Module 6: Data Clustering and Principal Component Analysis (PCA)

Video Transcripts

Video 1: Singular Value Decomposition (SVD)

Before diving into model building techniques, we will take a moment to talk about two important algorithms that arise, not only in machine learning, but in other areas of science and engineering as well.

These are the k-means algorithm for clustering and the principal components analysis for dimensionality reduction. Both of these are considered "unsupervised" learning algorithms, in contrast to supervised learning, because they do not make use of the output in the training data. And in different situations, you might want to apply clustering to either the inputs or the outputs. So it is best to consider them simply as tools in your toolbox and understand what they do to data.

In both PCA and clustering, we will assume that the dataset consists of real numbers only, so no strings or categories. The data set has N rows, or samples, and D columns, or features. The goal of these two algorithms is to find patterns in the data. PCA finds patterns in the columns, and k-means finds patterns in the rows. But they are not simply transposes of each other. They are fundamentally different. PCA looks for new columns that are linear combinations of the existing columns, and which capture the bulk of the variation in the data. K-means looks for rows that are similar to each other and creates groups out of such rows.

We begin with principal components analysis. Consider a dataset with temperature measurements from 100 different cities. Each temperature is measured with two thermometers, one reporting in Fahrenheit and the other in Centigrade. These two columns are highly correlated with a correlation coefficient of 0.99, and we know that there's an approximately linear relationship between the two because we have a unit conversion formula: $1.8C - F + 32 = 0$. This relationship is not exact, of course, because there are measurement errors that jitter the data about the red line.

Let's say our goal is to train a model to predict some other quantity, say, humidity, from these two measurements of temperature. Building a model with two highly correlated inputs is not a good idea. We would rather use only one input because it takes a lot more data to properly cover a two-dimensional input space, than a one-dimensional input space. This is known as the curse of dimensionality, and it states that the amount of data that you need to train a model increases exponentially with the number of inputs. So it will be well worth our time to reduce the number of inputs, or the dimensionality, of the model. In this particular case, we would like to reduce the number of inputs from two to one. Here's an easy way to achieve this. Just throw away one of the two columns. After all, they both contain the same information, modulus the noise.

If we throw away column C and keep column F, this is like projecting the data along the green lines and we are left with temperatures ranging from about 50 to 90 degrees, a spread of about 40. Alternatively, if we throw away F and keep C, this is like projecting the data down along the orange lines and the resulting values will range from about eight to 32 degrees, a spread of about 24. Between these two options, we should prefer the former, because it results in a larger spread or variance of the data, and this

will lead to a more precise model. Well, this is assuming that the thermometers have equal precision, which of course may not be true, but let's ignore that fact for now, and stick with the idea that, generally speaking, it is good to have input data spread over a larger area of the input space. In other words, we want to maximize the variance of the input data. OK? But if our goal is to maximize the variance, the best approach will be to project the data not onto a vertical or a horizontal line, but onto this red line, which will give us a spread of about 45. The red line is longer than both the green line and the orange line. We know from the formula that the red line has an inclination of about 1.8, and intersects the vertical axis at approximately 32 degrees. I say about an approximately because in general, we will not have a handy formula such as this one to give us the best line. But we will have to compute it from the data alone. And this is what principal components analysis will do for us.

Principal components analysis finds the directions, projecting the data in a way that maximizes the variance, and are therefore optimal for building machine learning models. PCA does this not only in 2D as in this example, but for any arbitrary number of inputs. Let's see how it works.

PCA is built upon a technique from linear algebra called singular value decomposition or SVD. SVD is a matrix factorization with many applications apart from PCA. And because it is so useful, you will find implementations of SVD in many of Python's numerical packages including NumPy, SciPy, and scikit-learn. In this lecture, we will use the version from SciPy. This is very similar to the NumPy version, but it's slightly faster. Scikit-learn, which is a machine learning package, offers a direct implementation of PCA, which can be integrated into a machine learning pipeline, as we will learn in future lectures.

