Now let's talk about our

next clustering algorithm, hierarchical agglomerative clustering. With hierarchical

agglomerative clustering, we'll try to continuously split out and merge new clusters successively until

we reach a level of convergence. Now let's see how hierarchical

agglomerative clustering actually works. So here, we're using the same example

as before, and we're going to try and come up with our different clusters. For hierarchical agglomerative clustering,

we start off by looking at the points, and identifying the pair which

has the minimal distance. So notice here, again, that the distance

becomes a very important factor in the success of our

clustering algorithm. So we need to keep into account which

distance metrics we're actually using. We see that these two points that

we have in green are the closest so we color code them here to highlight that

these are going to be our first pair. And then we continue to do this,

again, looking for the next closest pair of points and

the next closest pair. And, we can keep doing this. But the next closest pair can

actually be a pair of clusters. So we might have two different clusters or

a cluster and a point that are going to

be closest to one another. It doesn't necessarily just have

to be two different points. Now how we define the distance

from a cluster to a point or from a cluster to another cluster

will depend on our linkage criterion. And, we'll expand on this a bit later. But for

the distance to a particular cluster, maybe it's going to be the average

of points in a given cluster and the distance to that

average of given points. Or maybe it's the minimal distance

between all points in a given cluster to that point. So just taking the minimum distance. And if it's a pair of clusters, if we do

find it's a cluster that is the closest point, then we can go ahead and

merge them into their own cluster. So we see that, again, we don't have

that but here we move one more step and we see that the blue and the green that we

had merge together into the two greens. And, we can continue to see that we

can create more and more clusters. And we keep going creating each

one of our pairs moving forward. And now we see some of

them merging together. Merging further together, we have those

red dots all creating their own cluster. And now the number of clusters

will start to reduce. As we keep moving forward,

each one of them combining together. And at this point looking here,

we're at six different clusters. We can run again and we get down to five clusters as we

continue to find the closest linkage. Now we're down to four. And again,

as we continue to move up that ladder, we can continue to merge these

different clusters together. And then we're at three different

clusters, at two different clusters, and if we were to continue this,

we can end up with one large cluster. So this means that if we

allow this to continue, eventually we don't have clusters. So we have to come up with some type

of stocking criteria when we're using agglomerative clustering.

With the idea of using the average distances of all the points within

their respective clusters, how do we go about actually

finding our stopping point? Let's say we're at this stage, and we have five

clusters as we see here, and each one of

those clusters are color coded as we move forward. At this stage, we can say that the average cluster

distances for each one of our clusters which we have marked here with the same colors that we just saw in that

two-dimensional plot. We have each of the

average distances, and with that we have our gray dotted line which

marks a point where we are going to stop once all of these average distances

are above that line. In the next iteration, we find that light purple

and magenta clusters are going to be merged. Therefore, that average

cluster distance for that particular cluster

should go ahead and increase. We can visualize this change in that average cluster

distance as followed. For that new combined cluster, we now have this average

cluster distance that we see that is higher

than the previous two. Before we had the light

purple and the magenta, we merge those to that

higher version of magenta, and we see that we have a higher average

cluster distance. Now, we only have four

remaining clusters and as a whole they're a bit closer to that limit set to that gray line. In the next step, we can have that purple

cluster is going to merge with the teal cluster

in that top right corner. The new cluster formed, combining that teal and purple is now above

that threshold. Now we don't stop at

this point though. We are only going to stop once the minimum is above

that threshold. The minimum average

cluster distance is still not above

that threshold, we still have the pink and

magenta below that threshold. Now, in this next step once

we move to two clusters, magenta cluster and

the pink cluster merge together to create

this new pink cluster. Finally, once we merge these two, all the cluster distances

are above this threshold. They are big enough to therefore claim that the algorithm

has finally converged. Now we mentioned earlier

that we would want to merge clusters at some point that are closest to one another, but that idea of which cluster is closer is a bit of

