Optional: Getting Started with Linear Algebra for Machine Learning

Video Transcripts

Video 1: Introduction to Linear Algebra (7:14)

Welcome. Today we'll be talking about an introduction to linear algebra for data science machine learning. From an outline perspective, we'll be talking about a definition of what is linear algebra. We'll talk about one-variable equations, two-variable equations and systems of equations; and then we'll wrap up. So, what is linear algebra? Linear algebra is a branch of mathematics, and it concerns linear equations such as linear functions and representations in the vector space and through matrices. So, linear algebra is central to most all areas of mathematics, and the reason why that we'll be covering this is because Python is really good at representing the system of equations that are defined by matrices and vectors. So first let's start with a one-variable equation. A linear equation is an equation of a straight line, as we all know, written in one variable. The only power of the variable is 1 in this particular case, so linear equations in one variable typically take the form ax plus b equals 0 where a and b are real numbers and a is not equal to 0. So, these equations are solved by using basic algebraic operations.

For example, 2x plus 5 equal 15. In this case, we would expect the value of x to be equal to 5. So, let's take a look at this and with a little bit of code and see how we would express this in Python. So first we want to set our variable equal to 5. Then we want to express the equation 2x plus 5 and then check to see if it's equal to 15. It's a Boolean operator that we'll check to see. Let's run this code. And here we can see that the value is equal to true. Pretty simple. So now we can move on to two-variable equations. So, as you would imagine, we'll take the one-variable equation and add a second variable. So, in this particular case we're going to use the form of the equation will be ax plus by equal to r where r is a real number. So the solution of linear equations in two variables where ax plus by equals r is a specific point in R2, our two-dimensional space, such that when the x coordinates of the point are multiplied by a and the y coordinates are multiplied by b, those numbers added together equals r so as you would expect. So, for example, if we were to take 2y equal to 2x minus 4, we would reduce this equation to y equal to x minus 2.

So, if we wanted to write this in Python, let's take a look at what an example would look like. So first we want to import panda's library so that we have the ability to create a dataframe. So then, from the dataframe we can start to define our x and y data. So, what I'm going to do is I'm going to initially create a dataframe. I'm going to make the x variable a set of, a set within the range of minus 10 to positive 10 or 11, basically. And then I'm going to define my y variable equal to my x variable minus 2. And then we'll take a look at what the dataframe looks like once I've created my data. So, let's run that. So here you can see our data set. So far, my x values I have minus 10, minus 12. And it goes all the way up through twenty variables where I have 10 and 8, according to the equation that I created. Okay. So now we can go ahead and plot this. So, if we want to go ahead and plot this particular set of points, we can then look for interesting things such as x and y intercepts. So, let's go ahead and write the code for this.

So here I'm going to use PI plot or plt, and I'm going to create a figure first. Then I create my figure sizes, give it a little bit of colors and aesthetics. And then what I can do is go ahead and plot my x and y coordinates. So here I'm plotting my from my dataframe my x and y coordinates as we just saw. Then I'm going to create some labels, add a grid to the back of it so we can look up things relatively easily. Then I'm going to create horizontal and vertical lines, and then I can add some annotations which was what we were talking about earlier. The annotations will define the x and y intercepts. And then, if I run the code, you can see here that I've plotted my line, relatively simple straight line as we talked about before from the equation. And then I can see it on my grid. So, the next thing I want to talk about is a system of equations. So, systems of equations means we have two or more linear equations together, and we solve them simultaneously. So, let's take an example. In this case, we'll have two equations, the first one 2x minus y equal 2, the second one x plus y equal to minus 2. So, if we solve them together, we get y equal to minus 2 and x equal to 0. So, let's see what this would look like graphically. If we write the code in Python, we're going to do something very similar to what we did before. We'll import a few libraries that we're going to require for plotting and for our dataframe. In this particular case, now my dataframe is going to be defined in a similar way. So, we're still going to have 20 points, minus 10 to positive 10 or 20 points. And then I'm going to define my dataframe 1, my y values which are going to be equal two times my x values minus 2. So basically, all I'm doing here is I'm implementing the equations. And then I want to be able to plot those equations.

So, when I plot the equations, I'll get a chance to then look at where would those two, two lines intersect, and that would be my answer. So, let's add a few aesthetics to the plot. Let's take a look at it. Add some annotations so we know exactly what we're looking at, and then we'll be able to go ahead and run this. So, finally, just a few more annotations here just for simplicity. And the last part about this is we're going to show the diagram. Okay. So, let's take a look at what this would look like. So here we can see the two lines that we talked about. So, I've got my x plus y equal minus 2, line here in orange; and then, in blue, I've got my 2x minus y equal to 2, line in blue. And then I could see where they overlap. x is equal to 0; y is equal to minus 2. So that's a graphical way to solve the system of equations. So, let's wrap up.

What did we talk about? We talked about what is linear algebra. We talked about one-variable equations, two-variable equations and systems of equations; and then we looked at how we can, how we can code them in Python and then graphically solve problems related to either the two-variable equations or the one-variable equations or even system of equations. Thank you for your time.

****

Video 2: Matrices and Vectors (6:00)

Welcome. Today we'll be talking about matrices and vectors in Python. From an outline perspective, we'll start with an introduction, we'll talk about vectors and lists in Python, we'll talk about matrices, and then we'll wrap up. So, what are matrices and vectors in Python? So, a matrix is a special case of a two-dimensional array. Each data element is of the same size. So, every matrix is a two-dimensional array, but not vice versa. So if we look at the example here, we can see that we have four columns and three rows, so the shape of this particular matrix is going to be three by four, and if you think about each one of the columns, the columns represent vectors, or they can represent vectors.

So, what's a vector? So, we can think of a vector as a list of numbers. So, the vector is a tuple of one or more scalar values. Why is this important? Because this can help to represent lines and line equations. And so, in a three-dimensional space, for example, in the example that we have, because we have four columns of three rows, each one of the columns could represent x, y, and z coordinates. So, this can be valuable for things like machine learning, as we can use them in the description of algorithms and processes such as target variables when we're training an algorithm. So how do we represent vectors and lists within Python? So a list is a data structure in Python which is mutable, meaning that you can modify it, and it's an ordered sequence, meaning that when we define it, you can reference an offset or reference values by an offset, and then you can access particular numbers at that offset. So very much like a string, in definitions, instead of using a double quote, we use square brackets to define a list, and so a list we can use to actually, we can turn a list into what's called a NumPy array, and a NumPy array allows us to be able to treat that particular list more as a vector.