This is the formula for the singular value decomposition. X here is our input data. It has N rows and D columns. The columns are sometimes referred to as the features, and the rows are the samples. There will typically be more samples than features. So N is usually larger than D, And X is a tall matrix. Running SVD on X will decompose it into three matrices, U, Sigma (∑), and V transpose (V^T). Here, U is a tall matrix having the same dimensions as X, while ∑ and V are square D x D matrices. U and V are known respectively as the left and right singular vector matrices. Sigma is a diagonal matrix. This means that it has values along its diagonal only, and zeros everywhere else. The values on the diagonal are called the singular values. We will return to the interpretations of all of these quantities. But let's look first at how to compute them in code.

Video 2: Principal Component Analysis (PCA)

We'll begin as usual by importing our standard packages for numerical work, NumPy, pandas and Matplotlib. And we will also import some toy datasets from scikit-learn.

This time we're going to be using the Boston dataset. And here's what it looks like. We've loaded it and put it into a DataFrame. And what this dataset contains is characteristics, 13 characteristics, of 506 different houses in the Boston area. And this is used typically to predict the price of a house, given these 13 anonymous numerical characteristics. As you see here, I have not imported the target variable y. I don't need it, because we're only going to be working with the input data X in this case, because we're going to do PCA on it.

So PCA involves two main steps. The first is to normalize the data and the second is to perform the singular value decomposition on the normalized

data. Let's first have a look at a scatterplot of this data. So I'm going to call plot(kind='scatter') on, let's say, the sixth and the seventh column of this data. And this is what it looks like. OK? This data, two things to note. One is that the mean is not at the origin. So the mean of this data is around here in the 60s for column 6, and 2 point something maybe for column seven. And for PCA, we need the mean to be at 0.

In fact, we can see precisely what the mean is. If we say X.describe(). These are all of the means of all of the columns. So this as a vector is the location of the mean in 13-dimensional space, and this is the standard deviation. The standard deviation here is telling us that if we look here, the standard deviation for 6 is quite large because the data spans from 0 to a 100, in comparison to the standard deviation for column 7, which only expands from 0 to 10. And that difference can have negative effects on the numerical computation of singular value decomposition.

So the first thing we want to do is shift this data cloud to the origin and normalize it so that all of the standard deviations are one. Here we see the formula for the normalization of the data. We are going to subtract the mean from the data cloud to shift it to the origin. And we are going to divide it by the standard deviation in order for the normalized data to have unit standard deviation. So let's define a variable (mu) for the mean. And another called sigma for the standard deviation. And we can see here that the mean contains the same values as we saw up here, provided by the describe method. So now we have normalized data. We define it as (X−mu)/sigma. This is the DataFrame of normalized values. And we can create the same plot, the same scatterplot, but for the normalized data. We see that it looks qualitatively the same. Except now the mean of the cloud, which you'll recall is the balance point of the cloud, is at 0,0 and the spread

of the data is similar in the horizontal and vertical directions. We can also say Xnorm.describe(). And see that indeed all the means are very small values. So this is 10 to the minus 16, 10 to the minus 17. Essentially they're all zeros, and the standard deviations are all ones. So this verifies that the data has been normalized.

Once we do that, we can now perform the singular value decomposition on the normalized data. And we're going to use the version of SVD from SciPy linear algebra. So we import that. And now we call it svd on the normalized data: svd(Xnorm). OK, so if we run that we get a result. One thing that we're going to want to do is set full_matrices=False. If we don't do this, it's going to return matrices that are padded with zeros. And that's not what we want. So we set that to false and we're going to save the result into U, s, and Vt (V transposed). OK? So let's look at those three things. OK, I'm going to add a couple lines here. I'm going to define Sigma is equal to a diagonal matrix with the singular values s along the diagonal [Sigma = np.diag(s)] . And I'm going to define V as V transpose. So the reason I'm doing this is that SVD returns things that are slightly different from the notation, the convention that we have adopted. We have adopted Sigma to be a diagonal matrix. So I created that. And this actually returns, SVD returns the transpose of V. So I defined V to be the transpose. Sorry, that's Vt dot transpose [V = Vt.T]. OK?

