimensional vector, stack them into a matrix 'X', learn the four—first four, or first however many eigenvectors of that matrix. And then each eigenvector can be represented as an image, each eigenvector is '256' dimensions. We then put that back into the '16' by '16' matrix and show it as an image and we get something like this. So, here's the first eigenvector we learnt. This captures most of the information, the second, the third, and this captures decreasing amounts of whatever information is left and not captured by the previous eigenvectors.

So, if we took this image, and dot multiplied it with any other image here, and summed it up, it would equal zero. And if we took each of these images and dot multiplied it by itself and summed it up, it would equal one. And so, now we have an image that comes in at '16' by '16' and we want to project it into a few dimensions. What we do first is, we subtract off the mean from it. We then take the remaining, mean subjected image, and dot multiply it by each of these, and sum up the results. And we can get now four numbers, which is the result of that corresponding dot product of each of these images with the image that just came in. And now, we—if we want to reconstruct the image, we take the dot product of our mean subtracted image, we then multiply it by the eigenvector, and now that's the approximation, like we've discussed previously, of the original image, in that dimension only and we summed them up. And we get something that looks like this. So, here's the original image.

If we want to represent that only in one vector, we would represent in the mean, so that's this term here. But now if we want to represent it with one mean vector and nine eigenvectors, meaning we learn nine of these things from the original data, and now we take this image, we subtract off its mean, we take the result, dot multiply it by each of these images and sum up the results to get nine numbers and then we take the numbers associated with this dot product, multiply it by this image, and sum it with the same things for each of the remaining nine images, we will get something like this.

So, this image is a '10' dimensional— or really, a nine-dimensional representation of this original '256' dimensional representation of the three. And you can see by the time we get to '49' eigenvectors, we're very close. By the time we get to '249', so there's only seven remaining, we're almost identical. And the remaining ones would pick up on this noise here. And so, here's what I've been saying. We're representing this image, 'x', as being approximately equal to the mean, plus the dot product of 'x' with each eigenvector and then we scale the eigenvector and sum it up. This is the approximation that we're making.

Video 4 (18:16): Probabilistic PCA

Next, I want to take this idea of PCA and develop it in two directions. The first direction is called— using something called probabilistic PCA. So, as the name implies, this is where we take PCA and we build it into a—we view it as a probability model and learn the parameters using a probabilistic representation.

So, in that sense, I want to recall what we discussed previously, the connection between principal component analysis in the singular value decomposition. So, we've already discussed how data matrix 'X', automatically have the singular value decomposition. So, we can write any matrix 'X' as a product of a matrix 'U', 'S', and 'V' transpose, where 'U' transpose 'U' is the identity matrix and 'V' transpose 'V' is the identity matrix. So, here, I'm assuming 'X' has more columns than rows. If it had more rows than columns, then some of these transposes would be switched. Also, the matrix 'X'— 'S' is a diagonal matrix, so all the values in it are zero, except for along the diagonal. And all the values in 'S' are non-negative along the diagonal.

And so, therefore, this matrix that we get, 'X' transpose— 'X' times 'X' transpose, which is what we want to learn the eigendecomposition of, is equal to 'U' 'S' squared 'U' transpose, and similarly, we have this product, 'X' 'X' transpose 'U' is equal to 'U''S'...'S' squared. And so, remember the representation from a previous slide, where we had Lambda here and 'q' on both sides, here now we have those eigenvectors along the columns of 'U' and we have the Lambdas along the diagonal elements of 'S' squared. So, 'U' is the matrix of eigenvectors, 'S' squared is the diagonal matrix of the eigenvalues for our eigendecomposition of this matrix. So, now we want to think of a modeling approach to PCA. What we're going to discuss next is a model that uses the EM algorithm, so we have to define a generative process and then do EM to learn it, a point estimate of it. And what this model is going to be able to do that the original PCA couldn't, is handle some additional issues that might arise. For example, we might want to handle missing data, PCA can't handle that issue as easily, directly.

This can handle missing data fairly straightforwardly. It will also allow us to do additional things, like learn model parameters, such as noise. We can build it into a framework that's part of a more complex model and also learn uncertainties about certain parameters. So, there are some reasons why we might want to choose a probabilistic model; there are also many reasons why we wouldn't. So, this is just a different technique, that we're going to discuss. So, really, the idea of using probabilistic PCA is— can also be viewed as doing probabilistic SVD, because of the connection. So, remember we have that 'X' is equal to 'U' times 'S' times 'V' transpose, if we want to write it as an SVD. And that we can then get 'U' and 'S' and say that those are the eigenvectors and eigenvalues. So, we can directly learn this sort of a factorization without having to take the product of 'X' with itself. And so, now that's what we're going to do, we're going to say, that 'X' is approximately equal to 'W' times 'Z', where 'W' is a 'd' by 'K' matrix. So, 'd' is the dimensionality the data, 'K' is the number of eigenvalues and eigenvectors that we want to learn. But in this setting, we can call 'W' something else, we'll call it a fact loading matrix or a dictionary, so it has many different names, depending on the problem that it's being applied for—to. However, we're also going to relax the constraint that 'W' has to be orthonormal, so that's a crucial difference between probabilistic PCA and regular PCA. The columns don't have to have, unit length and they don't have to be orthogonal to each other, so that's a significant relaxation, we're going to make. And also, the matrix 'Z' now, is going to have vectors in it, 'z_i' along the i^{th} column of 'Z', where each one now is in R^K. And we're now going to think of 'z_i' as being the low-dimensional representation of the corresponding value 'x_i' in 'X'. So, if the i^{th} column of 'X' corresponds to the i^{th} data point, the i^{th} column of 'Z' corresponds also to the 'i' data point, except projected into a lower 'K' dimensional space. Okay, so now we have to define the generative process, and this is the process for probabilistic PCA.

We assume that the i^{th} data point, 'x_i', is a Multivariate Gaussian with mean equal to the matrix 'W', which is 'd' by 'K', times the vector 'z_i', which is a 'K' dimensional vector, plus some noise. And then, the matrix— I'm sorry, the vector, 'z_i', now is a–also a random variable from a Gaussian with zero mean and identity covariance matrix. So, this is the generative process for each data point. And now in this case, we don't know 'W' and we also don't know any of these vectors 'z', so now we want to learn both of these things. And the way that we're going to do this is through maximum likelihood. So, our goal is to find the maximum likelihood solution of the matrix 'W', under the marginal distribution. So, our goal is to integrate out all of the 'z's', the impact the 'z', we want to integrate it out of the model and then to a point estimate that maximizes the marginal likelihood, with respect to the matrix 'W'. So, in notation, what we're saying is that the maximum likelihood value for the matrix 'W' is equal to the maximum of the log of the joint likelihood of all the data given 'W', whereas 'z' is integrated out, and now, because we're going to make an independence assumption among our data, this likelihood is a product of the likelihoods of each observation, and so that turns into the 'arg max' over 'W' of the sum over each data point of the log of the likelihood of each individual data point, given 'W'.

However, if we actually calculated this marginal distribution, we would find that the likelihood of 'x_i' given 'W' is equal to a Multivariate Gaussian, with mean zero because 'z_i' has a zero mean and covariance equal to Sigma...Sigma squared 'i' times 'W' 'W' transposed. So, this is the covariance matrix of 'x_i', this diagonal matrix plus 'W' 'W' transpose. And now if we actually looked at what this density looked like, and then we take the log of it, we'll find that we can't solve it with respect to 'W', analytically. So, take the log of this thing, take the derivative of it, with respect to 'W', the matrix 'W', and you quickly find that we're...that we're in a bind, we can't do this, and solve for 'W', analytically. And so, what we're going to do is reintroduce the vector 'z' as the hitting—as the missing part of our model and then do EM on that. So, this is another example of an EM algorithm, of where EM is useful. So, to do this again, we have to set up the EM equality, we have the marginal likelihood here and so I'm writing the marginal likelihood we want to maximize over 'W' as an integral over 'z'. So, 'z' is actually integrated out here, but I want to show explicitly that it's being integrated out.

This marginal likelihood, which is hard to optimize over 'W', is equivalently represented by this— sum of these two terms. So, this left hand side is equal to the right hand side, where now I have introduced a 'q' distribution on each of these vectors, 'z_i', and I have noticed here— the joint likelihood of 'x_i' and 'z_i' together, so this is what we've been calling 'L'. And in the second line, I add to it the— I'm sorry, the Kullback-Leibler divergence of the 'q' distribution on 'z_i' and the conditional posterior distribution of 'z_i', given the data point 'x_i' and given the matrix 'W'. So, these are the Kullback-Leibler divergences. So, the sum of these two things, is equal to this left-hand side. Now, if we want to use the right- hand side to do EM in order to maximize the left hand side over 'W', remember we have to do two steps. First, we have to make this Kullback-Leibler divergence equal to zero and that's the E-step. And so, we set—in order to do that, we know the only way to do that this could equal zero is by setting 'q' to equal this posterior — this distribution here. So, we have to match these two distributions. So, in a particular iteration, we have a particular value for 'W', we calculate— we set 'q' to be equal to this conditional posterior distribution and we hope that we can calculate this and fortunately, we can in this case. That's going to set all of these KL divergences to zero and then we calculate this expectation, this integral, using that update for the 'q' distribution.