So, let's see what that looks like in code. So, if we take a look at this, I'm just going to import my Python library. I'm going to define just a simple list of two items, and I'm going to show you what the list would look like. So, remember the square brackets allow me to be able to define a list. Then what I'm going to do is I'm going to pass that list into NumPy array function, and I'm going to convert it to an array, and the one thing we'll notice here, so if we run our code, is that now the NumPy array has a shape. So the length of the list is equal to two, meaning there are two items, but in the case of the NumPy array, this is treated as a vector, and the vector has a shape of, because it's only single-dimensional, it's telling us, the two tells us that there are two items in the vector. So, it's a really powerful concept as that we want to be able to use. So, we'll be using the NumPy .array function quite a bit as we think about vectors and matrices. So now let's talk about matrices. So, Python doesn't have a built-in function for matrices or a way to represent matrices, so sometimes what we'll use are what are called lists of lists, and we'll define that as a matrix. So, in Python, let's take a look at what we would, how we could codify that. So if we take a look at this, I can just create a variable called A, and now A can actually include three different lists, and so here I can just take a look at that, and we can see that it represents, these are my effectively, my three lists, which looks like a matrix. So basically, it's a three-row matrix, four-column matrix, and the class, though, of this is of type list. So, notice it's not of type matrix, because it's slightly different. So now I want to talk a little bit about NumPy, because a NumPy library, this is the workhorse of linear algebra, because it does allow us to be able to represent the structures, like a multi-dimensional array object called the ndarray, and we looked at this a little bit earlier when we were looking at a vector. So, let's take a look at an example in Python of how we could use NumPy to be able to represent a matrix. And so in this particular case, I'm going to define the same style of array definition, only now I'm going to call it an array, and so I want to take a look at what would this look like, and it's going to look the same as my list, but when I print the type, the type is going to be a little bit different, and then I'm going to be able to take a look at this from a shape perspective and see that the shape should be equal to my rows by columns, which is what I would have expected. So, let's take a look at this. So, I can see my object. So basically, I have my lists, which is what I would have expected it to be, and then I can see that the shape of my object is three by four, as I would've expected three rows by four columns. So, in this particular case, it's much easier to be able to define the two-dimensional array using ndarray. So, keep that in mind.

So, from a wrap-up perspective, what did we talk through? We talked through an introduction of matrices and vectors in Python. We talked about vectors and lists and matrices and how to store them by using some of the internal functions within Python, but also introducing ndarray, and ndarray is going to be, as we mentioned earlier, the workhorse for linear algebra, and we'll see a lot more of it in the future. Thanks for your attention.

****

Video 3: Matrix Addition and Subtraction (6:09)

Welcome. Today we're talking about matrix addition and subtraction. From an outline perspective we'll start with a definition of what is matrix addition and subtraction. We'll talk about adding or subtracting scalar values. We'll talk about adding or subtracting two matrices and then we'll wrap up. So, what is matrix addition and subtraction? So, two matrices can be added or subtracted only if, they have the same dimensions or set of dimensions. So that is, they must have the same number of rows and columns. So, addition and subtraction is accomplished by adding corresponding elements. And let's take a look at this just from a example perspective. So, if we have two matrices on the left and the right in the upper row here, let's just take a look at it. And so, notice that they are of the same dimensionality, so they're two rows. And they have three columns each. And result matrix will also have the same dimensionality. So, we're not going to do the entire set of math, but let's just take a look at what this would look like if we were to do the manual addition.

So, the first thing that we're going to do, is on the left- hand side of the matrix, here I'm going to take the upper left corner, the number 1. And then I'm going to add that to the corresponding element in the second matrix here on the right- hand side and its upper left item, which is minus 4. And the result then in the same position would be minus 3. So, 1 minus 4 is equal to minus 3. And you get the idea and you do that likewise for all of the elements of the matrix. And notice that because it has the same dimensionality, the result has the same dimensionality as well. So, what about adding or subtracting a scalar value to a matrix? So, in this particular case, let's just say we have a matrix here is 2 by 2 matrix. And in this case, I'm denoting the upper left with a 11 and a lower right by a 22. And so, it's relatively simple to be able to add or subtract a scalar. So here, if I take a matrix plus a 3, then each corresponding element gets a 3 added to it. So here I have a11 plus 3 and then likewise a22 plus 3, all elements are, have 3 added to them. And likewise, for subtraction. So, it seems relatively simple, but you can imagine that if there's relatively large matrices, you don't want to do these kinds of calculations by hand and you certainly don't want to write it out. So, there are functions in Python that will help us to do that. So, let's take a look at what some of those functions are.

And in this particular case we're going to enter the code for you, so that we can take a look at how we can codify this. So, in this first case I want to create a 2 by 2 matrix. So, I'm going to use numpy to create my 2 by 2 matrix. And here I'm just going to use a random set of data. So, I want to create a 2 by 2 matrix and let's just take a look at it. So here random variables that I'm using, or random values. So, I've got a 2 by 2 matrix, 27, 22, 7 and 33. So that's relatively simple. So, the next thing I want to do, is I want to add 3 to those 2 matrices. So, let's go ahead and add 3. Again, relatively simple. And if we were to add 3 you can see here that I've got 30, so 27 plus 3 is 30. Just as I would expect. And then likewise, if I wanted to subtract 3, I could do the same thing.

And in this particular case, I would end up with the same thing. So, you can see how it's relatively easy to do scalar math functions directly with matrices, especially if they're stored as numpy matrices. So now I can move on to adding or subtracting two matrices. So, in this particular case we've already described mostly how this is done earlier. But just as a reminder, if I'm adding two different matrices, then I want to add the corresponding elements from those matrices to arrive at that end result matrix. So, as we talked about before, in this particular case, if I've got a which is 2 by 2, and b which is also 2 by 2, then I add the upper left elements together and then I, and correspondingly then the result then would be the lower right 2 elements together. And notice that I get the same size matrix as the result. So, this also is relatively easy to think about and do, but for large matrices, not something in particular that you want to do by hand. So, what I want to do is, I want to define a matrix a. So, in this particular case I'm going to also make it a random matrix. But in this particular case I want to make it a 2 by 2 matrix and then I want to add it to another 2 by 2 matrix. And then let's see what the result would look like. So, we calculate the adding of a plus b, we'll add the two matrices together, which is relatively simple. And then take a look at it. And then we also want to do a subtraction, so likewise subtraction would perform the same operation, only instead of addition we would do subtraction. So, let's take a look at what that looks like. So, notice here, the first thing we do is we create our 2 by 2 matrix a. We create our 2 by 2 matrix b. Then we add the matrices together and then we subtract them. And as we would've expected, the values check out for either the addition or the subtraction. So, it's a relatively simple operation. But the thing that you do have to keep in mind is that the matrices have to be of the same dimensionality. To allow you to be able to do addition or subtraction. So, let's wrap up.

So, what did we talk about? We talked about matrix addition and subtraction. We talked about adding scalar values to matrices. And then we talked about adding and subtracting two matrices. So, we covered basically each one of the different operations that you can perform when thinking about addition or subtraction with matrices. Thank you for your time.

****

Video 4: Dot Product and Cross Product (4:12)

Welcome. Today we'll be talking about dot products and cross products. From an outline perspective, we'll start with an introduction to dot product and cross product. Then we'll talk about dot product, which produces a scalar value, and then cross product, which produces a vector value. Then we'll wrap up. So, what is dot product and cross product? So, the dot product, or the scalar product, as I mentioned earlier, can be used to find the length of a vector or an angle between two vectors. So, dot product measures the similarity because it only accumulates interactions in matching dimensions. So, here the diagram actually talks about how can you use the dot product to be able to determine either the length or the magnitude of a particular vector or by using the cosine can you determine the angle between two vectors. And likewise, the cross product is used to find a vector, which is perpendicular to the plane spanned by two vectors. So, cross product measures the interactions between different dimensions, and it has many applications in physics in dealing with the rotations of particular bodies. So, this actually gives you a plane as a result. So, let's take a look at how this would be implemented in Python. So, in order to do the dot product, we'll use a NumPy function called dot.