Now that we have that, the next step will be to check that the decomposition is correct. So we can run this matrix multiplication by multipligma times V transpose (U @ Sigma @ V.T). This @ sign is matrix multiplication in Python. And we get something. We can compare that to Xnorm. These should be the same because the decomposition could, should give us exactly the same value. So we see here is –0.419, that corresponds to this. And we could go through this whole matrix and just

check that everything's the same, but that would be a lot of work. So instead we're going to use a function called allclose. And we're going to make sure that every entry in Xnorm is close to this product. And if that is true, then the two are the same. OK? So they are the same. In fact, U @ Sigma @ V.T is equal to Xnorm. One other thing that we can do is we can undo the normalization by inverting this, this formula. So X it will now be equal to mu plus sigma times Xnorm. So from the normalized data, we can recover the original data. So let's do that.

Mu plus sigma times Xnorm, which is now this (U @ Sigma @ V.T). But we're going to throw it into a pandas DataFrame. So let's call it DataFrame. And we put that in [mu + sigma* pd.DataFrame(U @ Sigma @ V.T)], and we get the original dataset. And we can also verify that these are close, that these are indeed the same thing. And if it's true, then we have verified that we can revert from normalized data to the original data with no loss in precision.

Video 3: Interpretation of Principal Component Analysis (PCA)

Now to the interpretation of U, Sigma, and V, and how they relate to our variance-maximizing goal.

Here you can see matrices U and V in terms of their columns, u_1 through u_D and v_1 through v_D. The columns of V are laying horizontally because they're transposed. The Sigma matrix is diagonal and its entries sigma_1 through sigma_D are the singular values. We can also express this matrix multiplication as a summation of D principal components, sigma_i, u_i, v_i transposed. Each principal component is a rank 1, N x D matrix, u_i v_i transposed, weighted by the singular values sigma_i. All of the D principal components combine to form the matrix X.

v_1 through v_D are the optimal directions for projecting the data. Each one is a direction in RD, the input space, and there are D of them. Furthermore, they are mutually orthogonal. So as a group, they form a basis for RD. In our 2D example, we would end up with two vectors, v_1 and v_2. And v_1 would be the direction with an inclination of approximately 1.8 that was the best 1D line for projecting the data.

The singular values sigma_1 through sigma_D, are organized from largest to smallest. So sigma_1 is larger than sigma_2 is larger than sigma_3 and so on. The value of sigma_i conveys the importance of the i'th principal component for the dataset. So the first principal component captures the largest amount of the total variance and each subsequent component captures a smaller share.

This is a very useful feature. It allows us to construct a sequence of approximations to the dataset X, starting from a very coarse approximation considering only the first principal component, up to the exact dataset considering all D principal components.

In between, we have approximations X tilde D sub r, with r varying from 1 to D. The superscript D is to indicate that this dataset is still in RD. We will project it down to the lower dimensional space in a couple slides. As a matrix multiplication, X r tilde r is obtained by keeping the r leftmost columns of U and V, and the first are singular values. Let's call these U_r, Sigma_R, and V_R.

Here are the singular values for the Boston data. The plot on the left shows singular values in order from the first and largest, to the last and smallest. This gives us a sense of the relative importance of the principal components, with the last few being practically negligible in relation to the

first. The plot on the right shows the cumulative sum of the singular values divided by their total sum. This gives us a quantification of the quality, or total variance captured, by the r'th approximation.

We can use this plot to decide on an appropriate level r for the truncation. For example, if we wish to obtain an approximation quality of at least 0.9, then the plot tells us that we must use at least four components. That is, we can decrease the number of columns in the input data from 13 to just four and still preserve 90 percent of the variance. This is definitely a good thing to do given the exponential nature of the curse of dimensionality.

The next step is to perform the projection of the data onto the lower dimensional subspace. This is done by multiplying the approximate data matrix in D dimensions, X tilde D sub r, by the matrix of principal vectors V_r. From the formula for the SVD decomposition, and the fact that V is orthonormal, we get that this is equal to U_r times Sigma_r.