ambiguous concepts, especially when they're

going to be multiple points belonging to each one of

these different clusters. Now there are several methods to measure that distance

between these clusters, and these different

methods are called the different linkage types. The first example that we

have here is single linkage, and that's going to be the

minimum pairwise distance between our different clusters. Given that we have our different clusters that

we have here on our data set, it's going to be the distance between the two closest points, say one from the teal cluster

and one from magenta, and we can see the blue

lines that connect each one of these according to which is going to be the

minimum distance between a certain point in the magenta and a certain

point in the teal. We take that distance between

those specific points, and declare that will be the distance between

those two clusters. Then we try to find four all

these pairwise linkages, which one is the minimum? Then we would combine those together as we move

up the hierarchy. We will talk through many

different type of linkages. A pro to the single linkage or the minimum pairwise

distance between clusters, is that it can help in ensuring a clear separation of

our clusters that had any points within

certain distances of one another as clear boundaries, but a con of this single linkage will be that it won't be able to separate out cleanly if there's some noise between the

two different clusters. It'll be very easy

to be skewed by certain outliers falling

close to certain clusters. Now, another linkage

type is going to be called the

complete linkage. With complete linkage,

instead of taking the minimum distance given the points within each cluster, we would take the maximum value. Taking the furthest

distance from each cluster, and from those maximum distances decide which one is the

smallest and then we can move up that

hierarchy to reducing here from four clusters

down to three. Now, a pro of this

method is that it will do a much better

job of separating out the clusters if

there's a bit of noise or overlapping points

of the two clusters, unlike with the single linkage. But a con of this is that

it can tend to break apart larger existing clusters, dependent on where

that maximum distance of those different

points may end up lying. Alternatively, we can also take the average of all the

points for a given cluster, and use those averages or those cluster centroids as

we've been introduced to, to determine the distance

between our different clusters. Now, the pros and cons of using the average can be seen as an average between

the pros and cons of using the single and

complete linkage, and that it may also break

up those larger clusters and also may be a bit drawn towards a noise but also

do a better job than either the single linkage or the maximum linkage in

regards to the cons of each. Then finally we have

the ward linkage. The ward linkage is going

to compute the inertia. If you recall the

inertia is going to be the distance square between each one of

our different points and their centroids, and picks the pair

that's going to ultimately minimize

that inertia value. It's trying to minimize

that sum of squares of the distances to the

cluster centroids. In that sense you can

think of it as something similar to k-means

in trying to come up with a new combining of

the different clusters. Again, the pros and cons of ward will be similar to

the average and that they will balance out both the pros and cons of

the min and max linkage. Now, in order to do

this in practice, to do this in Python, it'll be very similar steps

to what we've seen so far. We will start by importing our class here called

agglomerative clustering. We're then going to create

an instance of the class, so we say agg equal to

AgglomerativeClustering. With our different

hyperparameters, we can choose the number

of clusters here, we set the number of clusters is equal to three so that it'll keep building up until we

get to three clusters. We then have the option to

choose our distant metric, and here you see we

chose euclidean, affinity equals the euclidean, we use the euclidean distance. We can also define what

our linkage will be. Going through the

different linkages that we just discussed are available, we can choose which

one we'd like to use for our current

clustering algorithm. Then as before we would

fit the instance on the data and use that to predict clusters

for our new data. Now let's recap what we went

over here in this section. In this section we introduced the hierarchical agglomerative

clustering method, and how we can use

it to slowly build up to larger and larger clusters. This method becomes very

useful in business practice, and you may want to also see the subgroups that built up

to these larger groupings. We then discussed

stopping conditions, and how you may either have

a predetermined amount of groups in mind or a predetermined amount

of clusters in mind, or you can say to continue

up until you reach a threshold of minimum average

of our cluster distances. Finally we went over

different linkage types, including single linkage using the closest points to determine distance

between clusters, complete linkage using

the furthest points to determine the distance

between clusters, average linkage and ward linkage, which finds the

combined clusters that most reduce the

amount of inertia. That closes this video on hierarchical

agglomerative clustering. In the next video

we're going to dive into our next

clustering algorithm, DBscan. I'll see you there.