So, it returns the dot product of two arrays for two-dimensional vectors. So, as we talked about before, it returns the scalar value. So, what would the equation look like? So, if we have two single dimension or one-dimensional arrays, so we would have something like this where we would have a vector one, which is a1 and a2, vector two, which is b1 and b2. And then in this particular case, this looks a lot like just the linear multiplication. So, we would take a1 multiplied by b1, add it to a2 by b2. So, and again, because these kinds of functions are ones that you don't want to implement by hand, we want to take advantage of the functions, like in libraries such as NumPy. So, we're going to use the NumPy module or library, and then we want to define two one-dimensional arrays, and then we can actually go ahead and calculate the dot product. So, let's take a look and see what that would look like. So, in this particular case, if we were to do the math, which you can do on your own, you'll end up with the result of 20. So, basically, it's taking the same equation that we talked about above, but only now in this particular case, it's doing it across a, it's doing it across vectors with three elements each as opposed to two.

So, let's talk a little bit about cross products. So, how would we implement a cross product? So, given two linearly independent vectors, let's say a and b, our cross product is a vector, as we talked about before in the introduction, that's perpendicular to both a and b. So, we're expecting a vector as a result. So, the area is formed by the vectors pointing in different directions. The more orthogonal the better, and the cross product measures the area spanned by the two three-dimensional vectors. So, how do we calculate it in Python? So, let's take a look at what the code would look like for something like this. So, in this case, we're going to use NumPy again, because don't forget with NumPy, it allows us to be able to define vectors very easily. So, we define our vector a. We define our vector b, and now we can perform a cross product. And so, as the cross product, we would expect to receive a vector. That's the result. So, in this particular case, if we have vector a, which is 1 0 0, vector b, 0 1 0, our cross product or the plane that's perpendicular would be 0 0 1.

So, from a wrap-up perspective, what did we discuss? We talked about what dot product and cross product were and how they're used. Then we went through an example of dot product, which produces a scalar, and cross product, which produces a vector. Thank you for your attention.

****

Video 5: Matrix Multiplication and Division (7:27)

Welcome, today we'll be talking about matrix publication and division. From an outline perspective, we'll define what are matrix multiplication and division. We'll go through an example of matrix multiplication in Python. And then we'll go through an example of matrix division in Python. And then we'll wrap up. So, what is matrix multiplication and division? In mathematics, matrix multiplication is a binary operation, meaning that it's between two different matrices. So, what does that look like? If we were to look at this example for matrix multiplication, the rule is that the number of columns in the first matrix has to be equal to the number of rows in the second matrix in order to perform the matrix multiplication. So, in our example we have a 2 by 4 matrix on the left, we have a 4 by 3 matrix on the right. So, the number of columns in the first matrix which is four is equal to the number of rows in the second matrix which is four as well. And then the resulting matrix is a 2 by 3 matrix.

So, if we were to look at the operation itself in terms of performing the actual calculation. In order to determine the calculations for the resulting matrix, let's start with C, the C matrix on the far right-hand side, so in this case C11. That particular item is going to be equal to A11 times B11 plus A12 times B21, plus A13 times B31, plus A14 times B41 and that'll get us the first element. And then we in sequence go through the remaining first row and then the second row. And you can imagine how the calculations would take place from there and this gives an example of how to do it. Now technically speaking, there's no such thing as matrix division, at least it's not a defined function. However, we can approximate matrix division by taking the inverse of the matrix that we want to be able to divide by and multiplying it. So, we can actually perform that in matrix multiplication. So, this is what it looks like. So, if we basically say we wanted to divide matrix B by matrix A, it would look like taking matrix B and multiplying it by the inverse of matrix A. So, we'll go through that process and we'll take a look at how we would do that. So, let's take a look at multiplying two matrices in Python. There are two ways to perform matrix multiplication. The first one is to use the dot product function in NumPy, which for 2-D vectors it's equivalent to matrix multiplication. If you're going to be doing N-dimensional array multiplication, then you may want to take a look at some of the specifics of how the function operates. The other way to do it is to use what's called matmul. So matmul is a matrix multiplication, also a NumPy function. And that'll return the matrix product of two arrays, effectively the same thing. Again, these two differ in how they actually do calculations related to N-dimensional arrays. So, if you are working with N-dimensional arrays, you may want to take a look at the specifics of how these two functions work. But let's take an example, so let's multiply two matrices together, so we'll go through an example. So, here in this particular example I want to define a couple different matrices and then I want to go through the process of performing the matrix multiplication. And in our case, we're going to actually try both the dot product style, as well as matmul style. So, the first thing I'm doing is defining my two matrices, I'm defining matrix A which is going to be equal to this value, matrix B I'm going to show them by printing them, and then I'm going to perform the dot product. So, I perform the dot product by using the .dot function. And then I'm also going to do the same thing with matmul, so I want to use matmul. And then let's see if there's a difference between the two and as I said before, there's probably not going to be a difference, so let's just verify. So, I could take a look at my two matrices. I can look at the shape of those matrices just to make sure that the principles that we talked about earlier do actually work and then I can see the results. And the results are exactly the same, which is what I would've expected from either the dot function or from the matmul function.

So, now as I said before if we move on to dividing two matrices. We said before that it's not really a defined function, so the thing that I want to do is I want to be able to take the inverse first and then multiply it by the original matrix. So, in order to do that, we're going to use another NumPy function and this one we're going to use is called inverse or inv. So, let's go through an example here where if we take two matrices, we want to then take the second matrix, so the one that's the divisor and then we want to with it take the inverse first and then multiply it by the matrix that would be on the left-hand side of our division. So, let's get through the process. So, in this case, we've got a matrix B, we've got a matrix A that we're defining. So, we're going to show them, take a look at what these would look like. Then we want to actually take the inverse first of A and then we want to take the dot product of B by A, and then take a look at what the result would look like. So, in essence, what this gives us; first, it gives us the inverse of the A matrix and then we can do the dot product and then we end up with the final result. So, in essence, if we scroll all the way

back to the top, what we're doing here is we're implementing the exact same function that we had before. So, we had an A matrix here, the first thing we did was take the inverse of it and then we did a dot product. So, that's the result in how we would do it within Python. So, as an aside, there is an interesting concept that you can use as well, there's such a thing where you can divide a matrix by another matrix. But what ends up happening here is that it's merely a division of the exact same values. So, let's take a look at what this would look like. So, let me go through the process. So, here again, I'm going to define two different matrices once again. I'm going to take; we'll take a quick look at what the matrices would look like. And then we're going to perform just a standard division of A by B and let's take a look. But the thing that you do have to keep in mind is that it literally is a, it's a place by place or item by item division, so it's a little bit different than the inverse division that we were talking about earlier. So, and this would be the result. So, effectively as I would expect, if I'm performing this kind of a division, what I end up with is all ones as my results.

So, in wrapping up, we talked about matrix multiplication and division, what they are and how they're defined. And then we went through the process in Python of multiplying two matrices and dividing two matrices. Thanks so much for your attention.