So, we calculate now, that's the E-step. Then the M-step is to maximize 'L', to maximize this term over the matrix 'W', so that's the M-step. And so, again, if we wanted... if we want any hope for EM to help us with this problem, we have to know that we can solve this condition posterior in closed form and we also have to hope that the maximization of this term over 'W' is easy because remember, this maximizing this thing over 'W' was hard. If this thing is also hard, then we're still in trouble, so we hope that we can do this in closed form and fortunately, we can. And the result is this algorithm so I'm not going to derive everything from scratch. I'm just going to present the algorithm for doing EM for probabilistic PCA. So, we're given data 'x_1' through 'x_n', where each point— each data point is in R^d, where 'd' is a high-dimensional space. And we're going to model each of these data points as being a Multivariate Gaussian with mean equal to the matrix 'W' times 'z_i' and times...times Sigma squared 'i', and each of these vectors 'z_i', which is a 'K' dimensional vector is a standard Gaussian. So, 'W' is 'd' by 'K' and 'z' is a 'K' dimensional Gaussian. What we're going to output is a point estimate of 'W', that maximizes the marginal likelihood and also a conditional posterior distribution of each 'z_i', which gives us our posterior belief about what the lower-dimensional... low-dimensional embedding should be.
So, we don't just get a low-dimensional embedding for each point 'x_i' in the 'q' of 'z_i', we also get, some distribution on that embedding. So, the E-step, we have to set 'q' of 'z_i' to be equal to its conditional posterior distribution. This is a straightforward calculation. We can show it's a Multivariate Gaussian with mean Mu 'i' and covariance Sigma 'i', where Sigma 'i' is equal to the identity plus 'W' transpose 'W' divided by the noise parameter inverse. So, every single 'z_i' has exactly the same covariance in the conditional posterior. And then the mean of the conditional posterior is equal to the covariance times 'W' transpose 'x' Sigma— divided by Sigma squared. So, we calculate this using the most value for— most recent value for 'W' in both of these equations. And so, because 'x_i' appears here, the mean is what's different for each data point, uncertainty is the same. And then we calculate 'L', and I am not showing that calculation here. Then the M-step, it involves taking the derivative of the objective 'L' with respect to the matrix 'W' and solving, in that case, we can do that as well in closed form and we get this update.
So, we update 'W' after updating each of these 'q' of 'z's'. We update the matrix 'W' by taking 'x_i' times the mean, Mu 'i' transpose, so this is the conditional posterior distribution of 'z', the mean of it transposed. So, this outer product is a matrix that's 'd' by 'K', and then we sum it over every single data point. So, this is a 'd' by 'K' matrix. And we multiply it by the inverse of this matrix, which is the— noise variance, Sigma squared 'i' plus the sum over each data point of this term, which is the outer product of the posterior mean of the embedding 'z_i', which is 'K' by 'K', plus the posterior covariance Sigma 'i'. So, this is a 'K' by 'K' matrix, we sum it over every data point, and invert it, and the result is still 'd' by 'K'. And so, that's our update for 'W'. And then because 'W' has changed, we update each of these 'q' distributions, then we iterate back and forth, until the improvement to this marginal likelihood is small. So, even though we couldn't take the derivative of this with respect to the matrix 'W' and optimize it, we can still evaluate it at each value of 'W'. So, we still can use this equation where we take this— the log of it over each of these x_i's here to evaluate at each point 'W' to get the log marginal likelihood. And then we terminate when the improvement is small, the relative improvement stops increasing quickly.
So, now let's look at an example of this applied to an image processing problem of denoising an image, So, here, we have a very noisy image, I'm not sure how noisy it would show up for you but it's a noisy

image. And how we're going to denoise this image is by extracting eight by eight patches, where eight by eight is arbitrary. It could be more, it could be bigger, it could be smaller, but we'll say eight by eight. We then vectorize that into '64' dimensional vector. So, for this eight by eight patch, we extract it, we vectorize it in to be '64' dimensions, and put that along a column of a matrix. And we do that for every patch, so we get— we should get many patches and that constitutes this matrix 'X', which is 64 by—for this image, if there are... there are this many pixels, so roughly this many pixels, so there's about two, 250,000 of these columns in the data matrix. Then for each, then what we do is we factorize using probabilistic PCA to approximate 'x_i' as the product of 'W' with the conditional posterior mean Mu 'i'. So, we're saying that this 'X' is approximately equal to 'W' times 'Z'. Remember, we don't learn 'Z', we learn a conditional posterior of it and so when we learn this model, we learn the conditional posteriors, we learn a point estimate at 'W', we then replace 'Z', each column of 'Z' with the posterior mean Mu. And then we approximate 'x_i', which is the i^{th} column of this matrix as 'W' times Mu 'i'. So, we take... we take this matrix and we approximate it with this matrix, where now we have 'W' Mu, and then we take the i^{th} column here and put it back into the image. So, we take out the noisy patches, we replace it with the un—denoised patches, and then we reconstruct the image. And so, where has all the noise gone? It's gone in the noise term of the generative model. So, the original model has this as our expression of the point 'x_i'. We're assuming this a denoised mean and then here is where all the noise is contained. And so, when we reconstruct the image and see what it looks, like we see—we get this.
So, this is where we take the noisy patches, we replace them with the denoised patches and we get a denoised image on the right. Here's another example using missing data. It's a very extreme example. So, here's a 480 by 320 by three image, so the third dimension is because it's a color image, so RGB; 80% of those values are thrown away completely at random. And then we've extracted an eight by eight by three cube from many of those—many, many of those from this image. And then we filled in the missing data using EM. We've—after, filling in the missing data, we construct the image and we get this and you can compare it with the original image without the missing data and it's very close. So, we learn all the missing pixels in here. We fill them in using EM and we get this. So, it's an extreme example but also a clear example of how actually we have plenty of information in this missing image in order to get a very good sense of what the true image looks like.

Video 5 (10:04): Kernel PCA I

And so we've seen how, when we have an algorithm that uses dot products, so if we have an algorithm, that is represented purely in terms of the dot products of our data points, then we can generalize the algorithm by replacing the dot products with a kernel. This same exact generalization can be made with PCA. So let's see how that's done. So recall with PCA, that what we do is we take each data point, 'x_i', we take its outer product, so that's—we haven't gotten to the inner products yet. So we take the outer product of that data point with itself and sum them up and represent, that as this matrix product. So each—so the matrix 'X' had each 'x_i' along its column, we took 'X' 'X' transpose, and then we found the eigenvectors of this matrix. So now let's map the data to higher dimensions. So it's funny that we're discussing a dimensionality reduction technique, and now we're going to map the data to even higher

dimensions before doing dimensionality reduction, but hopefully in later slides it will be very convincing that this is actually a smart thing to do, that actually projecting to much higher dimensions before projecting to a much lower dimension can do even more for us.

So let's, let this Phi, as usual, be our projection of the data point 'x' to a much higher dimension from 'd' to capital 'D', where capital 'D' is much greater than little 'd'. And now with this higher-dimensional projection, what we want to do is, learn the eigendecomposition, of this. So notice, originally we wanted the eigendecomposition of the sum of these outer products, which we saw could be represented as, constructing this matrix. So in a previous slide, we showed how we can take this matrix times an eigenvector 'q', and that's equal to the k^{th} eigenvalue times the k^{th} eigenvector. So previously, we didn't use the subscript 'k'. However, there are 'k' to these pairs— there are many of these pairs that we can learn, and we sort them by decreasing order in terms of Lambda. So, now we want to find the vector 'q', and the eigenvalue Lambda, for the matrix from this higher-dimensional projection. So instead of having this original dimensional data, we project it to higher dimensions, then we have this problem. So now how can we do PCA in this higher-dimensional space? How can we learn 'q' and Lambda in even higher-dimensional space? For example, with Gaussian kernels, this is an infinite dimensional space.