Our next unsupervised learning algorithm we are

going to cover is density-based spatial clustering of applications with noise. The noise part is

going to be important as this is one of the

few approaches that truly clusters our data

rather than partitioning it and it will help us find outliers rather than putting them all into different clusters. We'll see how in just a bit. Now let's cover the learning

goals for this section. In this section, we're

going to discuss how this DBSCAN clustering algorithm actually works in finding

our different clusters. We will also discuss the input arguments and their importance for

determining our clusters, as well as discussing the outputs of our DBSCAN algorithm. Finally, we'll close out by

discussing the strengths and weaknesses of working

with the DBSCAN algorithm. So let's start here with a quick introduction

to what DBSCAN is. As we mentioned, a key part of this clustering algorithm is that it truly finds clusters of data rather than

just partitioning our data and thus works better when we have noise

in our dataset. We know that outliers

will show up in most of our datasets

and in reality, we should be able to create

our clusters and say that these outlier points do not belong to any of these clusters. Now, the basics of how DBSCAN works is that

we are working under the assumption that points

in a cluster should be a certain distance from one another within a

certain neighborhood. So we would randomly

select points from these higher density regions and slowly expand our clusters

and as we expand, we only include

points that are at a certain distance from the points that have

iteratively already been included within

that cluster, given that distance that we're

using from point to point. The algorithm ends

when no more points are of a certain distance

from the clusters already identified and thus all points will have been classified

as either belonging to a particular cluster or

otherwise they would be noise. Now this is all high level and

in just a few slides we'll make sure to visualize how this actually works in practice. Before we get to those

visualizations though, let's talk about the

inputs for DBSCAN as these inputs will be

of utmost importance to getting our clusters

identified correctly. First, as we've seen repeatedly with all our

clustering algorithms, we have to define the

distance metric used to define our similarity

between our different points. Then we have to

define the Epsilon. As we mentioned,

we are starting at random points and then

using those points, we're determining

if other points are within a certain distance

and if they are, they become part of the cluster. Now this minimum distance between the points is going to be considered part of

the same cluster if it's within a certain

Epsilon range. So that's going to

be our Epsilon. How far away a point needs to be, to be considered part

of that cluster. N\_clu, or often seen

as mean samples, which is actually the

argument used for Sklearn. These arguments, this input

will be the minimum amount of points for a

particular point to be considered a core

point of a cluster. Core points are going to be find by this N\_clu arguments and they're going to be find

as those points that have at least N\_clu neighbors,

including itself. So if we set N\_clu equal to 3, that means that that point has at least two other neighbors that are within that

Epsilon distance. A non-core point can

still be a part of the cluster if it's in the neighborhood of

that core point. But to understand this,

let's dive a bit deeper into the different classification of points given our DBSCAN model. So there are three possible

labels for any given point. First we have our core point, which we just defined as any

point that has more than N\_clu neighbors and all clusters will require at least

one core point. We then have density

reachable or border points. These will be points that can

be reached by a core point, but may have fewer than

N\_clu neighbors itself. These will still be a part of the cluster as long as they are in the Epsilon neighborhood

of a core point. Then finally we have noise. Noise is going to be a point that is not part of any cluster and that would be one that has no core points in that Epsilon

neighborhood of the point. If we have n clue equal to 4, and three points are within Epsilon and no others are nearby, none of these three are

going to be core points, and thus they are all going

to be identified as noise. Again, we'll visualize this a bit more clearly in the

videos to come. With those possible

labels for any point, we identify clusters as the connected core and density-reachable points

within our dataset. So that closes out this video, and in the next

video we're going to turn to that visualization

I keep promising that we're going to see

to clearly understand how the DBSCAN algorithm

works. I'll see you there.

