In this video, let's briefly bring

together the different clustering algorithms that we've introduced and

discuss some of the pros, the cons, and the use cases for each one. So, what will we cover

here in this section? In this section, we'll go over a review

of the clustering approaches that we went through throughout this course. We'll then summarize and compare each

one of these different approaches, as well as providing some guidelines for choosing which approach is best given the

business case that you're working with. So, let's review the clustering algorithms

discussed in the course so far. First we have K-means. And recall that with K-means we were

going to have to pre-determine that number of clusters that we're looking for. And once we do so, our clusters will

depend on coming up with some mean value that is trying to reduce

the distance from our centroids or that mean of that cluster to each one of

the different points within that cluster. With that in mind, we will get the result

that we see here for the shapes given. And we see that it doesn't

do a perfect job of getting shapes that aren't necessarily spherical. And we're going to dive a bit deeper into

the pros and cons of each, in just a bit. But this is just a recap and an intro to what we're working with, with

each of the models that we had introduced. So, next, we have the mean shift, which

does not require us to set that number of clusters as we had to do with K-means. But, rather, we'll iteratively

move towards those densest points, given a window, and we'll get the results

that we see here under mean shift. And notice that for

both K-means and mean shift, they're going to heavily favor

more of a spherical shape and may not have quite the flexibility

to find different shapes. Next we have ward. And what we mean here by

ward is the agglomerative hierarchical clustering with ward as

the linkage type between our clusters. Recall that ward linkage specifies

distance between clusters as the new combined

inertia of those clusters. And since we're linking closest clusters,

when we work with hierarchical clustering, we have a bit more flexibility in

combining clusters of different shapes. But some noise can throw this off, as it

did in our two circles example above. And while we can set means of

how we want clusters determined, they do not quite get determined on their

own accord as we saw with mean shift or as we will see in this

next one with DBSCAN. So, finally we have DBSCAN, which will

find those points which are closest to one another in order to create those clusters. And this will both create

its own clusters, so you don't have to predetermine

the number of clusters, and be able to identify clusters

of different shapes. Now, this may seem like DBSCAN should

always be the one to go with, but we'll dive a bit deeper into what can

make DBSCAN a bit more difficult, and at times not the ideal candidate. So let's dive deeper,

starting with K-means. With K-means, if we use MiniBatch

to find our centroids and clusters, this will find our

clusters fairly quickly. So it'll run with fairly low complexity

compared to the other models. If we don't already know how

many clusters we're looking for, with K-means we're going to have to search

through our k values and use something like our elbow method that we introduced,

to determine that number of clusters. It'll generally be a bit more skewed to

finding even sized clusters when we work with K-means. And it's not going to work well

with non-spherical cluster shapes, as we'll be looking at distance from the

centroid in every single direction as we move towards that mean. And, therefore, we'll only be able

to find more spherical shapes. Which is why it doesn't do a great

job with these different shapes that we have here. Next, we have mean shift. And with mean shift,

we do not have to guess k. The number of clusters

will be determined for us. Also mean shift will do a fairly good job

of finding uneven cluster sizes that will simply be moving towards that highest

density, given a specified bandwidth, so we can find uneven clusters. They don't have to be even in any means,

such as what we saw with K-means. Now, it can be slow with a lot of data. We said that K-means with

the MiniBatch can run very fast. The mean shift can tend to be a bit slow,

if we have a lot of data, as it's going to be searching for points for highest

local density for every single point. It will do a good job of finding a lot of

clusters, if they exist in the dataset. So, if you think that there are a lot

of clusters, this may be a good choice. It will not do a great job

of finding weird shapes as, again, we're looking for closeness in

every direction within a certain window. So, it'll tend to go towards

more spherical shapes. And it will be limited to using the

Euclidean distance within its formulation. So, we don't get to use

these other metrics, these other distance metrics that we

introduced earlier in the course. Now we move to a hierarchical clustering,

here with ward. And the strength of hierarchical

clustering really comes into play when we want to get a full hierarchy tree,

and see how some groups may

be subgroups of others. Now, you do have to come up with

some means of deciding the number of clusters on your own. Whether that's choosing the numbers

directly, or with a minimum average distance threshold, as we saw in our

course on hierarchical clustering. It will often find uneven cluster sizes,

as we can easily have a tiny cluster of one or two points that

are far away from the rest. There going to be many different

distance metrics and linkage options that can be chosen, which may make it

difficult to fine tune this type of model. And it can end up being very slow to

calculate as the number of observations increases. So, this also will have

fairly high complexity. Now, with DBSCAN, it seems you can

often get the best of both worlds, if you choose the right parameters. But finding those correct parameters