****

Video 6: Matrix Transposition (5:08)

Welcome. In this session, we'll be talking about transposition. From an outline perspective, we'll go through an introduction to transposition. We'll go through a couple of examples, and then we'll wrap up. So, first of all, what is transposition? So, the transpose of a matrix is a new matrix whose rows are the columns of the original. So, what does that mean? Let's look at an example. So, in this particular case, the matrix on the left-hand side and we'll get to the superscript T in a second, but the matrix on the left-hand side is a 3 by 3 matrix. And you can see very clearly the rows and the columns. And then, if we were to transpose that matrix, the resulting matrix we would take the take the rows of the left-hand matrix and make them the columns on the right-hand matrix. So, in this case, the 5 4 3 which is the first row now becomes the first column. And in the second row 4 0 4 becomes the second column 4 0 4. And then the third row on the left, 7 10 3, becomes the third column, 7 10 3, on the right-hand side.

So, the superscript T means transpose. Why would we want to transpose a matrix? So, the transpose of a matrix is very important for determining variances and covariances in linear regression, which we'll get to in future sessions. So, let's take another look. The matrices don't always need to be square matrices. So, what you can have if you end up with matrices that are not regular-shaped or square. In this particular example, we can look at them. In this case, we have four rows by two columns. And so, if we were to take the transposition of that particular matrix, we would then end up with two rows by four columns, which is perfectly fine. So how would we do this in Python? Let's take a look at how we can go through the process of doing transposition because it's not something we want to do manually, for sure, or even to write our own functions. So, the transposition method, which is a NumPy function is something we can use to actually calculate the transposition of a matrix. Or, if you do have a matrix already defined, in NumPy, you can use the dot T, dot T shortcut which will also perform the transposition. And we'll go through an example of it right now. So, if I go through the process here, let's just create some code which we'll show you how to do that.

So, as I said before, these are NumPy functions; so, we want to make sure that we're using the NumPy functions for transpose. So, the first thing I'm going to do is I'm going to create two different matrices that I can use for this process. I'm going to show you what those matrices look like, and then I'm going to do- perform both of those functions. First, I'm going to exercise the shorthand, which is to take a matrix and take the transpose by doing a dot T nomenclature. And then, secondly, I'm going to call the transpose function, which is transpose with the a matrix. So, let's take a look at what this would look like. So, in this first case, this is my original matrix. And then I'm looking at the transposition in both cases, in the first case, the shorthand. So, if I look at the original matrix, the first row is 0 1 0 or sorry, 0 1 2. And then, in the transposed matrix, actually in both of them, you can see that the first column now is 0 1 2. So, it effectively did the transposition. So those functions are really good to keep in mind, and the shorthand is actually even easier.

So, one important property of transposition of a matrix is that it's the product, when you take the product of two matrices, you can actually look at it in one of two different ways. So, if you take the dot product of two matrices a and b and take the transpose, it's actually equal to the transposition of matrix a multiplied or dot product with the transposition of matrix b. And, if you want more information, you can take a look at this Wikipedia article. But what that would that look like in Python? So, let's take a quick look. So, let's perform the operation. So first we want to calculate the dot product of the transpose of b times the dot product of a, and then we can take a look to see if it's identical to the dot product of a with the, sorry. the a and b dot product and take the transpose of the result. So, let's see if those two are identical. And, if we take a look, sure enough, the two are identical. So, this equation holds true. And, as I said before, if you want more information, you can look in more detail at the Wikipedia article.

So, in wrapping up, we went through what transposition is, we went through a quick introduction to it and then we looked at some different ways that we can perform transposition in Python. Thanks so much for your attention.

****

Video 7: Matrix Determinant and Inverse (6:28)

Welcome. Today we'll be talking about matrix determinants and inverse. From an outline perspective, we'll go through a definition. Then we'll go through some examples in Python. So, from an introduction perspective, the determinant's useful in determining linear equations and capturing how linear transformations change area or volume with changing variables and integrals. So, the determinant can be viewed as a function whose input is a square matrix and whose output value is a number. And likewise, determining the inverse, it assumes that we have a square matrix A which is non-singular, meaning that the determinant isn't zero, and then there exists some matrix which, when multiplied by the original matrix, yields what's called the identity matrix, or values of all one. So here the function. If we take, say, matrix A, multiply it by its inverse, it's also equal to A inverse times its matrix, so you can flip the two in terms of the dot product, and it's equal to the identity matrix. So, we'll go through some examples and show you how that works. So, let's start with the matrix determinant. So, it's very useful value in linear algebra because it does compute the diagonal elements of a square matrix. So, for our two by two matrix, which is square, it's simply the subtraction of the product the diagonal elements.

So, let's take an example. So, if we have a matrix AB and CD, the determinant's computed as AD minus BC. So, it's relatively simple for a two-by-two matrix. So, for much larger matrices, though, it's not something you want to calculate by hand. So luckily NumPy has a determinant function that we can use to be able to calculate the determinant for us. So, let's go through an example and take a look at what the determinant would look like. So of course, we want to include NumPy. We want to define our, first of all define our first matrix, and then we want to simply take the determinant and then actually go through the process by hand just to show you what it would look like and see if they're the same. So, let's run our code and take a look. So, in this particular case, we can see our matrix, and then we can go through the process and calculate, perform the calculation, across the diagonal, as we had indicated above. So here you can see that calling the determinant function yields the same value as if we were to calculate it by hand. So that's great. So, let's move on. So, let's talk about the matrix inverse. So once again, we have a NumPy function called inverse which will allow us to calculate the inverse of a matrix, and remember, we talked about the inverse of a matrix. For any matrix A the inverse, when multiplied by the original matrix, yields the identity matrix. So, let's see what that would look like. So, let's perform the operation on a matrix. So here we can define a matrix x, and then we can take the inverse of that matrix and then take a look at what that matrix would look like. And then, just to prove the point, we can actually take the dot product of x by y. In this particular case, ensure that it's the identity matrix. So, let's take a look at that. We can see here our original matrix, we can see the inverse, and then, if we multiplied by the, if we take the dot product of x and y, we can see it's the identity matrix, so the ones on the diagonal, which is what we would expect. So now let's look at something that's a little bit more interesting. So, we're going to look at an example that uses something called solve.

So, what solve does is it solves a linear matrix equation or system of linear scalar equations, and it computes the exact solution of x, which is well-determined linear matrix equation, which is ax equal to b. So, let's take a look at this example. So, let's run through this process. So, in this particular case, once again using NumPy, so all these functions are coming from NumPy. We define an A matrix. The first thing we want to do is take a look at the A matrix. Then we'll take the inverse of A, and then we'll take a look at what that would look like and print it, and we'll take- we'll look at the B matrix. We'll define it to be another matrix, and then we'll compute the value of a minus 1 times b, and then we'll solve that equation. So basically, what this will end up being is a solution to the linear equation x equal 5, y equal 3, and z equal to minus 2, and then we want to check does ax equal to b, which is what we would expect, and then we want to compare that to the dot product of a by x and then be able to take the, sorry. take the dot product of ax and then compare that to b. So let's see if that function all equates. So, here we end up with our arrays. We have our inverse. We have our B matrix, which that we've defined. Then we can look at the computation. So, then we check to see is ax equal to b, and then the value is equal to true. So, that gives us the ability to be able to validate some of the functions that we are able to calculate. So, what that's allowing us to be able to do is it lets us solve the equation for the particular value by using the inverse. So, finally, let's take a look at can we obtain the same result by using matrix multiplication, and the final piece of code that we'll take a look at is to be able to try the same kind of thing by using the inverse, multiply it by the B matrix, and then see if we end up with the same value. And sure enough, we end up with the exact same value.