So we want to see how we can sidestep that issue. So here's where some of this lights of hand come into play, the tricks, the mathematical things that we can do to fix this problem. So let's just reorganize the operations of the eigendecomposition. So all I've done is taken the previous slide, remember, '$Lambda_k$' was originally multiplied against the k^{th} eigenvector. Now, I'm bringing '$Lambda_k$' onto the left side by dividing it. So I've divided both sides by '$Lambda_k$'. And also originally on a previous slide. So I divided both sides by '$Lambda_k$'. Also, originally, I viewed this term first, I completed this term first, and then multiplied by 'q_k'. Now I want to switch the orders. I want to multiply my higher- dimensional projection of 'x_i', with the k^{th} eigenvector, divided by the k^{th} eigenvalue. And now I'm going to call that 'a' sub 'ki'. So 'i' is indexing the data point, 'k' is indexing the eigenvalue, and the eigenvector pair.

So now notice, that what I've done, and it's not clear why this will help me yet, but what I've done is shown that I can actually represent each data point, each— I'm sorry, I can actually represent each eigenvector 'q_k' as a linear combination of my data projected at the higher-dimensional space. So I can take my i^{th} data point, projected it to the higher-dimensional space, multiply it by 'a_{ki}', where 'i' is, index is paired with a data point, sum it over each of my end data points, and that's my k^{th} eigenvector. So I've just shown that I can— there exists some value for 'a_{ki}', such that I can represent it like this. Now I'm going to write 'a_k' as a vector in R^n. So 'a_k' now corresponds to the k^{th} eigenvector, 'q_k', and it's an 'n' dimensional vector, where n is the same dimension as the number of data points that I have. So now the trick comes into play where instead of learning 'q_k', we're going to learn 'a_k', So instead of learning this potentially infinite dimensional vector, we're going to learn and 'n' dimensional vector corresponding to the k^{th} eigenvector. Okay, so how do we do this? So now let's take 'q_k' out, and let's replace it with this representation of it. So 'q_k' appears up here. What we're going to do is we're going to take 'q_k' out of this, both of these terms, and we're going to replace it with this term right here, and we're going to bring '$Lambda_k$' back over to the right-hand side, and we end up with something that looks like this. So it looks more complicated, but eventually, it's going to make our problem even easier. And so now notice, that we have finally the dot products we're looking for, we took this value—this representation of 'q_k' and put it here and here and brought '$Lambda_k$' back over here, and we had dot products.

And so here is the first place where we can take these dot products and the higher-dimensional mapping and replace them with a kernel function between the i^{th} and the j^{th} data point. So we're making some progress. The next step, is to multiply the left, and the right-hand side both by the projection of the l^{th} data point for each of the 'n' data points. So we take this vector, Phi of 'x_l', It's a capital 'D' dimensional vector corresponding to the l^{th} data point, and we construct a matrix, where we have this along the l^{th} column of that matrix. So in a sense, what we can say is we're making a big matrix Phi, which is equal to the matrix of Phi of 'x_1' to Phi of 'x_n'. And now we multiply Phi transpose, against both sides like this. And now we're going to see that, that's going to simplify our problem.

So when we take each of these higher-dimensional mappings, transpose it, and multiply it on both sides of this equality, we're going to end up also with an equality that looks like this, and so, here is where we got—where we were able to introduce one kernel by multiplying this on the left side of both of these sides, we're going to get the other kernel. And so I'm not going to work it all out in mathematical detail, but what you can see is that, when you do that multiplication, when you construct that matrix Phi, which is, again, Phi of 'x_1' to Phi of 'x_n', and you take Phi transpose and multiply it by both sides like this, you get this equality. You get the kernel matrix squared times 'a_k' equals 'Lambda$_k$' times the kernel matrix 'a_k'.

And so now the kernel matrix 'K' is an 'n' by 'n' matrix constructed on the data. So 'K_{ij}' is the kernel between data point 'x_i' and data point 'x_j'. Now let's multiply both sides by the inverse of 'K', which is a positive definite matrix, and we get this equality. So we've done a lot of work, perhaps it's not clear, you know, how this relates to the original problem, maybe it's a bit too many steps. But we can work through it, and find that equivalently we want to find this eigen...eigendecomposition problem. Notice that, we're taking the matrix kernel 'K' now, it's 'n' by 'n', and it's that kernel times the vector 'a_k' is equal to the eigenvalue 'Lambda$_k$' times the same vector 'a_k'. So what we've shown, without ever really solving it, we've shown that whatever, that this 'a_k' that we've defined on a previous slide here, we defined this to be 'a_k', is now the eigenvector associated, the k^{th} eigenvector associated with the kernel matrix, constructed among my data. And so, again, originally, I could have mapped the data into such a high-dimensional space, like an infinite dimensional space, and now I'm representing it as a eigendecomposition of an 'n' by 'n' matrix, where 'n' is the size of my data.

Video 6 (08:06): Kernel PCA I

So, in short what kernel PCA does is it takes in the data, 'x_1', through 'x_n', so our 'n' data points, each one is in R^d. It defines a kernel function between all the data points. For example, the Gaussian kernel could be this one, so 'K_{ij}' would be the kernel between observation 'x_i' and 'x_j' which is equal to this equation here, if we use the Gaussian kernel. We construct this kernel matrix on our data and 'n' by 'm' matrix, so it's pair-wise between all of our data points, and then we solve this eigendecomposition by calling the function built into our program, whatever programming we're using, to get the eigenvector, and the eigenvalue, for that kernel matrix and we do it for the first 'r' eigenvalues and eigenvector pairs, whatever 'r' is. So 'r' is the dimensionality we want to project the data into, if it's—if we want to project it into three dimensions then 'r' is equal to three.

We get the first three eigenvalues, and eigenvectors of the kernel matrix 'K', and now we output a new coordinate system where we've taken our vector 'x_i', implicitly we've mapped it, up into this high-dimensional space, and learned 'r' different eigenvectors there in that higher-dimensional space, and then projected that data back down into the lower-dimensional 'r' dimensional space. But if you go back and look at the derivation, this projection is equal to this. So we learn, for the K^{th} dimension of our projection we learn the K^{th} eigenvector and eigenvalue. And then, for the i^{th} data point we look at the i^{th} dimension of the K^{th} eigenvector and multiply it. So the first dimension will be the i^{th} dimension of the first eigenvector of 'K' times the first eigenvalue and so on, and this is our low-dimensional mapping. So before I show this on an example, an immediate question is, how do we handle new data? So if a new 'x' now comes in how do we project that into the lower-dimensional space using exactly the same assumptions as the model was using.

So before, what we did, if you recall is we learned the eigenvectors, for example, if we wanted to project it using the K^{th} eigenvector 'q_k' we simply took the new data point 'x_o' and then multiplied it by 'q_k' and that's now equal to the K^{th} dimension of the lower-dimensional projection. So we had 'K' equals one to 'r', we did that dot product 'r' times, that's our 'r' dimensional projection. But now, Alpha, 'a_k' is different here. So how do we project data in this case? So first, let's recall the relationship of the vector 'a_k' with the eigenvector 'q_k' in kernel PCA. So we showed that the K^{th} eigenvector in the higher-dimensional space can be written as 'a_{ki}', so the 'k' is indexing the eigenvector, 'i' is indexing the data point. So 'a_{ki}' times the vector we get by mapping 'x_i' to a higher-dimension and then summed over all of our data points. And then we use the kernel trick to avoid having to work with or even define any of these higher-dimensional projections. We realize we can have only dot product representations that just use a kernel function.

So now, as with regular PCA after we map 'x_o' the new data point to the same higher-dimensional space we want to project it into the eigenvectors. So we take 'x_o', map it into the higher-dimensional space by the function Phi, take the dot product then of that higher-dimensional mapping with the first eigenvector and that becomes our first dimension in the lower-dimensional space. And we do that 'r' times to get our 'r' dimensional projection. But now if we plug in for each of these 'q's', if we pick the K^{th} dimension here so 'q_k' and we plug in what this is equal to. So this dot product will equal the K^{th} dimension in this projection. But now we take this and plug it in, what we get is that this value is equal to the sum over all of my 'n' data points of 'a_{ki}' which we have, that's– we read this off from the K^{th} eigenvector of the kernel matrix constructed on the 'n' data points times the kernel that I evaluate now using my new data point and all of the old data points in my old data set.

So I evaluate the kernel between a new incoming point with all the points–of my 'n' points in my endpoints in my data set. I multiply that kernel by the eigenvector I got from the matrix, the original matrix 'K', sum it up and that's my projection now. Okay, so let's look at an example of this. So here we have a data set in two dimensions, and the three rings are color-coded just for visualization purposes. Now, if we wanted to project this three dimension—this two dimensions into two dimensions using PCA there would be no projection we would simply have every data point is equal to itself, essentially. Perhaps there would be a rotation, but the data would essentially have the same relation to each other. Now, if we want to somehow separate these three rings for each other what we might want to do is take these data points and project it into a higher-dimensional space and then do PCA there. And then

that's going to allow us to perhaps separate these three rings from each other, and so here's what we did, that's what we did here. So what these plots are showing is taking each of these data points and calculating the kernel matrix, the point–pairwise kernel matrix using a Gaussian kernel, and then using that 'n' by 'n' kernel matrix and projecting onto its first two eigenvectors and eigenvalues, and we get something like this.