In our 2D example, X tilde D sub r is the projection of the original dataset X, which was in two dimensions, onto v_1, the first principal component, which turned out to have a slope of about 1.8 as we predicted. In the left hand plot, the original dataset are the blue points and the projected data are the orange points. But these are still in two dimensions. Each orange point is still characterized by two numbers.

The dataset we will actually keep is X tilde sub r, the projection down to the lower dimensional space, which you can see on the right. This data is now in one dimension as desired. You can also see that the spread of X tilde r sub r is about 45 as we had predicted.

Let's go back to the code to finalize the computation for the Boston data.

Video 4: Principal Component Analysis (Continued)

Now we'll complete the principal components analysis by using our SVD decomposition of the data and projecting it down to a lower dimensional space. So we're going to start first by importing the packages and quickly loading the data, normalizing it and doing the SVD. And let's say we've decided that the number of dimensions that we want to use is four, that means r = 4.

The formula that we use for finding the lower dimensional data is this one.

$$\tilde{X}_r^r = U_r \Sigma_r$$

The data projected onto four dimensions equals U_r (which is the first r columns of the U matrix) multiplied by Sigma_r (which is the r x r upper-left block of the Sigma matrix). So let's define those. U_r is going to be the first r columns of U. U_r is the first r columns of U, and Sigma_r is the upper r x r block of Sigma.

```
Ur = U[:,:r]
Sigmar = Sigma[:r,:r]
```

And then we can take U_r and multiply it by Sigma_r. And this is a matrix multiplication. And that is going to be equal to our X_r. OK? To make it look nice, Let's just throw that into a pandas DataFrame and have a look.

```
Xrr = pd.DataFrame(Ur @ Sigmar)
```

There we have it. This is our 506 homes in Boston, represented with four numbers instead of 13 numbers. And these four numbers have been calculated very carefully using the principal components analysis to capture

as much information as possible from those original 13 numbers. So that is what the principal components analysis has given us.

The next thing we want to do is consider a situation in which we have been given some new data. Let's say there's a new home that is represented with these 13 numbers in the original 13-dimensional space. And what, what should we do here? Should we put this into our original dataset and rerun the whole principal components analysis again? Or can we just project this down onto the principal components that we've already computed? Well, it could go either way depending on this data. If the new home is sufficiently different from the previous ones, so that we'd expect that the principal components would come out differently, then maybe we compute everything again. But usually not. Usually you'd want to just project this into the existing space.

And to do that, you would use this formula: $\tilde{X}_r^r = \tilde{X}_r^D V_r$

The projected data is the original normalized data times the first r columns of the V matrix. So let's do that. First, what we want to do is normalize the new home information by subtracting the mean and dividing by the standard deviation.

```
newhome_norm = (newhome - mu)/sigma
```

And then we want to take that new home normalized, and matrix multiply it by the first r columns of V. And we're going to call that the new X r r, so new home projected.

```
newhome_proj = newhome_norm @ V[:,:r]
```

OK, we do that. And let's look at what that looks like. These are four numbers that represent our new home in the principal components that we had calculated earlier. And we can now append that to our existing DataFrame using loc and putting it at the end of the DataFrame, just like that. And let's look at that DataFrame to see what it looks like. We now have 507 homes and the last one is the home that we have just added to our dataset.

Video 5: SVD and PCA Summary

To summarize, we have described the principal components analysis which is used to reduce high dimensional data to a lower dimensional space in a way that preserves as much of the variability in the data as possible.

The computation of the principal components has two steps. First, center the data, then call SVD. This produces matrices U, Sigma and V. The V matrix contains the projection directions in the form of linear combinations of the original input columns. These are ordered from most to least significant, with the significance given by the value of the corresponding singular value from the Sigma matrix. We can choose the number of components by plotting the cumulative singular values and finding the smallest number that captures the variance that we wish to preserve.

We wrote our code using the SciPy implementation of SVD. We saw how to integrate new data by first centering it and then projecting it onto the smaller subspace.

In future lectures, we will see that scikit-learn simplifies a lot of this with its fit and transform methods. For now, we proceed to our second unsupervised algorithm, clustering with k-means.