So, what did we talk about? So, in this case, we talked about matrix determinants and inverse. We went through some calculations which allowed us to determine the matrix determinant. We used the matrix inverse, and then we performed some interesting functions to be able to use some of these functions in linear algebra equations. Thank you for your time.

****

Video 8: Span and Linear Independence (9:30)

Welcome. Today we'll be talking about span and linear independence. From an outline perspective we'll start with an introduction, we'll talk about span, and go through the example. Then we'll talk about linear independence, go through an example of that, and then we'll wrap up. So, from an introduction perspective, what is span and linear independence? So span is a set of vectors that is a set that represents all linear combinations of vectors. And it'll look like a plane in a three-dimensional graph. And we'll see that in a minute. So, given a set of vectors, linearly independent vectors are written as columns of a matrix and are solved by the equation ax equals zero. If there are any non-zero solutions, then the vectors are called linearly dependent. If the only solution is x equal to zero, then they're called linearly independent which is what we'll be looking for. And we'll go through that example as well. So, let's take a look at span. So, the span is a two-dimensional plane passing through points in the coordinate space. And so, for a more detailed explanation you can check out this link that I've provided. I'm not going to go through all the math with you. It's a little bit too complicated for this discussion, but let's look at an example of how we could write this in Python. So, we're going to look at an example where a is equal to the set of a1, a2 in or two vectors actually a1, and a2 in a three-dimensional space or referred to as r superscript 3. So, let's take a look at the code and what this would look like. So, this is a little bit involved but I'll explain it as best I can as we go through this. So, because we're graphing it, I just wanted to make sure that we include all the libraries that we're going to need. And one thing that I'll point out as we start to include the libraries is that we're using the three-dimensional library called map plot 3D and that'll give us our three-dimensional axis'. So, I'm going to want to create our figure which is going to be what we're going to be drawing on. Then we can start to define some of the ranges that we're going to require. So, we have x and y minimums where we're going to plot within the minus 5 to 5 range. Then we can start setting sets of values.

Notice here also we're defining a z linear space which we want to make sure that we capture. And in addition to our x and y space. Next we're going to go ahead and start plotting so we create all the plot lines and we want to make sure that the, we're using very, very particular lines which are going to be x, y, and z plots which we're going to use as dashes which we want to make sure that we're drawing the vectors. Then we're going to define a linear function to generate the plane. So, the linear function is going to be defined by the equation ax plus by. And from there then we can start to create our vector coordinates. So, we're going to use NumPy arrays to be able to create our x and y coordinates. Then our z coordinates are going to be a combination of the xy coordinates based off of our function that we just defined above which is the linear function to generate the plane. And finally we start to move into, we need to draw the lines from the origin to the vectors so then we can start to draw and plot the different coordinates that are required on the plot in order to be able to show the vectors but also the plane that we want to be able to draw.

So here, once we get to the bottom, the last thing we want to do is we want to call the plot surface function which will allow us to be able to plot the surface area which is going to give us our plane. There's a number of different parameters that you can look at in a little bit more detail, but I wanted to show you the span concept, and it looks like I got an extra paren in here. So, if I run this code, you can see that I end up with my three-dimensional space, three-dimensional diagram. I have my two vectors, my vector a1 and vector a2, and you'll notice that I have a plane that's passing through those vectors. So basically, it is the coordinate space that includes those two vectors. So that, in essence, is the span that includes the two vectors that I've defined. So now let's move on to linear independence. And in order to describe linear independence, let's just start with dependence quickly because it's sort of the corollary to linear independence. But dependence in systems of linear equations means that two equations refer to the same line. So, there is basically an infinite number of solutions that will satisfy the condition of the equations. So, that's not what we're looking for because we're looking for independence. So, in the case of independence we want to solve the system of linear equations where they only meet at one point. So that single point is really what we're looking for. And if that's true, if there is only one point, then we can say that the two lines are linearly independent. And then here's a little bit more detail in terms of the math that you would use in order to be able to solve the equation. But basically, we write the system of equations as ax equal to b.

So, let's go through the process of determining linear independence. So here I'll write some code which will allow you to be able to see how we can do this by using some different algorithms. So, in this particular case we're going to use SciPy. But we want to define two equations. The first equation is going to be equal to y equal to 2x plus 1. And the second one equal y equal to 6x minus 2. And so, the result I want to see is there a single point that solves both equations or are they dependent? So, we can go through the process of creating the vectors that would represent those equations. So, we can do that using NumPy array. So, the first equation is going to be equal to a. The second equation is going to be equal to b. And then we can go ahead and solve for those equations. And we'll print this out as we go. But basically, the solve function allows us to be able to determine is there a solution to both of those equations and what is the value? And then finally, the all close will tell us whether or not the two equations are independent. So, let's go through this process. And so here you can see in the beginning where we showed the matrices first basically, or the vectors. So, a and b. And then we solved for a single point where that was true. So, it's basically go to .75 and 2.5. And then the last part was whether or not the equations were independent. And the all close function tells us that they are indeed independent. But if we don't believe that let's say, well, we should believe it, but let's just go through the process of how can we actually graph this to see what would it look like? So, let's go through this process. So, we'll use the same lines but now we're going to graph the equations. So, here the conjecture is that the two lines only meet at one point. So, actually I want to go through the process of proving to myself that that's true. And you can too. And so, with the code here we're going to basically go through the same process only now we're going to graph these two equations. And then we're going to see do those two equations meet at a single point? So, most of this is really just set up for the plotting of the function, or the two lines. And so, we go through the process of configuring our environment and then we can go ahead and plot those two lines. So, let's just go ahead and do that. And then here you can see if we plot the two lines, we can see that there is a particular, there is only one line. Or sorry, one point where they intersect, which is indicated here. And the intercepts are listed here as well.

The intercepts were actually listed above. So, you get the idea that you can either use the functions that we've defined here to be able to determine whether or not two equations are linearly independent by representing them as vectors. And then using the solve function and the all close function to be able to determine are they linearly independent. Or you can go through the graphing function and then you can actually visually determine whether or not that's true as well.

So, let's wrap. What did we talk about? So, we talked about span and linearly independent functions. We talked about span; we gave an example of span. We talked about linear independence. So, there is more information in the links that I provided. So, if you're interested in a little bit more of the detail, make sure you check out those links and thank you for your time.