And so of course depending on the kernel width you'll get different results, I'll just focus on this term which has a good kernel width. And what we see is we took each of these points and mapped it into a much higher-dimensional space that corresponds to the Gaussian kernel, projected it back down into the same two dimensions and we've essentially unwrapped our three rings here. So we've actually projected from R^2 back into R^2, except in this projected region we went up to first an infinite dimensional space then project back down into R^2 and we've unwrapped our data. So there's some– before, I end this lecture I want to quickly discuss some areas of research that discuss–that use this. So what we've done is closely related to something called spectral clustering, so that's where clustering maybe in this space isn't as good as clustering in this space because we want to perhaps cluster the rings together.

So in this way we can cluster these three rings together more easily than we could here. It's also an example of something called manifold learning and so this would be an example of a manifold that we want to somehow unwrap. So that Euclidean distance while used in this space is not ideal Euclidean distance in this space somehow is more useful. So K-means clustering here with three clusters would not learn the three rings, but perhaps K-means clustering here would learn these three parts right here.

Video 7 (01:53): Sequential Data

In this lecture, we're going to move on to another type of unsupervised learning. It's the last of the three main types, the first being clustering, the second matrix factorization, and now we're going to consider sequential data. So, these are going to be models, where we take sequences into account or somehow model some sort of a latent sequential behavior. Up to this point in this course, when we've been thinking probabilistically, we've been working in the *iid* setting. So, this is where we've assumed that every observation that we get is independent of every other observation, and identically distributed. Often, it's a reasonable assumption to make and the results will be very good. But it's also something that's done for convenience.

In many applications, this assumption is clearly a bad one. For example, to give a few examples, if we wanted to model rainfall as a function of hour, we wouldn't want a model that as an independent random variable. If we wanted to model the daily value of a currency's exchange rate, that's not independent in the day. So, the value today is dependent clearly on what it was yesterday. If we wanted to model speech, via some acoustic features that we extract, for example, here I show the— a signal that's where somebody saying Bayes' theorem, and then we extract some acoustic features. Clearly, if we look at column slice, there's a temporal dependence that exists in the data and we would want to take that into consideration when modeling speech.

Video 8 (07:15): Markov Chains

In this lecture, we're going to begin a discussion about sequential models, specifically, Markov models or Markov chain, and we'll discuss the simplest case of a Markov model, which is a first order Markov model. So let's begin the discussion on Markov chains, so here's a toy example, often this example in textbooks is introduced with the story of a drunk walking down an alley, so we're going to modify it a little bit. And so, let's imagine that we see a zombie walking in an alley and each time it moves forward, it takes a step, but because it's staggering it can move to the left, it'll take a step straight ahead, or it'll take a step to the right. And the probability of stepping left, straight, or right has probability 'p_l', 'p_s', or 'p_r', except in the case, where it's right up against the wall, in which case it's going to step straight with probability 'p_s^w' or it's going to take a step towards the middle of the street with probability 'p_m^w'.
So in this case if we model the walk...the random walk in this case, we're also going to make the simplifying assumption that the distribution on the next location only depends on the current location, and so, what we want to ultimately do is model the location in the street, width-wise, so where is, not necessarily length-wise but where is this random walker in relation to the wall? And we're only going to assume that the next location is dependent on the current location. So let's simplify the problem, by assuming that there are only a finite number of states. So if we wanted to measure the actual distance for example from the left wall as a function of time, that's going to be a continuous random variable. But we're going to discretize it and assume that there are only a finite number of states that these, this random zombie walker can be located in. In this case, we're going to add a latent variable 'S', which gives the position as a function of time, of the random walker. So for example, if at time 't', the walker is in state 'i', for example, this position one could be 'i' would be one, two, three, or four, etc.
So, let's discretize the state by the index of the position. So this would be the 20th position. And so now we want to say what's the probability of the position at time 't' plus one, or step 't' plus one given that at step 't' the position is 'i', and for simplicity, in this slide, let's just assume that the walker is away from the wall, then what this is...what this model would say, is that the position at 't' plus one given that we're in position 'i' at time 't' is going to be equal to 'i' plus one with probability of taking a step to the right. It's going to remain in the position 'i' with probability 'p_s', which is the probability of taking a straight step, or the position will be 'i' minus one with probability 'p_l', which is the probability of taking a step to the left. So this is a simple Markov chain. And, because the distribution of the state at time 't' plus one or any time only depends on the position at the previous time, we say that this, Markov chain has the first order Markov property. So, it would be first order Markov chain, if my position at the next time point is only dependent on where I'm at currently.
If my position in any given time is dependent on the previous two times, the previous two locations, that would be an example of a second order Markov chain. Most of the time when people are working with Markov chains, they're working with first order Markov chains because the added complexity of working with higher orders is very difficult, computationally. So, most of the time we'll make a first order Markov chain if we're going to use a Markov model. Now we can put the transitions, these probabilities into a matrix and that's also a very common way to represent a Markov chain. So, let's introduce this notation, the matrix 'M' which we're going to call Markov transition matrix, or a random

walk matrix, there are a few names we can call it. And, for simplicity let's imagine, that we have six different positions in the, in this alley that this zombie is walking in, and these are called 'states'. So every single position now is going to be called a 'state', that's a typical name for it, then we can represent the transition matrix in this way.

If we let the six positions be ordered based on where they are relative to the wall; so this would be the position next to the left wall, this would be the position next to the right wall, and these would be all the positions, the four positions in between, we would have a Markov transition matrix that looks like this, given that, I'm at one position, for example, the third position, I go to the first position with the probability of walking to the left, I stay at the third position with the probability of walking to the right, walking straight, or I go to the right position with the probability of walking to the right. And based on this simple modeling assumption that we're making for this toy problem, there's no probability of skipping other steps.

So for this specific Markov chain, if we're in the third position, we have zero probability of moving to the first or the fifth or sixth positions. And so we get something like this, we can represent this random walk matrix, this random walking zombie, or drunk if you will, with this sort of a matrix if there are six different positions. And, of course we can we can permute the rows and the columns as long as we know how to map back to a position. So, if we want to say the third state is equal to the location next to the right wall, we could do that, we just have to have a coherent, correct way of permuting these probabilities so that they relate to reality. Okay, but now we're just, we're still working within a toy example and in reality we aren't necessarily going to know these things.

Video 9 (09:36): First Order Markov Chain

So now, we've defined the latent variable for a Markov chain, which is a sequence of states. 's_1' through 's_t'. So, we have a state 's' is a value between one to capital 'S', which is an index of the finite...the finite number of potential states that we can be in at any given time, and then a sequence of these states, 's_1' through 's_t', so sequence of length 't' of states where each entry in this 't' dimensional vector is a number between one and 'S'. It's called a first order Markov chain. If the joint distribution of that sequence can be written as follows, so we have a joint distribution, of a sequence of length 't'. We first write that using the chain rule of probability. So, this is always true in every case. We can say that the probability the joint distribution is, is equal to this product of conditional distributions, where we have first a distribution on the first state, and then a product over the subsequent states of the probability of this state at time 'u', given all of the states up until time 'u'. So this is always true, we can always condition on more than necessary and say that this is a way of representing this joint distribution.

But now, the first order Markov chain makes the additional assumption that this conditional probability distribution can be written this way. So the probability of where I'm at at any time point 'u' is not conditioned on everything beforehand, But only on the previous time point. So the distribution of where I'm at at time 'u' simplifies to being only conditioned on the distribution of where I'm at on the location, the state that I'm in the previous time point. So notice that this is a very simple modification of the iid assumption, for the iid assumption we said that the joint distribution of 't' random variables, is equal to

the product of the distribution of each one individually. So this joint distribution, when we make an iid assumption, simplifies to a product of the distribution of each one separately or independently of any other point. Now the Markov, the first order Markov assumption simply says that the distribution on a given time point, is conditionally dependent on the value of the sequence at the previous timepoint. So this is the simplifying, this is the assumption, the distribution assumption made by a Markov chain, on a sequence of random variables.

It's a very simple modification but the algorithms that will accompany it and the applications that we can put it to will be fundamentally different. So, we're going to have fundamentally different modeling capabilities with this type of a model assumption compared with this type of model assumption. So now, to see how these conditional probabilities map to the Markov transition matrix, we can simply write it this way, that the probability of transitioning from state 'i' at time 't' minus one to state 'j' at time 't' is equal to the ij^{th} element in this transition matrix. So this matrix, if there are capital 'S' total possible states that the chain, the Markov chain can be in, then this an 'S' by 'S' matrix, where we're going to assume that every single row is a probability distribution. So, every row has non-negative values along that row, and they each sum to one, and the j^{th} column in the i^{th} row is the probability of making a transition to state 'j', given that I'm in state 'i'.