Video 6: Clustering and K-Means

We have now seen that PCA is a method of reducing the number of columns in a dataset in a way that preserves as much of the variability in the data as possible. Our next topic is clustering, which is a method for creating groups out of the rows. Recall that each row is a sample or a point in a cloud, whereas each column is a dimension of the space that the cloud lives in. So reducing columns and grouping rows are two very different activities.

Let's say we are given this bunch of fruit and asked to create groups out of them. We have here a banana, an orange, a lime, an apple, apricot, watermelon, strawberry, cherry, peach, avocado, and kiwi. But we are given no further direction on how to create the groups. We don't know what criteria to use or how many groups to create. Of course, there are many ways to solve this problem. Perhaps there is no best solution, but certainly some solutions are better than others. Here are a few good options. We could create groups by color into green, red, orange, and yellow fruit. Or we could do by shape into spherical and non-spherical fruit, or by the number of seeds into fruit with no seeds like this seedless banana, with one seed or pit, like the peach, the apricot, the avocado, and the cherry. Or fruit with a few seeds like the apple, orange and lime, and fruit with many seeds, such as the watermelon, strawberry, and kiwi.

These are good choices and they share some common characteristics. First, they group fruit according to their proximity in some space, such as color space, shape space, or number of seeds space. What characteristics we use and how we define proximity is a crucial design decision when applying a clustering algorithm. Second, they form a moderate number of

groups or clusters. The extremes of creating a single group for all fruit, or a separate group for every individual fruit are similarly uninformative and useless. When we create a moderate number of groups, this can lead to some insight into the data, which is precisely the goal of clustering.

Here's an example of an activity which would not be considered clustering. Suppose the fruit were labeled according to whether they were GMO, organic, or neither. And you were asked to create groups according to this label. Well, this would just be a sorting problem, putting fruit into three different bins, and the labels would guide the process. Because clustering works without such labels, it is considered an unsupervised machine learning method. And clustering algorithms create the group labels where there were none.

This very nice figure is from scikit-learn's documentation on cluster analysis. It shows ten different clustering methods, k-means, affinity propagation, etcetera, applied to six different problems. All of these six problems are on 2D data, so they are relatively small. They are designed to demonstrate some of the strengths and weaknesses of the different algorithms. The human brain is very good at solving problems like these. Just glancing at the problem, we can immediately hit upon a solution. For the top problem, we see two rings, an inner ring and outer ring. It looks like Spectral Clustering, Agglomerative Clustering, DBSCAN, and OPTICS have delivered the correct solution in this case, but not the others. On the bottom row, we see how they all perform on a dataset that uniformly fills all of the space. Again, DBSCAN and OPTICS seem to do the right thing. From this picture, it seems that DBSCAN and OPTICS are the best performing clustering methods, and indeed, these are both cutting edge options.

K-means, however, offers a good entry point for understanding how the algorithms work. And k-means is also very popular because it scales well to large datasets. And it's perhaps the most widely used of the algorithms. So let's start there. K-means is a centroid based method. That means that each cluster is characterized by a point mu_k called the centroid of the cluster. Data points shown here in blue are assigned to their nearest cluster. So in this case, each point evaluates its distance to both mu_1 and mu_2, and then joins the cluster that it is closest to. You can imagine moving the centroids around and, and the assignments switching from one centroid to the other. The problem that k-means tries to solve is where to place the centroids such that the sum of all of the squared distance is minimized.

Summing squared distances to a point is also the formula in mechanics for finding the rotational inertia about that point. Hence, the goal of k-means is to minimize the inertia of the clustering assignment. Even though this is an easy problem to understand, conceptually, it is in fact, a difficult problem to solve. And no algorithm is known that can reliably find a best solution for this for an arbitrary dataset in some reasonable amount of time. K-means is a good algorithm for quickly finding suboptimal solutions to this problem. Now, which suboptimal solution it finds depends critically on how we select the initial placements of the centroids. We will return to this point later.