****

Video 9: Eigenvalues and Eigenvectors (8:50)

Welcome, today we'll be talking about Eigenvalues and Eigenvectors. From an outline perspective, we'll start with an introduction. Then we'll talk about Eigendecomposition. We'll confirm an eigenvector and eigenvalue calculation. We'll reconstruct a matrix from those values. And then we'll wrap up. So, first, let's talk about what an eigenvalue and eigenvectors are. So, eigenvectors are unit vectors, which means their length or magnitude is equal to 1. And they're often referred to as right vectors, which simply means that they're column vectors and that if we wanted to say take the corollary of that, we would say that left vectors are row vectors. So, in this particular case, we'll be talking about column vectors. So, eigenvalues are coefficients that are applied to the eigenvectors that give the vectors their length or magnitude, for example negative values, reverse direction of an eigenvector as part of scaling it. And we'll talk about what that means actually when we get into some graphing. A matrix has only positive eigenvalues, that type of matrix is referred to as a positive definite matrix. And then if it's all negative, then it's referred to as negative definite matrix.

So, let's take a look at this equation. So, if I have a matrix A and then I multiply it or dot product it with the eigenvector, then that is equal to the eigenvalues which are dot product or multiplied by the same eigenvector. So, that must be true. So, let's take a look at the eigendecomposition, so what is the calculation of eigendecomposition? So, this example I want to take a 3 by 3 matrix and eigendecomposition, so I want to decompose the matrix into the eigenvalues and the eigenvectors that are associated with it. So, let's go through that process. So, let's write some Python code which will allow us to be able to do that. So, the first thing I want to do is import NumPy and there's a couple of things that I want to be able to use. And then from the NumPy library, I'm going to include eig which is my eigenvalue or eigendecomposition library, which will long me to be able to perform the eigendecomposition. So, the first thing I want to do is create very much like our equation, I want to create my matrix A and then we'll take a look at that. And then I can perform the eigendecomposition by calling the eig function. And then I can take a look at what the eigenvalues are and the eigenvectors. So, let's run this code. And we can see here, we started with the square matrix, we were able to see the eigenvalues that were provided, and then the eigenvector that's associated with it. So, if we go back to the equation, so basically what we did was we took this matrix here, and then decomposed it into the eigenvalues and the eigenvector.

So, let's take a look at how we can confirm that. So, how can we confirm that those are the right eigenvectors and eigenvalues? So, first, we want to define the matrix, calculate the eigenvalue and eigenvectors which we did, and then we'll test whether the first vector and value are in fact the eigenvalue and eigenvector of the matrix. And we know that they are, but it's a good exercise anyway, so we can go through that process. So, let's do that as our next example. So, let's just go through the confirmation of the eigenvector. So, here again, import similar functions, we define our matrix once again. And let me just scroll so you can see this a little bit better. We'll calculate the eigendecomposition. And so, now here we can see the various vectors that are returned and now we can use those to take our dot products. So, then we can start to we can implement the equation that we looked at above. And what we'd expect to see is that the equation holds true on either side. So, we want to see if the values actually do match. So, we can do the dot products and do the matrix multiplication ourselves. So, let's go through that process and then we can validate that the multiplication of the eigenvector by the second eigenvalue matches basically the left-hand side of the equation. So, let's run this. So, and I'll go kind of slowly through this. So, the first thing we want to do is confirm the first eigenvector, so this is the eigenvector value. The eigenvector multiplied by the first eigenvalue is equal to these values. And then the two vectors match as we would have expected okay. And then the second part is confirming the second eigenvector, so this is the second eigenvector that was returned. And then we can multiply the eigenvector multiplied by the second eigenvalue and then we want to see if that matches, and sure enough it's the same value. So, on one hand, we're actually performing the multiplication by the vector. In the second case, we're using the eig function and we're just validating that the match, so that all works out fine.

So, the next thing we want to do is we should have the ability to based on the equation reconstruct our original matrix. So, we can reverse the process and reconstruct our original matrix given only eigenvectors and eigenvalues. So, if we start with that, the first thing that we need to do is look at converting the matrix, the eigenvectors sorry need to be converted into a matrix where each vector becomes a row. And then the eigenvalues need to be arranged into a diagonal matrix and we can use the NumPy diag function to do that. So, let's take a look at what the code would look like which will allow us to be able to do that. So, let's go through the reconstruction process. So, again, using NumPy, so NumPy actually is very important for linear algebra, we'll use it quite a bit. So, we're including a series of functions that we're going to be able to use. So, the first thing we want to do here again define our matrix, our A matrix; we'll take a look at that A matrix. Here, we'll calculate the eigenvectors and eigenvalues to give us a starting point. And now we want to be able to create the matrix from the eigenvectors. So, we're going to set Q equal to the vectors which was returned from the eigendecomposition. Then we want to assign R equal to the inverse of Q. And then we can take the diagonal of the matrix from the eigenvalues, so we basically would take the eigenvalues which is up here above. And then create a diagonal value which we'll define as L. And then finally at the end what we should be able to do is reconstruct the original matrix. So, we can say that B, which will be the original matrix, is equal to Q dot product L and then we do product that with R. So, then that would produce the final matrix. So, let's take a look and see if that all works. Okay, so here we looked at the original matrix which, was 123, basically a 3 by 3 matrix, I'm not going to read it to you. And then we wanted to see if at the end, we ended up with the exact same matrix and sure enough it does match. So, we reconstructed our matrix after we had decomposed it into the eigenvectors and eigenvalues.

So, what did we go through today? So, we went through an introduction to eigenvalues and eigenvectors, how we decompose those matrix into its eigenvalues and eigenvectors. We confirmed those values. And then we reconstructed an original matrix from that. I've included here and a lot of the code examples came from Jason Brownlee produced a real nice set of code and there's a good explanation of eigenvectors and eigenvalues, so you may want to take a look at that if you want some more information. Thank you so much for your attention.

****

Video 10: Singular Value Decomposition (7:33)

Welcome. Today we'll be taking about singular value decomposition. From an online perspective, we'll start with a definition. Then we'll talk about how to calculate single value decomposition. We'll look at using SVD for pseudoinverse, and then using SVD for dimensionality reduction. Then we'll wrap up. So, from an introduction perspective, what is singular value decomposition? So, the SVD of a matrix, of a dimension n by d, is given by the following equation. So, we can say that X is equal to U by D by V transpose. Where U and V are square orthogonal matrices, which means that the matrix times its transpose is equal to the identity matrix. And then D is a diagonal of dimension d by n. And if you wanted a little bit more information, there's a link at the bottom, and it will give you a little bit more detail on what singular value decomposition actually is. But let's look at how we calculate single value decomposition. So, in Python, it's actually pretty easy. So, for us, what we're going to do is, literally just call the function, which is part of scipy library. So, the first thing we want to do for an example where we're going to use a 3 by 2 matrix. We can calculate the singular value decomposition by just calling the function SVD. So, let's go through the process. First, we define our A matrix. And then notice here that the constituent parts of the equation that we talked about earlier are returned from the SVD function. So here my U matrix; the D, diagonal; and the transpose of the V matrix are being returned as part of the invocation of the SVD function.

So, let's take a look at what this looks like. And then here you can see, these are all of the values that are required in order to be able to do my singular value decomposition. So that's great. So that was pretty easy. Next, we're going to talk about how we can use singular value decomposition for pseudoinverse. The reason it's called pseudoinverse, is because it's a generalization of matrix inverse. Matrix inverse is only applicable to square matrices. But what we have a situation where the number of rows and columns are not equal, or not square, we also may want to be able to calculate the inverse of that matrix. So numPy provides a function called pinv, which allows us to be able to calculate the pseudoinverse. So, I'm going to give you an example here of a 4 by 2 matrix, where we can actually run through that function. And then see what the result would be. Which should be our inverse, our pseudoinverse, by using a singular value decomposition method. So, the first thing we want to do is define our array function again.