And so we make this assumption that the probability distribution I'm transitioning between every single state is not dependent on time. to make for the first order Markov chain, that the probability of transitioning from state 'i' to state 'j' is the same for all time points 't', and so now given a starting state, 's_o', we can actually generate a sequence, a first order Markov chain, on 'S' states using this 'S' by 'S' matrix as follows, we're given our starting state, we continue to generate new states where at time 't minus one' we generate the state at time 't', using a discrete distribution where we pick out the row of the transitional matrix 'M' indexed by the state that we're in at time 't' minus one. So given what state we're in at 't' minus one, that indexes the probability distribution that we're going to pick from 'M', to then transition to the state at time 't'. And so this 'S_t' will be equal to 'k' with probability equal to the k^{th} entry in this row that is picked out by the state of 't' minus one. So, that's the basic model of a first order Markov chain, it's very simple, very straightforward, but now we have a few problems. First, of course as usual we make the modeling assumption, where we assume we know the model parameters, and then we get the data.

The sequence in this case of state transitions, but then in reality what we have first is the data, the sequence of states, and now we want to go back and infer the latent variables or the model parameters. So in this case, that's the Markov transition matrix. So let's look at a simple maximum likelihood, it's very intuitive. What we want to do is, we want to find the transition matrix 'M' that maximizes the probability of an observed sequence, and we know that we can represent this by taking the log of this likelihood, and maximizing over 'M', and given the Markov assumption, what that amounts to, is maximizing over a sum of every single transition's likelihood. So here we're summing over every single transition. We have a sum over the index at the states at two different time points. So given we're at time 'u' and we're in state 'i' and we make a transition from state 'i' to state 'j', where we're now at time 'u' plus one, we have to now sum over all of those possible indicators to pick out the correct event, and then, that is going to pick out the correct log probability. So this is the log joint likelihood of an individual sequence, given 'M' and now if we just take the derivative and maximize subject to the constraints that

'M' only has non-negative values, and that each row has to sum to one, we can find that the maximum likelihood transition matrix, for transitioning from state 'i' to state 'j' is equal to the sum of the total number of times in our data set or in our sequence that we make this transition.
So, we are summing here an indicator of transitioning from state 'i' at time 'u' to state 'j' at time 'u' plus one. We're summing over every single time point, and so this numerator is summing the total number of times we actually make the transition from state 'i' to state 'j', and then the denominator is summing the total number of times that we are actually in state 'i' to begin with, which would then give us the total number of times we transition from state 'i' to any other state. So, this is counting the number of events of going from 'i' to 'j'. And then, this is normalizing so that when we sum this term over 'j', we sum to one. So it makes a lot of sense. It's very intuitive the maximum likelihood solution for the Markov transition matrix. For example, if we wanted to model the probability that it rains tomorrow given that it rained today, all we would do is get a sequence of events where every single time point would be a day, and we would have an indicator of whether it rained or not in that day. And then, we would sum the total number of times, that it rained two days in a row. And then, we would divide by the total number of times that it rained, period. And that would be the fraction of times that we saw rain two days in a row.

Video 10 (11:44): State Distribution and Stationary Distribution

So now, let's look at some more fundamental properties of Markov chains, and we're going to ask some questions. First, we're going to ask, can we say at the beginning of generating data from a Markov chain which state we're going to be in at time 't' plus one. So we start at time one or time zero however we index it and then we start the Markov chain by generating data from a Markov transition matrix 'M', and we want to say, if I do that and go to step 't' plus one can I say, where I'm going to be, what state I'm going to be in a 't' plus one, at time 't' plus one before I even start this entire process. So, we can do this, we can use basic probability calculations to find what this is. So let's imagine that first we're at step 't', so we're imagining we're at step 't' right now, and we have a probability distribution, we're able to somehow figure out standing at time one what the probability distribution is of where I'm going to be a time 't'. Let's call this 'p' this probability of being in state 'u' at time 't', so we're going to call it that probability this and we're going to pretend like we know what this is.
In that case, the distribution on where I'm at, at 't' plus one. So, if we want to find the probability that I'm in state 'j' at time 't' plus one, I simply write that as a marginal of a joint probability of being in state 'u' at time 't' and in being in state 'j' at time 't' plus one and now sum over all values for 'u' to integrate or sum out this...this state variable at time 't'. So I represent this marginal probability as a marginalization of this joint probability, but then I take this joint and write it as conditional times prior. So I can write this joint probability as the probability given that I'm in state 'u' at time 't' of transitioning from state 'u' to state 'j' at time 't' plus one, times a prior probability of being in state 'u' at time 't' to begin with. So, I multiply these two things together, I sum over all of the states for time 't' and I get my marginal distribution of where I'm at, at time 't' plus one. So let's now change notation, just change the notation and let 'w_t' be a row vector of length 's' that gives me a probability distribution on the 'S'

different states at time 't'. So 'w_t' is my marginal probability of which of the capital 'S' states I'm in at time 't'.

Then, we can say that this probability is the vector 'w_t' plus one evaluated at dimension 'j'. We can say that this prior probability is the vector 'w_t' evaluated at dimension 'u', and now this probability of transitioning from state 'u' to state 'j' because we're assuming that the probability is not changing with time, is simply the element, the ujth element in the Markov transition matrix 'M'. So this probability, we can read off by looking at the uth row and the jth column of the matrix 'M'. And so now we have vector matrix...a vector matrix way of writing this that's time 't' plus one. My 'S' dimensional...capital 'S' dimensional row vector of the probability of where I'm at that time is equal to the same probability of where I'm at except at time 't' times the matrix 'M'. So that's what this sum is doing, it's taking this vector times this matrix. It's just two different ways, two different notations to represent the same thing. So we take the row vector 'w_t', we multiply it by 'M' and we get the distribution at 't' plus one, and so we have this iterative way.

For example, we know that 'w_t' can be written as 'w_t' minus one times 'M'. And if we keep stepping backwards, then once we arrive at the state that we're in at time one, we can say the state that I'm in at time 't' plus one is equal to the state that I'm in at time 't', at time one times the transition matrix raised to the power 't'. And what this notation is saying, is it's important to know it's not an element wise exponential, so we don't take each element in 'M' and raise it to the power of 't'. We have 'M' times 'M' times 'M' and this happens 't' times. And so now, if we know what state we're in at time one, for example we know where we're starting a time one we can let this vector 'w_1' be a vector of all zeros except for a one placed in this dimension corresponding to the state we're in, which is simply another way of writing a probability on my state that I'm in now that is deterministic. If it's probability one of being where I'm at now then it's a guaranteed thing, and so this vector will then pick out the correct row of this 'M' raised to the 't' and that will be the probability that I'm in any given state at time 't' plus one, given that I'm in a particular state of time one.

So now we've given away to represent my probability of where I'm at, at time 't' plus one, given my probability of where I'm at, at time 't' it's simply this vector-matrix product using the same Markov chain and then we know how we can unpeel this to represent it as a starting vector times a power of 'M'. The question is now, what's happening as 't' goes to infinity, is anything interesting happening, is it converging to anything important? So we're going to define this distribution 'w' infinity which is a limit as 't' goes to infinity of 'w_t'. So this is the distribution on where I'm at an infinite number of steps from now. This 'w' infinity has a name and it's called the stationary distribution of a Markov chain 'M'. So the stationary distribution of the Markov chain 'M' is equal to 'w' infinity according to this notation. So, there are many technical results that can be proven about this, we're just going to recite a couple of properties about this. So, if the following two conditions are true, then the important thing to know is that this 'w' infinity will be the same vector no matter where I start out. So, given any state that I start out at 'w' infinity my distribution on where I'm at an infinite number of steps from now is the same in all cases. So, just telling me where I'm starting is not going to help me say anything about where I'm going in the infinite distance.