The basic k-means algorithm proceeds by iterating between two steps, an update step and an assignment step. The update step is where we adjust the positions of the centroids. In the first iteration of the update step, we choose the initial placements of the centroids. The simplest way to do this, but not necessarily the best, is to select centroids randomly from the dataset. That is, according to a uniform distribution. We will see later a better way to do this. In the assignment step, we classify or cluster the data

points with their nearest centroid. Subsequent iterations of the update step place each centroid at the mean of the data points in its cluster. Iterating this way is guaranteed to converge to a solution, but not necessarily the best solution.

Let's look at an example. This is the result after the zeroth iteration. We have placed four centroids at random locations in the data space, and assigned each data point to its nearest centroid. Each point is colored according to the cluster it belongs to. The inertia is the sum of the squared distances from the points to their centroids. In our starting iteration, the inertia is 4,537.1. In the next iteration, we move the centroids to the middle of their respective clusters and repeat the data point assignment step. In the process, we see that a few points have changed color. This point was previously with the blue cluster and is now switched to green. And a couple of the red points are now purple. Notice that the inertia has decreased from about 4500 to 684.8.

Let's take another step. This time the inertia went down only by about five to 679.5. And indeed only one point changed membership, from red to purple, in this iteration. Another iteration. Again, a small decrease, and one other point switched from red to purple. And in this iteration it seems that no points have changed color. So we have reached convergence and the algorithm is complete. Now, this may not be the solution you're expecting to get. The algorithm has created two clusters, a red and a purple one, from a single blob in the bottom right. That seems a little strange. It has also joined two blobs into a single blue cluster at the top. As we said, the solution is not necessarily optimal. Now which solution we arrive to depends uniquely on the initial placement of the centroids. So we can always try again and see if we arrive at a better solution with a lower inertia.

Let's do that. Here is another initial placement of the centroids. Let's see how it evolves. This run converged to something that, visually at least, seems more reasonable. It also has a lower inertia than the previous, so we can say objectively that it is a better solution. To obtain a good solution with k-means, it is not sufficient to do a single run of the algorithm. One must execute an ensemble of runs, each with a different randomly chosen initial condition. From the result of the ensemble, one selects and keeps only the best solution. OK. But until now we have been assuming that the number of clusters K was given. How do we select K? The most common way of doing this is to run the ensemble of k-means over a range of different values of K, say from one to ten clusters. And to keep the best one of those ensemble runs.

So let's try that. K = 1 is trivial. Everything is classified into one large cluster. And the inertia is maximal, 5,818.5 in this case. The opposite extreme is to set K equal to the number of data points. Then the total inertia is zero because every point can be assigned to its own cluster. Between these two extremes, the inertia will decrease monotonically with increasing K. Here we have solutions for different values of K. Notice how the inertia decreases as K increases. From these runs, we can construct the plots shown here. On the left, we see the decreasing trend of the optimized inertia with respect to K. We see that we get diminishing returns with each new cluster. While there's a huge benefit in going from one cluster to two clusters and from two clusters to three, the benefit of going from say, eight to nine clusters is very small.

We want to choose a value of K that is on that limit between large and small differential benefit. This optimal K is sometimes easy to spot in the inertia plot. And it looks like an elbow, or a sharp change in the slope. In this case,

the elbow is not obvious, but it becomes pretty clear in the graph on the right, of the percent differential improvement. Here we see that going from one to two, and from two to three, and from three to four clusters all decrease the inertia by over 150 percent. Beyond that, the benefit is much lower. So it's clear that the optimal number of clusters in this case is four.