So as promised, let's start to

visualize how DB scan actually works. And as we have in our past clustering

algorithms, we're going to start with this two dimensional data set, and we're

going to come up with clusters, depending on the visits and the recency and how

far away each point is from one another. So we start at a random point,

here we have this point in pink, and then we look at the radius

epsilon around that point, and we'll have to define that epsilon and

here we define as 1.75. And we look,

we create that 1.75 epsilon and we look around, and we see,

is there enough points, given our n\_clu within that circle,

to start a cluster. And we see that there are four points,

again, we include that point itself, even though, even with that point, we get up to

five, so we now have our first cluster. So every point within that epsilon is

going to be part of our first cluster. And then we process each new

point in the same way, so we move on to our next point here,

and anything within that radius within that epsilon radius,

gets included as part of that cluster. And we keep moving along. And then here we see that this point

while it is part of the cluster because it's near one of the core points as we

looped through we saw that the point to the right of this that it is

part of within that epsilon radius, was a core point with four points. This one is not a core point,

but it is density reachable. So it will be part of our cluster, but

we will highlight that this particular point is going to be border point,

a density reachable point, and not one of our core points, so we'll

leave that as part of a lighter pink. And we keep going down this chain, adding on points according to

those that fall within epsilon, keep running through and

we see these are all core points because they all have at least four points

including themselves in there. And we move along and

then this point only has three, so this one again is going

to be a border point but it is near one of the core points so it

will count as part of the cluster still. We see we highlight that in light pink,

and we see we can keep moving along, and eventually we have all of

our points within the cluster, we stopped the circle on all the points

and then if there are no neighbors left, we will randomly try

a new unvisited point, to potentially start a brand new cluster. And when we do that, here we start

with the blue we need to check is this going to be a core point once again. So we check again within epsilon of

this new random point that we sort out, we see that it is a core point,

bow we have started our new cluster. Now, this point again is going to be that

density-reachable point, but it will still be part of the cluster because it's

near another point that is a core point. And we can continue to move along

to build out our cluster here, and you see again we have a density-reachable

point, we've had a couple so far, but all those are near core points, so they

still are going to be part of our cluster. And then we see here that we

have with n\_clu equal to 4, we only have three within this cluster, so this is going to be a density

reachable point, but not a core point. And then when we move over to this

point over here, we see that the only one within that radius is going to

be that density reachable point, so there's no core points within this radius. So if there's no points within this

radius that are not core points, then this becomes a noise

point it becomes an outlier. So this isn't part of

either of our two clusters, and it's labeled as an outlier point which

is why we have marked it here in gray. Now, I want you to take a moment, and

given that DBSCAN method that we just walked through, notice which points

tended to be the core points, as we have them labeled in a darker hue. Which ones were those density reachable

points which are still part of our cluster, but don't have the number

of points that make it a core point, given our n\_clu, and then which

point we have labeled as an outlier. Now that we understand how

the DBSCAN algorithm works, let's discuss some strengths and weaknesses

of working with the DBSCAN algorithm. So as we saw, with the DBSCAN algorithm,

we'll not need to specify the number of clusters as DBSCAN will

automatically determine the clusters, dependent on how close

points are from one another. It also allows for noise and will not automatically determine the

outliers are part of a particular cluster. It'll also do a strong job of handling

arbitrary shapes as it's going to be searching out points that are within

epsilon distance of one another, and will stop whenever a gap

occurs no matter what that boundary shape between the clusters are. Now some weaknesses, it's going to

require two parameters, which means we need to search over more possible

values to find that optimal solution. Also those hyper parameters can be

very difficult to fine tune in higher dimensional space. And then finally will not do well