can prove to be a difficult task. Now, with DBSCAN, it'll be able

to find clusters of uneven sizes, as long as it reaches the and

clu amount that was predefined. It will create a new cluster. Assuming, again, that you have if n

clu's equal to 4, as long as you have 4 points within that epsilon radius,

you will create a new cluster. It will work with distance

metrics of your choosing. So, you're not limited to

just Euclidean distance. DBSCAN will be able to easily move

along a cluster in small steps, thus being able to find

clusters of uneven shapes. Now, there is a danger if you

choose to small of an epsilon, that you will have too many clusters,

which is probably not ideal or trustworthy for most business cases. And, finally, the main disadvantage is

that it can have a great difficulty determining clusters of

different densities. Now, to bring it all together, I would

say, take a look at this page if you're ever trying to decide which one of the

different clustering approaches to use. If you look at the parameters for K-means,

you just need to choose the number of clusters, mean-shift, bandwidth, which may

be a little bit difficult to fine tune. For hierarchical clustering, you choose

the number of clusters, but you can also visualize the clusters that are created

as they grow one on top of the other. So, it becomes a bit easier to

choose that number of clusters. And then the neighborhood size could be

fairly difficult to choose when you're working with DB scan. Now the scalability of each. With K-means, you can scale to

very large number of samples. So very large data, probably want a medium

amount of clusters not too many clusters. And this is both using MiniBatch,

which will help speed things along. Mean shift will not be quite as

scalable with the number of samples. So, as we increase the number of samples, it tends to take quite some

time the complexity increases. With hierarchical clustering, you can use

large, so not very large like K-means, but large number of samples,

as well as a large number of clusters. And then DBSCAN, again, will scale quite large number of

samples and a medium amount of clusters. Now, we have here the different

general use cases. But I want to skip more

to the applications. All I want to highlight for

the general use cases is that, again, with DBSCAN, you can also use it for

outlier detection. Unlike the others, it'll do a good

job of determining those outliers. Now, in regards to the applications,

we have that's for K-means, you can find few clusters

of roughly the same size. I would say this is a quick and dirty way. If you know the number of

clusters that you're looking for, then this may be a good way to get started

in your clustering of your dataset. With mean shift, you can identify

the number of clusters on its own. So, if you don't know the number of

clusters, this is a good choice, often used in video. And also, again, if you don't know

those clusters, this is a good for business case, especially if DBSCAN

may be difficult to fine tune, or if you have clusters of

different densities. And then hierarchical

clustering will be good for business cases where you may want

to find the subgroups as well. So, if you don't just want the groups but the subgroups that built

into those groups. And then, finally, DBSCAN, that's often

used for computer vision applications, but also for business cases where you don't

know the number of clusters, and they are of similar density, then you can use

DBSCAN to identify those clusters for you. So, just to quickly summarize. What we went through here,

were different clustering techniques where clustering is just unsupervised

learning, meaning we don't have labels. But we can come up with

groupings of our data to see if there's different segments of

our data that can be clumped together. And we discussed several approaches

that were possible such as K-means, Hierarchical Agglomerative Clustering,

DBSCAN, mean shift. And all this can be

implemented using scikit-learn. And if you're interested in learning more

about the different hyperparameters that can be passed through, or

even more clustering methods, I would suggest looking at the link and

feel free to dive deeper and experiment with everything

that you have there available. Now, just to recap. In this section, we had a review of the

different clustering approaches that we've discussed throughout this course. We summarize and compared each one of

the different clustering approachlabes, and then finally provided some guidelines for

choosing which approach is appropriate for the given business situation. That closes out our section

here on clustering, and in the next videos we will move

on to dimensionality reduction. All right, I'll see you there.

Clustering approaches summary:

* k-means, mean-shift, hierarchical clustering, DBSCAN

Lab

Welcome to our lab here on

different clustering methods. We have here the same dataset

that we used earlier on, and that we will be looking

at the wine quality. The data contains various chemical properties of the wine, such as acidity, the sugar

levels, the pH levels, the alcohol levels, and also

contains a quality metric, 3-9, with nine being the highest, and the color,

either red or white. We're going to be using

these chemical properties, so everything besides

quality and color, in order to cluster our wine. With that in mind, we can see actually to see whether or not our clusters will relate to the cluster of either

red wine or white wine. We'll see that in practice

later on in this notebook. First thing that we want to do is import all the

necessary libraries, we're also going to change

our directory here to data, and we're going to

pull in our dataset, winequalitydata.csv. We call data.head and we

take the first four rows. You see here that

we transpose it, and this is just for

a bit of readability. So really fix acidity,

volatile acidity, those are going to

be our column names, and we get a quick peak at each one of our

different columns. What may stand out is the different scales of each one of these different features, as well as the fact that color