Video 7: Clustering in Scikit-Learn

Now that we know how k-means clustering works, let's see how to run it in Python using scikit-learn. So we'll begin by importing our standard packages. This is a plotting function that we'll use later. And this is the data that we're going to be working with. This is data that I found on Kaggle for mall shoppers. And the setup is that you're a store owner and you'd like to learn more about your customers. And so you set up a rewards program and you collect data from 200 of your customers. Now this data includes their gender, their age, their annual income, and their spending score. And what you'd like to do is to see if there are patterns in this data that you can use to better target those customers. So one thing that I notice is that there's a CustomerID column in this DataFrame. We can use that as a handy index. So we'll do that. Also, I'm only going to be focusing on Annual Income and Spending Score. So let's only keep those columns. And here is what our data looks like. 200 customers, their Annual Income and their Spending Score. So let's draw a scatterplot of that. I can say plot scatter of, or I can actually do it with pandas, I can say X dot scatter, X dot plot kind equals scatter. And I want to pass into this, x is equal to the Annual Income and y equal to the Spending Score.

```
X.plot(kind='scatter', x='Annual Income(k$)', y='Spending Score (1-100)')
```

Here's the scatterplot. Already you might be able to start to see some, some clusters. Like there's a middle cluster in here, and it looks like it's surrounded by four, sort of peripheral clusters. So let's see if k-means will help us to identify those automatically. To run k-means, we're first going to create a KMeans object by running the constructor from the clusters package and passing in the number of clusters that we want to create. In this case, I'm going to say five. And also specifying a initialization strategy. I'm going to say random. And what random does is that it selects the initial centroids at random from the dataset. So that is our KMeans object and I am going to assign it to kmeans variable and run that.

```
kmeans = cluster.KMeans(n_clusters=5, init='random')
```

OK, now we have a KMeans object. The next, next thing we want to do is to run the algorithm. And we do that by calling the fit function on the KMeans object and passing in the data.

```
kmeans.fit(X)
```

So we can run that. And now the algorithm has concluded, it has created our clusters.

```
KMeans(init='random'. n_clusters=5,)
```

And we can see what are the cluster labels by looking at the labels attribute within the KMeans object.

```
kmeans.labels_
```

And here it is. It tells us that the first data point should be classified as class is 0. The second is class two, and so forth. We can now run, or plot a scatterplot with the function that I defined at the very beginning. It's called

myscatter. And I'm going to pass in the, the data and the labels, and see what happens there.

```
myscatter(X,kmeans.labels_)
```

So this is what k-means did. It created five clusters as requested. And those clusters conform pretty well to what we'd expect. We have a central cluster and we have some peripheral clusters. So aside from some weirdness here on the boundaries, it looks like k-means is doing a pretty good job.

Next, I'm going to talk about an improved k-means algorithm or initialization strategy called k-means++. And, we can use k-means++, all we have to do is change the initialization strategy to k-means++. And now we have a k-means++ object. I'm also going to chain here the fit function. So I can just call fit right here and pass in the data. And let's see what that does. OK, so now this has created the object and already run the algorithm. But that was a little boring. I'd like to see what's going on under the hood. So I'm going to pass in a flag called verbose and set it to 1.

```
kmeanspp = cluster.KMeans(n_clusters=5, init='k-means++', verbose=1).fit(X)
```

And if I run that, now I can see what the k-means algorithm is doing. First of all, I see that it's run several times with different initializations. And so it is running an ensemble of k-means algorithms. And so it, it has swept over many initializations. And for each initialization I can see here what the inertia is doing. So the inertia starts high and then ends lower. OK? And it seems like it's converging to an inertia about 44 thousand in each case. This one didn't do as well.

Alright, but what does k-means++ do, as opposed to the random initialization that just sets centroids to existing data points at random?

K-means++ tries to distribute these more intelligently throughout the dataset. So it picks the first centroid at random, and then subsequent centroids are chosen such that they are far away from centroids that have already been chosen. And in this way it avoids clustering initial centroids, which could lead to numerical problems. In fact, k-means++ works a lot better than random initialization. So much so that it is the default in k-means initialization. So you could remove this and get the same result because it's the default. Let's look at what the scatterplot looks like for my plus plus run. I'm going to put that here and I'm going to pass in the plus plus labels.

```
myscatter(X,kmeanspp.labels_)
```

And I get something that looks pretty much identical to the other, the plain vanilla k-means. And that is because this is a really simple dataset. But k-means++ can make a big difference on large datasets. So just keep that in mind.