with clusters of different density. So even if we have two clear groups,

if for one group the points are about

five units away from one another, and the other is one unit away

depending on our distance metric. Depending on that distance between

our two clusters that are, on average five units away or one unit

away, it may be difficult to determine the differentiation between

those two clusters. Now, let's walk through how the DBSCAN

algorithm can actually be used using Python, so first things first we import the class

containing our clustering method. So from skleam.cluster, we import DBSCAN,

we then create an instance of that class, and pass in the necessary

hyper parameters. Here we're setting eps=3, and

the min\_samples=2, so that's that n\_clu that we've been talking of an epsilon

is the epsilon we've been talking of. That distance from every single point

in order to include it as a core point, or within the cluster. We're then going to fit that instance

on the data, so just calling db.fit. And then we can't call db.predict because

of the way that the algorithm actually works, if you recall it's the finding

the points iteratively by scanning through each one of the different

points within that data set, so it's just creating clusters

within that fitted data set. You can't call predict with the DBSCAN. If you want it to fit on a larger data

set, then you just include it in that fit, and then you can come up

with the different clusters. So we get our db.labels, and

just to note for those labels, we're going to have Class Zero class one,

and if there's going to be an outlier, any outlier as we saw can happen

with DBSCAN, will be labeled -1. Now let's recap what we

learned here in this section. In this section we discuss

the DBSCAN algorithm, and how we'll come up with its

own clusters dependent on which points are within a certain

distance of the other points. We then discuss the inputs and their

importance, specially that of the epsilon and n\_clu chosen, as well as the outputs,

and understanding the difference between a core point, a density reachable

point, and just outliers, or noise. And finally, we discussed some of

the algorithm strengths and weaknesses, such as it being able to better determine

clusters, or arbitrary shapes, but perhaps having difficulty determining clusters,

that may have different densities. Now, this closes out our discussion

on DBSCAN, and in the next video, we'll introduce our final clustering

algorithm, the mean shift clustering. All right, I'll see you there.

Here, we will be discussing our

final clustering algorithm, the mean shift algorithm. Now let's go over the learning goals for

this section. In this section, we're going to cover

the mean shift clustering algorithm and how we use the concept of moving towards

the highest density to help determine our different clusters. And then we're also going to

discuss the strengths and weaknesses of working with

the mean shift algorithm. Now, the mean shift algorithm works

similarly to k-means, in that we will be partitioning our points according

to their nearest cluster centroid. For k-means, though, the centroid represented the mean

of all points within that cluster. Well, with mean shift, that centroid

is going to be the most dense point within the cluster, which in

principle can be anywhere in that cluster. And the algorithm will assign

points to a cluster by moving to the densest points

within a certain window. So, how do we calculate this local density

to say where the highest density point is? In order to do so, we're going to calculate

the weighted mean around each point. So what do we mean here when we

are asking for the weighted mean? We can think of the weighted

mean as assigning more weight to those points closer to

the original point within our window. So say we select this black point

to start, we calculate the weighted mean in the local neighborhood, or

within this window, this pink square. And it would find that

the densest point given the weighted mean would be here in pink. And note on the side that the new mean

does not have to be at a data point and can be somewhere else within this window. So how do we go about using this

to create our different clusters? So the steps are going to be that

you choose a point and a window. So we saw that window size,

start at a random point. We calculate that weighted

mean within that window. And then we shift the centroid

of the window to the new mean. So we shift that square so it's now

perfectly around that new weighted mean that we just found, that new denser point. We then continuously repeat steps 2 and