But this is only true, if first we can reach any state in the Markov chain by starting from any other state. So that says that if I start in a particular state and then I transition according to the Markov transition

matrix 'M' at some point I can eventually reach any other state no matter where I start. Also, the sequence can't have any sort of hidden loops where we're converging between states in some deterministic way. So, there can't be any sort of deterministic convergings, transitions for example, I always transition, for example I have a state like this, three states like this and I always transition like this between these three states in a deterministic loop, we can't have anything like that encoded in 'M'. In...in those two cases if those are true, then no matter where we start we're going to converge to the same stationary distribution, we're going to have the same uncertainty or the same level of belief of where we're going to be an infinite number of steps from now no matter where we start.
So, if we have converged with stationary distribution and we've shown that we have this sort of a representation where 'w_t' plus one is equal to 'w_t' times 'M'. And we...and 'w_t' is converging as 't' goes to infinity to a vector called the stationary distribution. what that means is that 'w' infinity is a vector that has this property where if we take 'w' infinity times 'M' we get 'w' infinity back. So we're looking for a vector having this property to be our stationary vector. And this is closely related to the eigenvectors of the transpose of the Markov transition matrix. So remember that 'q_1' and '$Lambda_1$' are the first eigenvector and eigenvalue of a matrix 'M' transpose. If 'M' transpose 'q_1' is equal to '$Lambda_1$' 'q_1', so this is the property of an eigenvalue and an eigenvector and the first one is the maximum value for '$Lambda_1$'. So there would in general be many of these pairs, but the first one is the one corresponding to the maximum value for Lambda. And so it can be shown that for this specific matrix, if we have a Markov transition matrix, if we have a matrix 'M' such that all the values are non-negative and all the rows sum to one, then the first eigenvalue of the transpose of that matrix is equal to one. And the first eigenvector relates to the stationary distribution in this way.
So remember by definition, the eigenvector has to have an 'L_2' norm of one. And so we can get the stationary distribution, which has an 'L_1' normal equal to one. So we square—'q_1' we square them, sum them up, that equal to one, with this 'w' infinity we want to sum them up without squaring them then have that equal to one. We can recover the stationary distribution by taking the first eigenvector of 'M' transpose and normalizing it. And so, there's a clear relationship between eigenvectors of this matrix and the stationary distribution. So in practice what that means, is that when we want to know what the stationary distribution is, we don't simply keep multiplying 'M' by itself until we hopefully get close enough to some level of convergence, what we can do is take the transpose of 'M' and call a built-in function to find its first eigenvector and then perform a simple normalizing of that eigenvector to get the stationary distribution.

Video 11 (13:45): Ranking Algorithm

Now in the rest of this lecture I want to discuss two different applications of Markov chains. The first is an example of ranking objects. And so this example is going to show how we can use the stationary distribution of a Markov chain that we construct on our data to rank objects. The specific data that we're going to consider for this type of model consists of pairwise comparisons. So imagine we have 'S' objects or 'S' teams, 'S' teams or 'S' athletes that are competing against each other. Or we could have 'S' objects being compared where users are selecting their preference between two different objects. Our

goal now is to rank the objects or rank the teams or the players from best to worst using a Markov chain. So that means that we need to somehow find where a Markov chain can be applied to this problem. The way we're going to do this is by constructing a random walk on all of the objects. So let's focus only on teams for now, if we want to rank sports teams. Then every team is going to be one state in the Markov chain.

So if there are '1,000' teams that exist, there are going to be '1,000' different states, and so our Markov transition matrix 'M', will be '1,000' by '1,000'. We want to construct this transition matrix so that the stationary distribution of that matrix first exists, and secondly can be interpreted as telling us who the best teams are, and who the worst teams are, and also give us a degree of everywhere in between. So notice, without telling you yet, what the model actually is, and how we construct this, that we aren't going to actually have a sequential model of the actual events. So that's the thing to pay attention to, that even though we've motivated Markov chains as being a sequential model, the way that we're going to construct this matrix does not take time in to consideration, that we're actually using the Markov property as a modeling, an artificial modeling, construct where we can now interpret the stationary distribution of a Markov chain as giving us something meaningful that doesn't— but the Markov chain that we're constructing doesn't actually correspond to anything meaningful in reality.

So this is where an abstract sort of model is very useful where we kind of—We don't enforce that it has to actually correspond to something happening in reality. Okay. So we want to construct a Markov chain between these objects, again, we're just going to focus on ranking teams because it's intuitively very easy. So we want each team now to be a state, and we want to construct a probability of transitioning from every team to every other team. And the way we want to construct these probabilities is that we want to encourage teams to transition from the losing teams to the winning teams. So we want to move to teams that win more, we want to have a higher probability of moving to teams that win, and a lower probability for moving to teams that lose. For example, one way that we can do this problem is we can construct a transition matrix, where we can only transition between teams if we have a game played between those two teams. So if two teams don't play each other, then we have no information about how they relate to each other directly, and so we aren't going to have any probability of making a transition from a team to another team. So, for example, if team 'A' beats team 'B', then we're going to create a link between state 'A' and state 'B' or team 'A' and team 'B' where we now have the potential for transitioning between those two teams in either direction. Except now we want to make a probability of transitioning from one team to another, and we want the probability of transitioning from 'B' to 'A' to be high if team 'A' beats team 'B'.

So if 'A' beats team 'B', then the probability of transitioning from 'B' to 'A' should be high, relatively speaking, and the transition from—the probability of transitioning from team 'A' back to team 'B' should be low. So it's not a symmetric probability. We have more probability of going from 'B' to 'A' than we do from 'A' to 'B' if team 'A' has beaten team 'B'. And there are many ways we can create rules about these probabilities. One thing we can do is take in to consideration the score of the game. So if it's a close game, that will make it a weaker skewing, whereas if one team beats another team by quite a bit, then the transition will be very biased. Going from 'B' to 'A' will be much higher than going from 'A' to 'B'. So these are arbitrary modeling choices that can be made. So, before I show you the results of doing something like this on real data, I want to give just my own arbitrary choice of how to score these things.

So this is what I've chosen to do. What we do is we initialize a matrix 'M' hat to be a matrix of all zeroes. So if there are '1,000' teams, then it's '1,000' by '1,000' matrix of all zeroes. Then we iterate through every game one time to construct the transition matrix as follows. If we have a particular game between team 'j_1', so this is the index of team 'A', 'j_1', and the index of team 'B', say, is 'j_2'. So imagine we have a game between team— with index 'j_1' and team with index 'j_2'.

Then we update the transitions in this way. So we want to update the probability of transitioning from team 'j_1' back to team 'j_1'. So this is a self transition. If I'm in state 'j_1' now, then I can still at the next time point stay in state 'j_1'. So you notice in all of these Markov matrices that we've been discussing we have not assumed that the diagonal is equal to zero, that the diagonal can also have some probability. In that case, we have a self transition. So we say that team 'j_1' can now update the weight of transitioning to itself by taking the old weight, adding to it an indicator that it won the game. So this is either zero or one, depending on whether that team won the game, plus, for example, we can add the fraction of the total points that it had. So this—if it's a close game, this will be close to point five. If it's a blowout, this will be close to one. It's a fraction of all the points scored in the game that were scored by team 'j_1'. So we update the self transition by taking the old value and adding to it these points. Similarly, we update the transition between team 'j_2' to itself by taking the old value and adding to it an indicator that team 'j_2' won. So either one of these two events will be true, and the other will be false, plus team 'j_2''s fraction of the total points. So these are updating the weights of the self transitions. Now to update the weights of transitioning from team 'j_1' to team 'j_2' we take the old value. So if they've already played each other, then this will be non zero. If they haven't, then this will be zero.

We add to it an indicator that team 'B' wins. So if team B corresponds to 'j_2', then we want to increase the weight of transitioning from 'j_1' to 'j_1' by adding a point one to that transition plus the fraction of points that team 'J_2' scored. And then transitioning from team 'j_2' to 'j_1' takes the old value, adds to it an indicator that team 'j_1', team 'A', with index 'j_1', won, which now increases the weights of transitioning from 'j_1' to J_2 because 'j_1' beat J_2, plus the fraction of points. So we just manually do this. I've made up these points, but it's a meaningful Markov chain, where now...we tally up all of these points, we then construct 'M' by normalizing every row so that it sums to one. And now we can intuitively imagine that there's more flow towards the good teams. If I imagine in my, you know, head walking from team to team, what's going to happen is the teams that won a lot, and win by a lot, are going to be transitioned to with higher probability, and are going to stay to them. So it's easy to think for a while about why this could work.

For example, if a team wins a lot of games, but plays against very bad opponents, then if we transition to those opponents with high probability, we'll go to that team that wins a lot of games. But if a particular team has only beaten very bad opponents, then we're never probably going to make it in our random walk to those bad opponents in order to get to that good team. So this is where some of the intuitions have to come in to see why it's not enough to simply win a lot of games, that you have to also win a lot of games against other teams that win a lot of games. In that case, you're going to transition to those other teams more frequently, which means because you beat those other teams, they'll transition to you more frequently. So the stationary distribution can encode the probability of which team I'm going to be at an infinite distance from now. And we can interpret the highest probability team or the highest probability state as corresponding to the best one.