is going to be a string. The rest are numerical

and this one is a string, and this will come into

play in just a second. We then look at the shape

of our data and we see that we have 6,497 rows, with just 13 columns. Then we're going to look

at the datatype like I just mentioned for each one

of our different entries. The reason why this is

important is because for k needs to work as well as

scikit-learn algorithms, we're going to need for our

data to all be numerical. Otherwise you won't be

able to pass it through. When we check this out, we are fortunate to see that

all of our values, again, are not going to be using quality or color when we

create our clusters. All of the other values

are going to be floats, they are not going to be objects like we see with

color or even images. We're then going to check

the value counts for our different wine

colors as well as for our different qualities. Again, those qualities

ranging from 3-9. So we check the color and we see the majority of our dataset

is going to be white wine. We can even call, as we've seen earlier,

normalize equals true, and we can see that a bit over 75 percent of our dataset is going

to be white wine, whereas 24.6 is going

to be red wine. Then we can check this

out in terms of the value counts for the quality, and we see that the

majority of our quality is going to center around that 5, 6, and 7 value with very

few very low quality, and very few very high quality. Now, we want to look

at a histogram, breaking down the quality by red and white wine

given our dataset. So we're going to first initiate these colors,

red and white. Now these are just going to be objects pointing to

a certain color. So from SNS, that's seaborne, we're going to pull

our color palette, and we want red to be associated

with the second value or the third value because

it's Python indexing, and then the white object

pointing to the color palette, and the fifth value. We're then going to explicitly tell our histogram what our bin range

is going to be. We don't want to combine

any one of our values. We saw that nine up here

could end up being very low, especially when we split it

between red and white wine, and we want to ensure that

we have a separate bin for every single one of our

different quality values. With those different

objects created, we're then going to

initiate our axes, so our bounding box

using plt.axes. Then we're going to zip together this list

of red and white, which is just a string

of red and white, as well as the red

and white colors that we initiated up here. So red will associate with red in that first iteration

through the fore loop, and white will

associate with this white in the second iteration. That will be color for

the string and plot color for these different

objects that we have here. We are then going to take

a subset of our data, so we say data.loc. We want to locate where the color is equal to the color

that we have specified, either red or white. Then we only want the

column of quality. We're just taking a

histogram of the quality. We then call it q\_data,

that's our subset.hist. We say again, we want to use just the bins that

we specified above, so 3, 4, 5, 6, 7, 8, and 9. We set Alpha equal to 0.5 because we want

it to be somewhat see-through as we will be plotting one histogram

on top of the other. We then set where do we want

to plot this ax equals ax, that axis that we

initiated earlier. Then the color that

we're going to use is going to be that plot color, which is either going to

be this red object or this white object that

we had defined up here. Then we're just going

to create a label for our legend later on, labeling the white

as white and the red as red using this string. We then want our legend. We want our x label and y label. We set our x limits, our xticks are going to be in-between each one of

these values so at 3.5, 4.5, and so on. Our labels are just going to be the different bin range values, 3, 4, 5, and so on. So you run this. We can see here our different breakdown of

red and white wine and see that red is slightly more

centered around this 5, 6. Hence the white wine

has a higher peak at that six or more

values of six, somewhat of a normal

distribution. Whereas red is going to be

a little bit flatter with a bimodal five and six in regards to the

red wine quality. We're then going

to in question 2, examine the correlation and skew of our relevant variables. So Everything except

for color and quality. We're not going to drop these, but we do want to exclude

these when we look at our cross-correlations

between each one of our different values. That's because we're

going to be using our clustering algorithms without either of these two values. Then on top of that,

we're going to perform any appropriate feature

transformations or scaling. Now what's important is we have to recall that we are using distance metrics when we use our k-means or any one of

our clustering algorithms. Something that wasn't

mentioned throughout lecture is the importance of

width distance metrics. This should have already

clicked as we were thinking through what we've done with each one of our supervised

learning models. If we are using distance, it'll be of utmost

importance that each one of our

features are going to be on the same scale. We don't want any one of our different features being more heavily favored or causing further distance

than the other one. So we just want

their variation to be changing what

our clusters will look like rather than

their actual magnitudes or their built-in values. Then we're going

to finally examine the pair wise distribution

of the variables with pair plots to verify the scaling and the normalization efforts that we went through. We're also going to

make sure that there's a normal distribution

that just makes things a bit cleaner so that we don't have

a heavy skew in one direction when we take each one of our

distance metrics. So we're specifying here that our float columns

are going to be all of our columns except

for color and quality. We're then going to create our

correlation matrix by just specifying that we only want those columns and calling.corr. Then finally, just to make