The next thing we're going to talk about is DBSCAN. DBSCAN is a different clustering algorithm. It's called density-based spatial clustering of applications with noise. And we saw in the scikit-learn toy problems that k-means wasn't the best-performing of the clustering algorithms. And DBSCAN actually did a lot better. It was amongst the better performing and cutting edge of clustering algorithms. So we're going to learn about that in a subsequent video. But for now, all I just, I want to show you is how easy it is to go from k-means to DBSCAN. The syntax is pretty much identical. All you do is you change the constructor from kmeans to DBSCAN. You pass in the appropriate parameters, and you call the fit function, And it works just the same. So we can run DBSCAN, find our clusters according to DBSCAN.

```
dbscan = cluster.DBSCAN(eps=9, min_samples=3).fit(X)

myscatter(data, dbscan.labels_)
```

Now notice that the number of clusters is not among the parameters that we pass into DBSCAN. So DBSCAN selects its own number of parameters. And in this case, it has chosen to split this dataset into seven clusters. The five that we saw with k-means plus one blue one over here, and one yellow one over here. And also DBSCAN does not classify all of the points. So all of these red points are considered by DBSCAN as outliers. This can have advantages and disadvantages. I, for one, am not sure what this clustering means to a store owner, but I think it could be useful.

Next, I'm going to talk about prediction. Let's say we have two new customers. Let's say they were late in filling out their surveys. And one has an income of 30 and a spending score of 20, the other has income of 80 and a spending score of 20. And we want to incorporate them into our data. So this is the new data. We can plot that into our scatterplot and this is where our new customers fall. It would be reasonable to classify this customer as class 0, because it's smack in the middle of that cloud. And this one is, as class 1. Now it turns out that only k-means can do that.

K-means can predict or assign a class to a new customer with the predict method. So I call the predict method on my new data. And I get, as we thought, that one customer should be in the zeroth cluster and the other should be in the first cluster. So I emphasize just that only k-means can do this. DBSCAN does not have a predict method. So that's one advantage of k-means over DBSCAN.

Video 8: DBSCAN

The DBSCAN method avoids some of the pitfalls we've encountered with k-means. First, DBSCAN is a centroidless algorithm. This means that it does not require the initial step of randomly placing the centroids. And this eliminates randomness from the algorithm almost completely. As a consequence, we never have to run ensembles of DBSCAN. A single run will suffice. The lack of centroids also means the number of clusters arises naturally from the algorithm. We do not need to run algorithms for various values of k, and look for elbows in the inertia plot as we did with k-means.

A second clear advantage of DBSCAN over k-means is its ability to create curved boundaries between the clusters. In k-means, the area that defines a cluster is always a convex polygon. This limitation is apparent in the first two toy examples from the scikit-learn. K-means is incapable of identifying clusters that are not linearly separable.

Also, DBSCAN has a built-in outlier detection feature. Points that are sufficiently removed from the bulk of the data cloud are designated by DBSCAN as outliers and are not classified at all. Outliers show up as black dots in the scikit-learn examples. Here's how it works. DBSCAN takes two parameters, a radius epsilon and an integer min_samples. It begins by drawing a ball of radius epsilon around every point in the dataset. If the ball captures at least min sample points, in this case three, then that point is designated a core point. These are all of the core points that were found in this example. Core points are then joined to other core points in their epsilon neighborhood to form clusters. Points that are not core points, but that contain at least one other core point in their epsilon neighborhood are

called boundary points. These are included in the cluster of their neighboring core points.

Now it is possible, although it does not happen in this example, that a boundary point could be adjacent to two different clusters. In this case, it is assigned randomly to one of the options. And this is the only source of randomness in DBSCAN. The remaining points, those that have no core or boundary points in their epsilon ball are the outliers. DBSCAN will assign these a cluster number of –1. Of course, no algorithm is perfect and DBSCAN is sensitive to its two input parameters, the epsilon radius and the minimum number of neighbors of core points.

Ultimately, clustering is somewhat of an art. There are many other algorithms to explore and the more of these that you are familiar with, the better equipped you will be to choose the right algorithm for any given problem. This is a rich and interesting topic and I encourage you to dig deeper and play around with the algorithms available in scikit-learn and elsewhere.