The second highest probability team to corresponding to the second team. And so on. So to give you an example of this on real data, what we have here are all of the college basketball games from the '2014' to '2015' season. So in total there were '1,549' teams. So I'm assuming that this is taking in to consideration all divisions of basketball. So there are this many different college basketball teams that are in the data set. In total there were '22,033' different games that were played between two teams throughout the entire season. So this is the scale of college basketball. I created a Markov chain exactly like on the previous slide, constructed the stationary— I found the stationary distribution by finding the first eigenvector of the transpose of that chain, and normalizing. And then ranking the teams based on the probability in that stationary vector. So that's what's shown here. These are the '25' most probable teams. This score is the probability of being at that team an infinite number of steps from now. So you see that even though— that even though this is the best team, the probability is still extremely low. So the probability of being at the team Villanova is very small, but still more than every other team. And then here on the left what is shown is the 'USA Today' coaches poll. So many coaches were asked to give their belief of what were the top '25' teams, and this is expert opinion. And all of their polls are taken, combined, and a final top '25' ranking is given based on their collective rankings. And so you can see that the Markov chain, a very quick Markov chain without using much expertise— with using zero expertise about what basketball even means, there's no meaning to what the Markov chain is doing, it has no interpretation whatsoever about what exactly it's finding the stationary distribution of, is still able to give a top '25' ranking that compares very favorably with the opinions of '30' experts who do this full time. So Villanova, Villanova, North Carolina, North Carolina. You can, you know, pause the video and see that Markov chain is— The Markov chain in this case is able to learn a top 25 ranking purely on the results of the data, of the results given by the games. So this is the data set. It takes it, and is able to compare favorably with the experts.

Video 12 (10:43): Classification Algorithm I

In the last part of this lecture I want to discuss another application of Markov chains, this time to a classification problem. Actually, this time it's not supervised, fully supervised, learning which we've discussed in the first part of this course, it's something called semi-supervised learning. So in this scenario we imagine that we have data that we want to classify, but we have very few labeled examples. You can look at the example here that I'm showing where each of these black dots corresponds to a covariate vector 'x_i' in R_2, and we want to label each of these using a binary classification where it could be either the red class or the blue class. But we have an extreme case, where we only have one labeled example of the red class, and one labeled example of the blue class, and everything else is completely unlabeled. So, if we wanted to learn a linear classifier like we've been discussing in the first part of this course, we would simply have these two points to learn the model on. And so, we would perhaps learn a classifier that looks like this.
The idea behind semi-supervised learning is that we don't want to throw away the unlabeled data because that somehow gives us information about the structure of the data set. In this toy example, that structure is the rings, and so looking at this example we might say that everything on the inner ring

should be labeled blue, and everything in the outer ring should be labeled as the red class. Semi-supervised learning is the general approach to data modeling where we build a classifier. It's a classification problem or a regression problem where we use all of the information of the covariates in combination with this small set of labeled examples that we have. So, we use all of the data in 'x', and whatever 'y's' are available to us. So there are many ways that we can develop a semi-supervised classifier. I just want to look at the one example that relates to Markov chains. So in this case, the modeling motivation is to start a random walk at a particular point 'x_i'. So, we're going to define a classifier where we take any particular point that we want to label. For example, we want to label 'x_i'. It doesn't have a label. And we start a random walk at that point. So, the random walker at 'x_i' moves between the data points. So it moves from any point to any other data point according to a random walk that we have to define. But we start at the point 'x_i' that we want to label such that the probability of transitioning from any point to a nearby point is high, and the probability of transitioning from any point to a point that's far away is very low. So we start a random walk at the point 'x_i'. We then move from data point to data point according to this random walk, random transition matrix that we construct on the data set. And, we continue to transition until we reach a labeled data point. Once we reach a labeled data point, the transitions terminate, the random walk terminates, and we define the label of the point 'x_i' to be the label of the point that it—that the random walk started at 'x_i' terminated at. So, let's look at one example of how we can construct the random walk matrix. For example, if we start at this point, we want to now have a transition matrix on every other point. So, this is a zoom in of the previous slide. Imagine we have 'n' total points in our data set.

So, we want to construct a transition matrix between this point and all 'n' other data...all 'n' points, including itself. So, there's going to be an 'n' dimensional probability transition distribution strictly starting from this point. So, if there are 'n' points, you can imagine that we have an 'n' by 'n' transition matrix, where we construct the probability of transitioning from the i^{th} data point to the j^{th} data point by taking the Gaussian kernel between those two points. So that takes proximity in to consideration. If 'B' is small, then of course the probability of transitioning from this point to this point might be high, if 'B' is large enough to consider that close. But the probability of transitioning from this point to this point, or this point to this point, or any other point outside of the range defined by 'B', that transition probability will be very, very small, essentially zero. So, we construct the Gaussian kernel between all the data points, it's an 'n' by 'n' kernel. We then normalize each row of that kernel matrix so that we have a probability transition matrix. So, we have the Gaussian kernel between all the points. We normalize the rows of that kernel matrix, so that we can now have a Markov transition matrix. Finally and crucially, if the i^{th} data point is labeled, we then redefine the transition for the i^{th} data point to be a self transition probability one. So, if 'x_i' has a label, no matter what the label is, we look at the i^{th} row of the transition matrix, we set everything to zero, and then we set the self transition probability to be equal to one.

So, all the labeled data will transition to itself for all of infinity. It will—with probability one, always transition back to itself, and so that's how we can now terminate the walk, Once we start at an unlabeled data point, and we walk, you know, around along this data set, we finally hit the red point, and now we simply stay at the red point because we'll transition. We'll have a self transition with probability one. So, that's how we terminate. What we've done in the language of Markov chains is, we've created an absorbing state. So, imagine that we have 'S' states, we'll become a little more abstract

now, if the probability of transitioning from the i^{th} state back to the i^{th} state— So, if we're at state 'i' at time 't' minus one, and the probability of a self transition so that we're at state 'i' at 't' time 't', if that's equal to one, so if the probability of a self transition is equal to one, then we say that the i^{th} state is an absorbing state since we can never leave it. It's like a black hole. So, the question now is, for this modeling problem, and in general, in fact— this is a general result, given an initial state, 's_o' is to equal to 'j', so given that I start in the j^{th} state, given a set of absorbing states with the index 'i_1' through 'i_k'— So, imagine we have 'k' absorbing states— and these are their indices, and also given a Markov transition matrix 'M', what is the probability that the Markov chain terminates at any of these particular absorbing states, given that we've started at state 'j'? So, what we're looking for is a 'k' dimensional probability distribution that says the probability of terminating at any one of these different absorbing states, given that I start in state 'j'.

Can we calculate that? So now, if we can calculate this probability for this general Markov problem, we've solved our semi-supervised classifier problem because this—in the language of our semi-supervised classifier, this probability is the probability of starting at a point 'x_j' that's unlabeled, and terminating at one of the 'k' different labeled points. And then, we just take their corresponding labels, and if we want to know what's the probability of terminating at a point with label one, we sum the probabilities of terminating at any point with a label one. So, the answer to this problem, or at least the first way to approach it, is to simply start a random walk at state 'j' and keep track of the distributions on states as a function of time. So, we start the random walk deterministically at state 'j' at time zero, and so in that case 'w_o', using the same notation previously as a vector of all zeroes, it's a vector of—with 'n' minus one zeroes, and the one in entry 'j'. So 'w_o' is going to be an 'n' dimensional vector if we have 'n' data points for our problem, with a one in the j^{th} dimension, and a zero everywhere else because we know where we're starting. So, in this case it would be 's' dimensional.

Then, if 'M' is the transition matrix, it could be any transition matrix, we already know that the probability of where I'm at at time 't' plus one is equal to the probability of where I'm at at time 't' times 'M'. And so therefore, what we want is we want 'W' infinity, which is equal to 'w_o' times 'M' infinity. And, actually, because this Markov chain is— does not satisfy the conditions that we had previously, meaning that given that we start at any state, we can reach any other state, that's not true because now given we start at a terminal state, at an absorbing state, we cannot ever leave that state and reach any other state. So, the stationary distribution in this case is not going to be the same for all starting points. So, if we wanted to be more clear we could simply say superscript 'j' here says, where the one is, what the starting state is, and then that's going to lead to a final distribution that is dependent on the starting state.