sure that we are not seeing each one of our difference

correlations with themselves, which would obviously have

a correlation of one, every value with itself will

have a correlation of one. We're replacing them

across that diagonal, so for x in range of the

length of our float columns. However how many

columns we're doing, we're going to replace

the diagonal, so.iloc. If you think each these

values being the same, that we are going to be

zeroing out each one of the different values in our correlation matrix

along that diagonal. So let's see what

this looks like. Again, you see that 0, 0, for fixed acidity and this will come into play because

we're going to look at the highest correlation between each one of our

different features. We can see here it's a

little bit difficult to quickly visualize what

those highest values are. To get that pairwise

maximal correlations, we are going to

call corr\_mat.abs. So we're taking the

absolute value. Because whether it's negative

or positive correlation, we want to see if they're

highly correlated, and then we call idxmax. We see that fixed\_acidity is most highly

correlated with density. We could also even just call.max if we want to see what

the maximum values are. We see some high correlations

between certain values. The reason why this is important, is if you recall when we discussed earlier in the lecture as well as in the last lab, if we had high correlation

between different values, then we start to reach that problem of high

dimensionality. We know that, that causes a problem whenever

we're working with distance metrics as we are with most of our

clustering algorithms. So we're not going to

do anything there. There are some fairly

high correlations, but not high enough for us

to exclude certain values. But that is the

reason why we want to start to investigate how high the correlations are

between each one of our different values

that we're going to use for our

unsupervised model. We're now going to

look at the skew of each one of our columns. We can just call for

those float\_columns. We're looking at the skew,

we just called.skew. Now you recall that

zero means no skew, positive value

means a right skew, a negative value

means a left skew, meaning it's not

normally distributed, Right skew means

heavy right tail, left skew means heavy left tail. Then we're going to

sort those values from highest to lowest. Then we're just going to take

those that are above 0.75. We look at that and we see

that each one of these values, we are saying that above

0.75 has a heavy skew. In order to help to

correct that skew, or just saying for col on each

one of these skew columns, only taking the index values. Because we don't care

about the actual values, we just care about each

one of these column names, or going to change that data to the log

version of itself. That will help

normalize our features. We run this and we've replaced all of our

different columns. Then on top of that,

as I mentioned before, it's of utmost importance that all of our features

are on the same scale. In order to ensure this, we are importing our

standard scalar as we've done before from

sklearn.preprocessing. We set sc to that

StandardScaler object. We call fit\_transform on all

of our float \_column data, so we're going to replace all

of our data float\_columns. Then we're going to investigate

this briefly and see that our values all are

on a similar scale. Finally, just to make

sure that we get a visual of what these

actually look like, we're going to run the pair. In order to run our pairplot, we want all of our

float\_columns as well as color. The reason why we want color is because we are going to break apart our scatter plots as

well as our histograms, and our pairplots

will show by color, to start to investigate that

natural differentiation between our different features

and the color values. We're just ordering

it white and red, and then we're just

saying our palette here. We want read to be equal to the red that we defined earlier. Then for white, we're going to use gray as the coloring for it. Now I'm going to run this. Pairplots generally take

a bit of time to run. I'm going to pause the video real quick and we'll come back

once it's already ran. Now, we ran this pairplot, and we did that to see the relationship between each one of our different features. But on top of that, because we're also looking at the breakdown between

red and white, we can look at

these two features. Again, we have more

than two features or two dimensions to work with. One, where we create our k-means. But even with these two features, we can begin to see that

there is somewhat of a clustering between the

red and the white wines. We can see that they're probably will be a pretty clean

classification given our data that will show us

which wines are red and which are white without actually having those

labels available. Now that closes out this question number

two and our video here. In question number three, we will start to fit

a k-means cluster, and see what kind of

clusters we actually come up with without the

labels to our data.

Now for question number 3 here, we're going to continue by

fitting our first k-means clustering model and we're

going to use two clusters. We're going to use two clusters not identifying

the red and white, we are not going to

have that included in our dataset and we're

going to examine the clusters according to the red and white

wine to see if that automatically

clusters according to this red and white

differentiation. So what we do is we import from sklearn.cluster,

our k-means model. We are then going to

initiate our model and say that we want two clusters,

recall with k-means, we need to say how many clusters

we want and then we call km.fit on just our float columns. So not including both

the quality column or the color column. We then call km.predict on those same columns

and we set that equal to its own column

within the dataset. We'll see why we do

that in just a second. Once we do that, we

can call data.head. We can see all the way

here at the end that we create this new column

that's either zero or one. We're going to see

how that relates to this color column to see if all the reds

were identified as zero and all the whites as one. So in order to do that, we're going to only take the subset of columns

of color and k-means, k-means being the

one we just defined. We're then going to group by

each one of these objects, so we're aggregating

by both of them. Then.size is just going to give us the count of that breakdown. Now it's going to