3 until convergence, until there's no shift, meaning that we

have reached the local density maximum. And we'll call this the mode,

so when the mode is reached. And then we repeat steps 1 through 4 for

all data points until, finally, data points that lead to the same mode will all

be grouped together in that same cluster. So let's visualize how

this is done in practice. So let's visualize how this

actually works in practice. So we start with the centroid

at a given point. And then given that window,

we sample that local density and then we follow the gradient

towards the denser direction. So we keep moving towards

the highest density. So we keep reclaiming where that

densest point is, and we create our new window around it, and we see we move

along each one of our data points, until ultimately, we find that local

density maximum and we stop there. We can do this again

starting at another point. We can sample the local density. And again, follow that gradient

towards the denser direction. And we see that we move along

towards that densest direction. And again, we end up finding

that same local maximum, so we would assign those

both to the same cluster. We can do this again

starting at another point. This time starting further away at

a point that will probably lie outside this cluster. We sample that local density and follow

the gradient towards the denser direction. And you see that it moves along as we

move towards that denser direction. And then it finds that local

density maximum and stops there. And to keep going, we can start at

each one, the different points, sample that local density, follow the

gradient towards that denser direction. And here again, we see that that point

finds the same local maximum, so we would end up labeling

it as the same cluster. And we keep going like this. And eventually, it's going to find,

for us, four unique local maxima. So we see them laid out here,

each one of our four local maxima. And it's going to assign the points

to the centroids that they fall into. So we see here that all the pink

fall under that pink centroid. We see all the teal values falling

next to that teal centroid. And all of the blue values

falling under that blue centroid. And now, we have, as well,

the purple with its purple centroid, and we have our four different clusters. And no cluster number is needed, or any

distance parameters need to be defined. It's just going to move towards

that densest direction and figure out those clusters for us. Now let's hone in a bit into what

we mean here by this weighted mean. That mean that we keep moving towards

as we get higher and higher density. So that new mean is going to be calculated using the sum over points

within the window. And we see this in both the numerator and

the denominator. We're also going to have this weighting,

or this kernel function. That's going to allow us to give

a certain weight according to how far each one of these different points

are from the previous mean. And we see that in the numerator,

we weight this according to each point. So we're going to weight that and

then take the distance of that point. And those that have a higher distance or a

lower distance will have a higher weight. And the common kernel that's used

is going to be the RBF kernel, which is going to be similar

to your Gaussian kernel. Giving more weight, again, to those values

that are closer and less weight according to the normal distribution for

those values that are further away. Now let's talk about some strengths and

weaknesses of working with the mean shift. The mean shift is model-free. It does not assume the number or

the shape of each one of our clusters. So that's going to be a pro that we didn't

see when we worked with something like k-means. We can use just one parameter. We don't have to tune over more than

one parameter like we did with DB scan, that parameter being the window size or

the bandwidth. And it will be robust outliers. We have that window size and

it won't be affected. And it can have those outliers outside

of each one of our different clusters. Some weaknesses, the results will

heavily depend on our window size. So it's going to depend on

the bandwidths that we choose. And selection of that

window of that bandwidth is not going to be an easy

thing to decipher in general. And also, finally,

can be slow to implement. The complexity is going to be proportional

to mn squared, where m is going to be the number of iterations that it has to

do, and n, the number of data points. So the more data points that it goes,

it's going to be more and more complex. You see that's n squared complexity. So if we have a large data set,

this may take a while to converge. Now let's walk through the syntax that

you need in order to perform mean shift using Python. So first thing that we want to do

is import the class containing that clustering method. So from sklearn.cluster,

we import mean shift. We then create an instance of this class,

setting ms = MeanShift, and we pass in our parameter, bandwidth=2. So again,

our window here will be equal to 2. And then we fit the instance on the data. And we can use that to predict

clusters for new data. So we call ms,

that instance of our class, .fit, on X1, so it finds our clusters using X1. And then we can call ms.predict

on X2 to see which clusters they fall under given the new data. So to recap, in this video, we talked

about the mean shift clustering algorithm and how we used the concept of using

a window, as well as the densest point within our window, to find our

different centroids of our clusters. And we discussed the algorithm's

strengths and weaknesses, such as not needing to define

the number of clusters, as well as understanding that this model

will have a higher overall complexity. So with that, we close out our

different clustering methods. And in the next video we will compare and

contrast all the different methods that we discussed and which ones are best

to use for which use cases.