So, we define our array. Then we're going to just take a look at what that would look like. The A matrix. And then calculate the pseudoinverse. And then finally we can just take a look at that by printing it. So, a little typo there. So, let's look at what this would look like. So here, notice that it's a 4 by 2 matrix. And then the pseudoinverse of that is going to be a 2 by 4 matrix, which is what we would have expected. And then the values are going to be the inverse values according to the inverse calculations. And you can look that up if you

wanted to look into more detail of what an actual inverse is of a matrix. So lastly, I want to talk about singular value decomposition for dimensionality reduction. And this is a popular use of singular value decomposition. Because typically, data with large numbers of features, such as more features or columns than observations, which are typically rows, you want to reduce it into a smaller set of features that are relevant to the prediction problem. So especially with very, very large matrices, you want to reduce it down to a much lower rank that can approximate the original, and then you can use that to be able to be able to use that for your machine learning algorithms, or for your calculations. So, let's take a look at what this would be in Python. So, what we could do here is that we can take advantage of some of the functions that we've already discussed, but primarily we're going to use singular value decomposition, which allows us to be able to perform the function.

So first we want to define our arrays, as we've been doing before. So first we're going to create a matrix that's relatively not too large, but relatively large to allow us to be able to show what it would look like to reduce it down to a smaller set. So, the first thing we want to do is, once again, invoke our SVD singular value decomposition function. Get our core constituent matrices. Then what we want to do is we want to create an n by n diagonal matrix. So that matrix would allow us to be able to define the diagonal, specifically. Then we want to populate it with n by n diagonal matrix. We'll select the number of elements that we want to be able to reduce it to. In this particular case, I just want to reduce it down to a 2 by 2 matrix. Then I can go through the process of reconstructing my original matrix. And then take a look at what the final matrix would look like in the reduced form. So, let me type again.

Let me go through this process. So, let's take a look at what we've done. So, in essence, we had our original matrix. We reconstructed the matrix, which is no great shakes in that particular case. But then we want to be able to say take a look at the T matrix. So, this is our reduced dimensionality, so this is the U dot product with the diagonal. So now, I've taken my relatively large matrix, and notice here the unbalanced nature of it, so in this particular case, I have many more columns that I do rows, and I've reduced it down to a 3 by 2 matrix. And then the other way that I did this is just a different way to calculate the same reduced dimensionality is I'm taking a dot product with the transpose of T, which is another way to arrive at the same, same result. But the reduced dimensionality is really the key component of what we're doing here. So, you can imagine that this would be really important for large matrices where you do have that level of imbalance in terms of features. So that'll come in handy certainly in a lot of the problem domains that we will be working with.

So, in wrapping up, we talked through singular value decomposition. We calculated SVD. We looked at how to use SVD to create a pseudoinverse. And then we used SVD for dimensionality reduction. And there's a great paper here or a great article that you might want to talk a look at, where a lot of this code was derived from. But this, by Jason Brownlee, who went through the process of defining how you can actually use these functions for singular value decomposition, so I encourage you to take a look. Thanks so much for your attention.

****

Video 11: Principal Component Analysis (8:02)

Welcome. Today we'll be talking about Principal Component Analysis. From an outline perspective we'll start with an introduction. And we'll talk about how you can manually calculate PCA using Python functions. And then we'll talk about how to use sklearn, one of the classes in sklearn to be able to define PCA process. And finally, we'll wrap up. So, first of all let's start with an introduction to PCA. So, it's a mathematical procedure that transforms a number of correlated variables into a smaller number of uncorrelated variables called principal components. So that's important because it produces a number of variables in our data by extracting the important ones from a larger pool and it reduces our dimensionality with the aim of retaining as much information as possible. So those principal components are really important. So, each component will examine the magnitude and direction of the coefficients for the original variables. So that's how the technique works. The larger the absolute value of the coefficient, the more important the corresponding variable is calculating the component. And then how large the absolute value of the coefficient has to be in order to determine whether or not it's important is really subjective. So, you kind of have to know your data. And that's really the importance of a data scientist in domain knowledge. It allows you to be able to interpret the data to determine were the particular identified components really the important ones. Next, we'll talk about how to manually calculate PCA. Because there is no PCA function, for example in NumPy, we can easily calculate it by using a step-by-step function. At the end of this file there's a link, which will give you the details on how you can actually implement it using a step-by-step function. But my code will go through it in more detail.

So, in our example we want to take a three-by-two matrix. We want to center the matrix, calculate the covariance matrix of the center data and then use eigendecomposition of the covariance matrix. And those details are identified in the article at the end of this Jupiter file. The PCA is an operation that we can apply to a data set, and it's represented by an n by m matrix. A, the results in a projection, which we'll call B. And that will be, actually, a reduced set of data. So that reduction allows us to be able to identify the principal components themselves. So, let's take a look at what the code would look like if we were to do this manually. So, let's go through the process. So here, as I mentioned, even though there is no PCA function in NumPy, a lot of the functions that we're going to use in the manual process do come from NumPy. The one thing that is important, we are going to use the eigendecomposition function as well, in order to get our eigenvectors and eigenvalues. So, the first thing we're going to do is create our matrix, which is our three-by-two matrix. The first step is to calculate the mean value of each column where we assign it to M. Then we want to calculate the centers of the columns by subtracting the mean values. And we'll call that, that matrix, the result of that equal to C. So that's basically A minus M that will allow us to be able to center the columns. The third step is to calculate the covariance matrix of the centered matrix. And so, we can use the covariance function, which will allow us to be able to do that. Step four we'll perform the eigendecomposition of the covariance matrix by calling the EIG function from NumPy. Then we'll take a look at what the eigenvectors and eigenvalues are. And then finally we can go through the projection process. So, now we can project the data into the subspace, which is basically a reduction, and then we could take a look at that as the reduced set of values, which should be our principal components. So, if we go through the process and you can take a look through this, but you can see our original matrix.

We centered our columns. We went through the principal component analysis for the eigenvalues. And then finally we went through the PCA reduction. So, keep these in mind. The eigenvectors for the PCA components here, the eigenvalues for the PCA variance and then the reduction itself. Those will all be important in a minute when we talk about how we could use PCA using sklearn. So, the next step we'll look at is how do we calculate principal component analysis dataset using a class, which is available from sklearn. So, the sklearn learn library gives us the benefit of a to be able to define a class that we can use over and over again, which is in essence giving us the ability to do projections on different size arrays, once we have actually performed the fit or matrices. So, when we create a class there's a number of components that we can specify as parameters, which we'll go through in a minute. And once we fit it the eigenvalues and the principal components can be assessed on the PCA class via the explained variance and the components, attributes that we'll see in a minute. So, I want to take the same example, at least same source data that we used before and then compare the results of the manual process to the process that I'm about to show you using the PCA class method. So, let's take look at that code as well. So, if we go through the process here we're going to use NumPy again to allow us to define the array, but now we're going to use sklearn and we're going to import the PCA class. So, we define our array, three-by-two array, take a look at the original matrix as we've done in the past, and now what we can do is we can create a PCA instance. And then what we can do is fit the data. So, we basically are going to pass into the PCA class instance that we created and we're going to pass into the fit function or fit method, the matrix that we want to be able to fit. Then what we can do is we can print or take a look at the values and the vectors. So, here we take a look at the components and then we can look at the variance, which we talked about earlier, and then finally I can actually go through the transformation process to see the end result.