be a Pandas series, so we're just changing

it to a DataFrame and we're renaming that

column at first, by default, it will name

the column at zero, so we're just calling it number. We run this. We can see that for zero, the majority of them are

going to be that red wine, with only 87 white being

identified as zero. For white wine, for one, only 23 were identified as red, and 4,811 were

identified as white. So we can see that it

did a pretty good job without any labels separating out our dataset into two

different clusters that are very highly related to our

red and white clusters. Now we're going to fit a k-means model with clusters

ranging from 1-20. Now with this, we are assuming that we don't know

the number of K that we want, we don't know how many

clusters we want and for each model we're going

to store the number of clusters as well as

the inertia value. Then we're going to plot that cluster number versus the inertia and

see if we can find that elbow that

identify that this would be the best number of

clusters given our dataset. So we start with an empty list and then we range from

values from 1-20, we call k-means and we

initiate with that number. We then fit it on our float

columns and then we take our KM list and

then keep appending on this pandas series that

will have the clusters, which is just that for

loop at that point, the inertia for

this fitted model. Then we can also save

the model as well, just the full-on model if we

wanted to access that later. So I'm going to run this and this again and take

just a second to run. Again, I will pause the video and we'll come back

when it's done running. Okay. That has now

run and we now have our k-means list of our different clusters

and their inertia, as well as their models. That list, if we think

about our pandas series, recall that that's

going to be each one of our difference indices

for that series. So we're going to

concatenate each of those series together

using axis equals one. Then we're going

to transpose it so that our different column names

are going to be clusters, inertia and model and

we'll have that for each one of our

different clusters and their respective inertias for each of those

different cluster values. We're then only going to

take clusters and inertia. So once we have those

as our columns, we're only selecting

those two columns, we're setting our

index to clusters. Those are going to be

that number of clusters. That will allow us to

easily call plot data, which is now our pandas

DataFrame that we have created here.plot and say that we want a line connected by each one of markers

connected by lines, markers being Os here. Then we just want our x ticks

to go from zero to 21 or 20 and then our x

limit go from 0- 20. We run this and with our x

labels being cluster and our y labels being inertia and we try to see if there's

any shrunk elbow. It doesn't seem like there's

quite that may be a bit of that at four perhaps, where it starts to decline, how much it's going to be

really declining as quickly. So maybe you choose four, but probably the best practice is if you know that there's

some type of clustering here as we did with either

the quality of the wine and we knew that they were

6,7 values there from 3-9, or if you knew that

there's red or white wine, you choose that as

one of your case. Now that closes out our

discussion here with k-means. In the next question, and in the next video, we are going to start

to discuss using agglomerative clustering to create our different clusters. All right. I'll see you there.

Hi and welcome back for question number 5 of

our notebook here. In this question

we're going to fit in agglomerative clustering

model with just two clusters. We are then going to go ahead

and compare the results of agglomerative clustering

to that of K-means, and then also compare that

against a red and white wines, and see if the numbers and the groupings seemed

to be the same. We'll then also going to

visualize the dendrogram. The dendrogram is going to be that subgroups building up to those larger groups

that we saw during lecture when talked about

agglomerative clustering. We're going to see how we can

do that in Python as well. The first thing

that we want to do is import agglomerative

clustering. We're going to create our object here and

pass in the arguments. We're going to say that

we want two clusters or specifying the number of

clusters is equal to two. If we wanted, we can also pass in the distance threshold

as an argument here. If we do do that, if you want to do

that back at home, you just have to make sure to set the number of clusters

equal to none. You have to do number of clusters or the

distance threshold, you cannot do both. Here we'll do number of clusters. We want it to compute

the full tree. If you did a certain amount get cutoff to save computation time. If you want to save

computation time, you could set this to false. But this will run fairly

quickly and it'll allow us to see everything

that built up within our tree. Then we're setting

our linkage toward. Again, that linkage means that we're finding

the clusters that reduced the inertia the most

between any other groupings. Once that has been initiated, we're then going to fit

it to our data just using the float columns

as we did before. We're going to add that in as another column

within our data set. So we had K-means before, and now we're also going to

have the agglom data set. I run this and this will

take just a second to run, but not too long. Now we have our data and we can then use the same

method that we saw before. We're going to take our data and take the subset

of the colors, the agglom column that we just created and the K-means

that we created earlier. First, we're going

to group by color and agglom and see the counts, so we'll run this. We see, again, that the

red and white wines, we're able to group

them appropriately. We're able to see that for red, only 31 were of agglom class 0, and 1568 were of agglom class 1, whereas the majority of white was classified as agglom

class 0 here. We have the zeros and ones, very highly related, very highly correlated with our