Video 13 (13:57): Classification Algorithm II

So, let's try to calculate now what's the probability distribution on terminating any given absorbing state given that I start at a particular state. Towards this end, let's rewrite the Markov transition matrix in a way that's going to be much more convenient. So, let's group the absorbing states at the bottom and the non-absorbing states at the top. In that case, we can write the Markov transition matrix this way. So,

if we imagine that we have 'K' absorbing states, then the bottom 'K' rows of this transition matrix will have a zero in this portion and the 'k' dimensional identity in this portion. So, if we have 'S' different states and we have 'K' absorbing states, this zero will be a 'K' by 'S' minus 'K' matrix, all zeroes, and then this final 'K' by 'K' matrix will be the identity, that's going to model the fact that we have a self transition with probability one. So, given that I'm in one of these bottom 'K' states, the probability of self transitioning is equal to one, according to this construction. And now, let's break up the top half which is going to be 'S' minus 'K' rows and 'S' columns according to the same way that we broke up the bottom half. So, 'B' is now going to be a 'K' by 'K' dimensional matrix— I'm sorry, it's going to be an 'S' minus 'K' by 'K' dimensional matrix. 'A' is going to be an 'S' minus 'K' by 'S' minus 'K' square matrix.
So, 'A' it says the probability of transitioning from a non-absorbing state to another non-absorbing state, and 'B' has the probability of transitioning from any non-absorbing state to any of the absorbing states. So, if we make a transition from any of these top rows to state index by 'B', then we transition to an absorbing state, and our Markov chain's going to essentially terminate there. Okay! So, we make this simplifying assumption. Now, we again, write the observation that the distribution of where I'm at at time 't' plus one can be broken down to the distribution of where I'm at at time zero times the matrix 'M' to the power 't 'plus one. And so, this is the matrix 'M' multiplied by itself, 't' plus one times. And remember, 'w_o' is a vector of all zeroes except for a one in the index of the state that I'm starting at. So that's tells me where I'm going to end up at, where I'm going to be at time 't' plus one. And now, we want to know what happens as 't' goes to infinity. So, in order to understand what happens, when 't' goes to infinity here, we really need to understand what happens when 't' goes to infinity here.
So, we want to know what's going on with this matrix 'M' to the power 't'. So, let's inductively figure this out by looking at the first two cases. So, 'M' squared is equal to this matrix times itself. And so, we have 'AB' times 'A0' is a squared. 'AB' times 'BI' is 'AB' times plus 'B'. And then, '0I' times 'A0' is zero. And '0I' times 'BI' is just 'I'. So, here we've done the same sort of rules when these are scalars, the same exact logic follows when these are submatrices as well. Okay, so we've calculated for 'M' squared. Let's now look at 'M' cubed. So 'M' cubed is this times itself three times. We already know what 'M' squared is so we'll take this and put it in here. So that's figuring out that the 'M' squared portion, and then we just multiply it by itself again to find 'M' cubed. Okay, so we have 'AB' times 'A' squared '0', that's a cubed 'AB' times this term—so a times this term plus 'B' times this term. We have a 'B' here and then we have 'A' times this term. So, we have 'A' squared 'B' plus 'A' times 'B' plus 'B'. Zero times 'A' squared plus 'I' times zero is zero. And zero times this term plus 'I' times 'I' is 'I'. Okay, so we've already now started to uncover a pattern, which we're going to return to in a second. But first, I want to take a one slide digression about the geometric series.
So what we're going to see is this is an example of a matrix version of the geometric series. So let's look at the scalar version of a geometric series first, and then we can see the relationship to the matrix version. So, let's let 'r' be any number between zero and one. And now, we want to calculate the sum of 'r' to the 'u' where 'u' is zero to 't' minus one. So, we take 'r', we raise it to the power 'u'. We do that for 'u' equals zero to 't' minus one and we sum all of those up. We can show that this sum which is a geometric series is equal to one minus 'r' raised to whatever this top thing is, plus one. So since we have 't' minus one, we raise it to 't'. And then the denominator in all cases is simply one minus 'r'. So no matter what 't' is here, we have one minus 'r' in the denominator, and then in the numerator we raise 'r'

to the power equal to one plus this top number. And so, if we let 't' go to infinity, since 'r' is between zero and one, 'r' to the infinity is equal to zero. And so, we simply have one divided by one minus 'r' is the limit of the geometric series. So, how can we prove this? It's very simple, very straightforward, and very clever. So, if we let 'Ct' equal to this sum of 'r' to the zero, which is one plus 'r' to the one, which is 'r' plus 'r' to the two up to 'r' to the 't' plus one—so we're letting this term be equal to 'C' subscript 't'. We want to know what 'C' subscript 't'; so we want to know what is this value. Call it 'C' subscript 't', and now let's take the left and the right-hand side and multiply it by 'r'. So, 'r' times 'C' subscript 't' is equal to this numerator times 'r', and so we simply shift everything. One times 'r' is 'r'.

So let's just write it underneath 'r', 'r' times 'r' is 'r' squared, so let's write that underneath 'r' squared. But then, down here—up here we have 'r^{t+1}' times 'r', that's 'r^t' and so we have this term here. So, we've simply taken the numerator and multiplied it, both sides, by 'r'. Now let's take this top part and subtract this bottom part. So this minus this, is therefore, equal to this minus this. And so, we have 'Ct' minus r times 'Ct' here, and notice that we get cancellations in this case, so this is the cleverness of this thing. So we have one which doesn't have anything here, but everything in between cancels, so we have one minus 'r' to the 't'. And now we solve for 'Ct', which is what we cared about to begin with and we define that to be the sum. When we solve for 'C_t' we get what we want. When we let 't' go to infinity we get the limiting case. Okay, so returning now to the absorbing state problem. We've seen the pattern from the previous slide, what we can show is that the pattern is as follows, that 'M' to the 't' is equal to the matrix 'A' to the 't' down here, we always have a zero and we always have the identity. And then, up here we have 'B' times the sum of the matrix 'A' to the 'u' where 'u' ranges from zero to 't' minus one. So we have this sum.

If you want to go back, we can see that for three we have, we simply bring out the matrix 'B' and multiply by the sum of 'A' squared plus 'A' plus the identity matrix. And so, in general if we raise it to the power 't', we have this term, if 't' equals three, then we have two terms we have 'A' squared plus 'A' plus identity. So, the matrix 'A' raised to the power zero is the identity matrix. Okay, so now we need to, we have two things that we can show. First we have to prove—we have to show, we won't prove it here, but we can show that as 't' goes to infinity, the matrix 'A' raised to the power 't', infinity is equal to zero. So, the limiting case, the limit of this thing as 't' goes to infinity is equal to the matrix of all zeroes, and then, finally here's the matrix form of the geometric series, as 't' goes to infinity, we have that this sum is equal to the identity matrix minus 'A' inverse. So, notice here we had, because we were working with scalars, 'r' was a scalar, we had one divided by one minus 'r'. The matrix form of that is the identity matrix, one becomes the identity matrix minus 'A', and notationally we can't say one divided by a matrix.

Notationally, we have to take the inverse. So this is the corresponding matrix version of one divided by a scalar is the inverse matrix. Okay, so we can plug in, what we care about is the infinite limit so we plug in zero here, we plug in this term here, and then we find that after an infinite number of steps, starting from a particular state encoded in the vector 'w_o', where I end up at in an infinite number of steps is equal to 'w_o' times this matrix. So zero's here, I have this term here in the upper part, and I have the identity down here. And so, what this, what this vector matrix product is doing, because remember, 'w_o' is a vector of zeroes except for one in the state corresponding to where I start. What this vector is really doing is simply picking out a row of this matrix because I'm never going to start at a terminal, at an

absorbing state, so I'm not going to start here. I'm going to start at one of the 'S' minus 'K' non absorbing states, and so I'm literally just picking out a row of this matrix. And so, I'm picking out the probability of which of the absorbing states I terminate at. And so for example, in returning to the semi supervised classification problem, given that I start at point 'x_j', the probability that I terminate at the i^{th} absorbing state is simply equal to the ji^{th} element of this matrix which I can calculate, because 'A' and 'B' are submatrices of my random walk matrix, that I constructed using the kernel.

So, let's look at an example, here's our data set, the black points are unlabeled, the blue and the red point are the two labels that I have. We constructed a Gaussian kernel with a good kernel width. So this was tailored to this problem. And, we see that the—and we color code each point based on the label that we end at. So, this red, this point here is color coded red, almost perfectly red. We have an average between red and blue except it's almost entirely red because with probability almost one, a random walk started here is going to end up at this red point. So it, we don't know how many steps it's going to take. It might take, you know, '1,000' steps just moving around back and forth down here. But, eventually it's going to hit this point, this labeled point with probability very close to one which is why it's labeled red. Similarly, for any point started in this region with probability essentially one, a random walk started at any of those points is going to terminate at a blue point. So, what we're saying here is that we have tuned the kernel width such that the probability of jumping over this chasm is zero. And so, of course that means it's, this method's very sensitive to that setting.

If we increase the kernel width, then we get something that looks like this. So, I don't know how it shows up on your screen, but this is essentially purple because we have an average between red and blue here. So, we get something that transitions from strongly red to purple here, which is saying that given that I start here with 50 percent chance I end up making it here, and with 50 percent chance I end up jumping this gap and ending at this blue point here. If I start at the inner part, I have a much easier time making a direct line to the blue labeled point so I don't jump out this way with as much likelihood. So this, I show this slide to give you a sense that it is something that's sensitive to parameter setting. For a toy example like this we can look and see what the proper setting is, but in practice you won't necessarily know what the manifold looks like. You won't know what these contours look like. So things like cross validation or kernel parameter learning techniques would be necessary for that case.