And so, let's take a look at what this looks like by using the class. And so, if we take a look at this, and I'll go back in a second to the other data that we looked at, so we can see the PCA components and the PCA explained variance, and then the final transform. And so, that matches with the results that we saw before. So here, the PCA components are in essence the eigenvectors, which are the same. The eigenvalues is the PCA variance. And then the reduction, with a little bit of rounding is actually the same. And so, the manual process matches the using sklearn with the PCA class. And so, you can actually perform it either way, but I just want to give you a feel for how you can calculate PCA on room.

So, in wrapping up, what did we discuss? We talked about introduction to PCA, what it was. We manually calculated PCA by using a number of functions in NumPy in Python. And then we calculated PCA using sklearn in the PCA class. And as I referred to earlier, there is a great article by Jason Brownlee who talks about the manual process. So, if you're interested in the manual process and some of the references to why it works, you may want to check that out. Thanks so much for your attention.

****

Video 12: Maximum Likelihood Estimation (9:20)

Welcome. In this section, we'll be talking about maximum likelihood estimation, and we'll be doing it by example. From an outline perspective, we'll start with an introduction. We'll set up some data that we can use in our examples. We'll talk about modelling ordinary least squares with Statsmodels. We'll calculate ordinary least squares using MLE methods, and

then we'll wrap up. So, what is maximum likelihood estimation? It's a technique for estimating parameters of a given distribution by using observed data. So, the example, we want to have some data where we know it fits a normal distribution, but the mean and variance are unknown. So, MLE can be used to estimate them using a limited sample of the population by finding particular values of the mean and variance so that observation is the most likely result that has occurred. So, let's take a look at this from a graphical perspective. So, because it's probabilistic, if we know the distribution or the function or the population, and we know say an x value, we can determine the y or the likelihood that it would have occurred. So, we're going to go through this in a little bit more detail, but first let's set up some data that we can use in order to be able to go through our examples. So, let's just start with that. The first thing I want to do is import just a number of different libraries that we're going to be able to use, and I do want to start this off by when we generate the data I'll use seaborn to be able to plot the data so that you have the ability to just take a look at what the data would look like, because some of it's going to be random, but as we determine some of the regression lines and regression fits, it'll be important to be able to have a visual perspective on what the data may look like.

So, let's start here. So, we're generating some data. We want to identify 100 points that we're going to be able to use, and we'll set our x values. And we will create a little bit of randomness that we want to be able to put into the model so that it's not actually just sort of a straight line. So, rv is actually using a random function to be able to do that. Then I want to calculate my y values. And what's really important here is the fact that like the calculation of my y value is equal to 5x plus my random variable. Then I'll create a data frame, which will be associated with that, and I'll add a constant value. And we'll talk about that in a minute. But let's just take a look at the data and by plotting it. So, this is my dataset, and so here you can see sort of a pseudo random regression line that was identified here, not an actual regression line, but it's pretty much the fit that's expected. And then a couple things that you want to be able to notice here, the first one is that this particular x coefficient is really important because remember the number 5, because we'll be talking about it later. So, let move on. So, if we wanted to model ordinary least squares with Statsmodels, because this is sort of regression-like, continuous data, we can use the OLS, ordinary least squares, to calculate the best coefficient and the Log-likelihood as a benchmark.

So, let's go through that process. So, we can make a call to the OLS function. And so here, I want to first split my features and target data. So, the first thing I want to do is split my x and y and then perform a fit of my data. So, I'm going to split this to just be a constant and x for my y, or sorry, for my x dataset, and then I'm going to pass in my y values. So, let's go ahead and do that. And take a look. And a couple things to notice, and it's really, it's a nice set of data that's returned for the regression results, but a couple things that we want to notice. One, is that the x value, as we just talked about before, has been at least from the dataset, is determined to be close to 5, which is good. And then the constant, which is ultimately going to be my y intercept, is really close to 0. And if we kind of look at this, and this diagram is a little bit off, but if you look at, you look where x is equal to 0, you can see that it's just a little bit above 0 here. So, that's pretty close to the values we're looking for. Okay. So, I just wanted to draw your attention to that, because we're going to need that in a minute. So, let's calculate OLS using MLE methods. So, we just used the Statsmodels to be able to calculate it by calling the OLS function. So, now we're going to use MLE methods.

So, the first thing I want to do is calculate OLS. So, let's go ahead and run this process. So, the first thing I want to do is define what's called a likelihood function. So, here, I want to calculate my own regression function, so I'm going to go through the process of defining inputs that I want to be able to use to be able to calculate what I think would be the regression function for my OLS. So, the other thing that I want to be able to do is compute a probability density function, which would be important as I look at the distribution of the function. So, some of this, as we go through this, it's going to be important because we want to be able to just say pass our parameters in and then go with what a best guess process might be in order to define what the actual distribution function would look like. So, let's go ahead and run that so I can define my function. So, now I've got my function defined, and now that I have a cost function, or defined this regression function, I can initialize it and then minimize that function.

So, the next thing I want to do is I want to take a guess at what my values might be and then pass that into a particular regression function. So, I'm calling my regression function with a specific method, so I'm using a Nelder-Mead method, and I'm not going to go into details on that, but you can look it up if you wanted to get into a little bit more detail, especially as we're looking at using SciPy optimize. So, that particular function would be important, and then I want to define the results. So, let's check the results. So, I determine my function value. I have a number of iterations that I ran through in order to get a best fit and then a number of function evaluations. So, as I go through the process now, I want to be able to check my results. So, here I can just run results because that was the value that was returned. So, if I take a look at my results, I can see here this, the array that's returned. I can see that my optimization was completed successfully, and I could see that I've returned an x array. So, while those results are great, it's a little bit difficult to compare that to the results that we had before. So, what I want to do is I want to pull some of that data out and put it into a data frame which will allow us to be able to compare the data to the results that we had before a little bit more effectively. So, this is what I'm doing here. So, I'm going to basically pull out some of the existing data from my results and then put it into a data frame. And then I can compare it to the data that I had before. So, here you'll notice, so I've run through the process, and now I'm going to compare my constant value and my x value, which were the two important values that I was looking for. So, here I've got basically really close to 0 minus 0.0016. Let's just go up to the values that we had here before and notice that it's 0.0017. So, pretty similar, and 4.9919 for my x value. That's exactly the same. So, we expected that to be very close to 5. So, notice that the calculating the MLE based on these two methods comes out to be about the same.

So, you could use whatever one that you want, and whatever one may be more simplistic for you. So, from a wrap-up perspective, we went through introduction to maximum likelihood estimation. We set up some data that we could use, and then we showed you the data using a plot. We modeled least ordinary squares using OLS and Statsmodels, and then we calculated OLS using MLE methods. Thanks so much for your attention.

****

----------**END**----------