red and white wine. That was a similar story when we worked with K-means as well. Now, the numbers are a bit flips. Doesn't really

matter whether zero or one, that's arbitrary, but the fact that they

are separating them out into specific classes. We see 1576 versus 23, 1568 versus 31, maybe

not quite as well there. Again, agglom maybe

didn't do quite as well for the white wine either, but still did a good job of classifying each of these

two separate classes. Then if we want to look at

both of these in total, this will be a little

bit difficult to read given that trade off between zero and one and also just having this multi index, I would suggest just

looking at these top two that we just discussed. But if you want to dive

deeper and see for red wine when we had agglom, how much of the K-means were in agreement, and

these would be agreement, both one and zero, 1563, and you can break it

down accordingly, and take a deeper dive into where the mismatches

may have happened. Again, though the clusters

are not identical, the clusters are very

consistent within a single wine variety,

either red or white. Now we're going to plot

out our dendrogram. I don't want to walk through all the different pieces of code. This is just for plotting

out dendrograms. That's all you'll really need in order to use this moving forward. But your fitted model should have these children which will help us identify the breakdown

of our model. We use this hierarchy.linkage that we imported

from scipy.cluster, which will allow us, again, to create what we need to pass into our dendrogram that

we're going to use. We're going to initiate

our figure and our axes, we're going to create the

colors that we want to use, and set the link color palette. So how are we going to

link each of these? You'll see this red and gray come into play in just a

second once we plot it out. Then we call hierarchy, which is what we imported here, dot dendrogram to plot

out our dendrogram. Now, Z is equal to that hierarchy

linkage object that we created just here above. Some important arguments. First, let me run this

so we can see what this looks like before going

through the arguments. We see the dendrogram and we see how it broke down

from side to side. We also see this one down a

certain amount of levels. The same go all the way

down to the bottom. If we wanted to see all the

way down to the bottom, we can change that and would

take some more time to plot. But we see also the number we said show leaf

counts equals True. We can see the number that shows up in each one of these

different subgroups. How many rows showed up in

each of these subgroups? Now, if we wanted

to see less data, we could set this p equal

to something like 10. I run this again. Now, you only see if you count the bottom lines

that we have here, there's only 10 lines. So it's breaking it down,

so you could see up until they're only

10 subgroups left. That's dependent

on using the last p. You can also write here level, and you can say how many

levels down you want to go. Just to highlight, this

is about two levels down. If we were to run this

just one level down, we can see that just breaks

out into these two subgroups. Again, I changed the p and the trunking mode

at the same time, in order to see how much of that dendrogram we actually

want to visualize. Now, we're going to stop here. In the next video, we're going to discuss how

you can actually incorporate these different clusters into creating your different models, seeing the performance of each, and then closing out this video with another

walking through of the performance with

different levels of different types of clusters.

I'll see you there.

Now in this question, we're going to

explore the idea of using clustering as a form

of feature engineering. The first thing that we

need to do is create a variable that we're

going to try and predict, as when we're doing our

feature engineering, this will now be for

supervised learning. We are going to create a

binary target variable y, which is just going

to denote whether or not the quality is

greater than 7. Greater than 7 will

be equal to 1, 7 or less will be equal to 0. We're then going to

create a variable called x with k-means, and that's going to be

from our original data, so it's going to be

a Pandas DataFrame, and we're going to take that data and everything that

we've worked with so far. If you recall, we added on agglom as a column as well as

k-means as a column. We'll drop quality,

color, and agglom, which will leave that k-means, so we have all of our float columns plus

that k-means column. Then we're going to create

another Pandas DataFrame, which is x without k-means, hence that's just taking

what we just created from x with k-means and

dropping k-means column. Then for both datasets, we will use

StratifiedShuffleSplit with 10 different splits. We will fit 10 different Random Forest Classifiers,

and with that, compute that ROC AUC score

of these 10 classifiers, find the average of each and

see which performed better. The one with k-means or

the one without k-means. In order to do so,

we're going to have to first import our Random

Forest Classifier. We will also import our ROC AUC score as well as

our StratifiedShuffleSplit. Hopefully you'll recall

all of that from the course when we did

supervised learning. We're then going to create

our target variable, which is just when the

quality is greater than 7, we said that equal to 1. If we say just this part, data quality greater than 7, that will return either

true or false setting it, astype int, converts that

true to 1 and the false to 0. We then initiate our

objects x with k-means, which is just going

to be our dataset that we currently

have worked towards, but dropping agglom,

color and quality. We still have k-means

in our float columns, and then x without k-means, we'll take that x with

k-means that we just defined and drop

the k-means column. Now we have these two

different Pandas DataFrame, one is just the float columns, which is x without

k-means and one is the float columns with that

k-means column as well, which is x with k-means. We're then going to initiate our StratifiedShuffleSplit

object, and then we're going to define this function which will allow us to pass in an estimator

and that estimator, a spoiler alert here will be

Random Forest Classifier, but we'll see how

we'll use this again for logistic regression as well, then an x and a y. Our different features and

then our outcome variable. First we initiate an

empty list of ROC AUC, and that's because if you recall, we're going to create

10 different values and then take the mean

of each of those values, so we'll append to each of those values to this empty list. We take train index

and tests index, four values in our

sss.split for x and y depending on the x and y that we passed in here

within our function. Because this sss is defined

to have 10 different splits, when we run this for-loop, we are running through

10 different iterations of different

StratifiedShuffleSplits. Different splits of

our data that have ensured that there's

a stratification that's a certain amount of

data quality greater than 7 shows up in each one of our different train

and test sets. Then we said x train and x test, using those train indices

and test indices and we set y train and y test with those train indices

and test indices. We can then call that estimator

that we defined up here, that we're passing into our

function and call,.fit, on our training set

that we defined. Then we can come up with

our actual prediction, which is going to be

estimator.predict on our test set, on our holdout set. We can do the same for our

predicted probabilities. If you recall, if we

want that ROC AUC score, then we need the

predicted probabilities to actually create that. We get the probabilities, that's going to output the probabilities for

both of the classes. We only want the positive class, so we're taking all rows, but only the first column, not the 0 with column. That's going to be our

different scores and then we can call for each one of

our different iterations, the ROC AUC score for

our actual values, that's the y test, as well as the scored values

that we just computed. We will continuously append that to our empty

list so that we get all 10 different ROC values. We then take the mean of

that list and we will have the average for the

different ROC scores across those 10 different splits. Now that we have that

function defined, that will output that

average across the 10list, we could set our estimator here to RandomForestClassifier. We have estimator

equal to this object. We pass that in to our

function that we just defined, along with

X\_with\_Kmeans. This is with the

column of Kmeans, and that's going to be our X as well as our target column y. Then we're going to

do the same thing running that function to get, the same estimator except

on X\_without\_Kmeans. With our dataset without

that extra column. We run this. We can see that

without\_Kmeans cluster actually did worse than

with\_Kmeans cluster. We performed better when we had our Kmeans cluster as input

into our RandomForest. Now, what I'd like

to do is explore the idea of changing

the number of labels that we will incorporate when we create this

new feature or this now new set of features if we think about this in

regards to one-hot encoding. We're going to say, for

n equals 1,..., 20, we fit a KMeans algorithm

with n clusters. So first, one cluster, two clusters, three

clusters, so on. We then have to one-hot encode

it, because otherwise 19, label number 19

will be thought of as greater than label number

5 or label number 10. Instead we want those

each one-hot encoded, so that there's no ordinal value to each one of those

different values. And once we have our one-hot encoded

version of that column, we're then going to fit a

Logistic Regression model and compute that

average roc-auc-score. Then we're going to plot

that average roc-auc-score for each one of our different

numbers of clusters. So I'm going to run this while explaining because it may

take a little bit of time. But the way that we start off, is that we're going to set X\_basis equal to just

those float columns. We're then going to initiate our StratifiedShuffleSplit

with only 10 splits as we did before. We're then going to define this new function,

create\_kmeans\_columns. As I mentioned, we

can't just create that one column with

multiple labels, we have to one-hot

encode those labels. We say km equals Kmeans with a number of clusters equal

to whatever n we pass in. We're then going to fit on

just our float columns. Then when we call km.predict

on our X\_basis here, we're actually outputting each one of these different labels. If the number of n

was equal to 20, we'd have values 1, 2, 3, 4, all the way through 19, actually starting

from zero up until 19 to have our 20

different clusters. We then take that column

that we just created and we call pd.get\_dummies

on that column. Now we create, if there is

19, 19 different columns. Having a one or a zero, if that column

happened to be a one, a two, a three, so

on and so forth. We then concatenate,

just those flow columns to those new kmeans

columns that we defined. So that maybe up to 20

columns that we're adding on. Then once we have this dataframe, the idea is that we will

be able to pass that in as our dataframe and

then fit our models. Initiate our estimator

as LogisticRegression. We say the ns, the

number of clusters that we want to run through

are 1 through 20. We're then going to get

our list of roc and auc values by calling that get\_avg\_ roc\_10splits that we defined just above,

in the cell above. We pass into that the estimator. Our X value is going to be

this create\_kmeans outputs. Remember this output

will actually output a pandas dataframe, that's going to concatenate onto that original data float columns, our new labels, one-hot encoded. Then using that same

target variable for each n in our different ns

that we have defined up here. We're then going

to plot that out. We initialize our plot and then we just plot the ns versus the different roc aucs that

are output given the model, given the function that

we're running here. We've already ran this. Let's look down at the results. We see it jumps

around quite a bit as we add on and reduce

some of those clusters. That closes out. This is just after

over 10 iterations. That closes out our section here on the different

clustering methods. Gives you an introduction

to how you can also use these different

clustering methods to actually do some

feature engineering. With that, we close out

our section on clustering and in lecture we will move on to Dimensionality Reduction. All right, I'll see you there.

# Summary

## Distance Metrics

Clustering methods rely very heavily on our definition of distance. Our choice of Distance Metric will be extremely important when discussing our clustering algorithms and to clustering success.

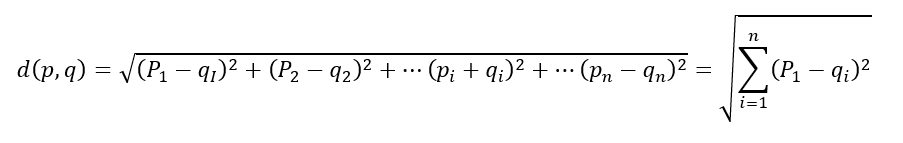
Each metric has strengths and most appropriate use cases, but sometimes choosing a distance metric is also based on empirical evaluation to determine which metric works best to achieve our goals.

These are the most common distance metrics:

**Euclidean Distance**

This one is the most intuitive distance metric, and that we use in K-means, another name for this is the L2 distance. You probably remember from your trigonometry classes.

We calculate (d) by taking the square root of the square of each of this changes (values). We can move this to higher dimensions for example 3 dimensions, 4 dimensions etc. In general, for an n-dimensional space, the distance is:



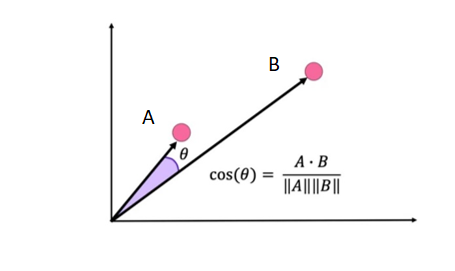
**Manhattan Distance (L1 or City Block)**

Another distance metric is the L1 distance or the Manhattan distance, and instead of squaring each term we are adding up the absolute value of each term. It will always be larger than the L2 distance, unless they lie on the same axis. We use this in business cases where there is very high dimensionality.

As high dimensionality often leads to difficulty in distinguishing distances between one point and the other, the L1 score does better than the L2 score in distinguishing these different distances once we move into a higher dimensional space.

**Cosine Distance**

This is a bit less intuitive distance metric. What we really care about the Cosine Distance is the angle between 2 points, for example, for two given points A and B:



This metric gives us the cosine of the angle between the two vectors defined from the origin to two given points in a two-dimensional space. To translate this definition into higher dimensions, we take the dot product of the vectors and divide it by the norm of each point.

The key to the Cosine distance is that it will remain insensitive to the scaling with respect to the origin, which means that we can move some of the points along the same line and the distance will remain the same. So, any two points on that same array, passing through the origin will have a distance of zero from one another.

Euclidean VS Cosine distances

- Euclidean distance is useful for coordinate based measurements.

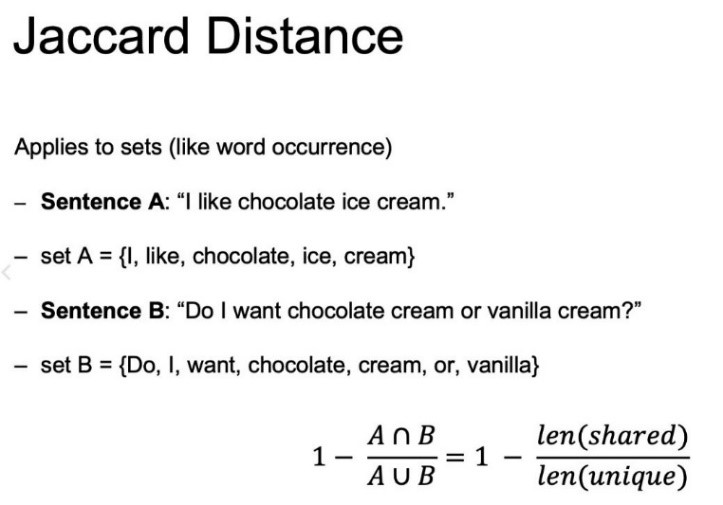
- Euclidean distance is more sensitive to curse of dimensionality

- Cosine is better for data such as text where location of occurrence is less important.

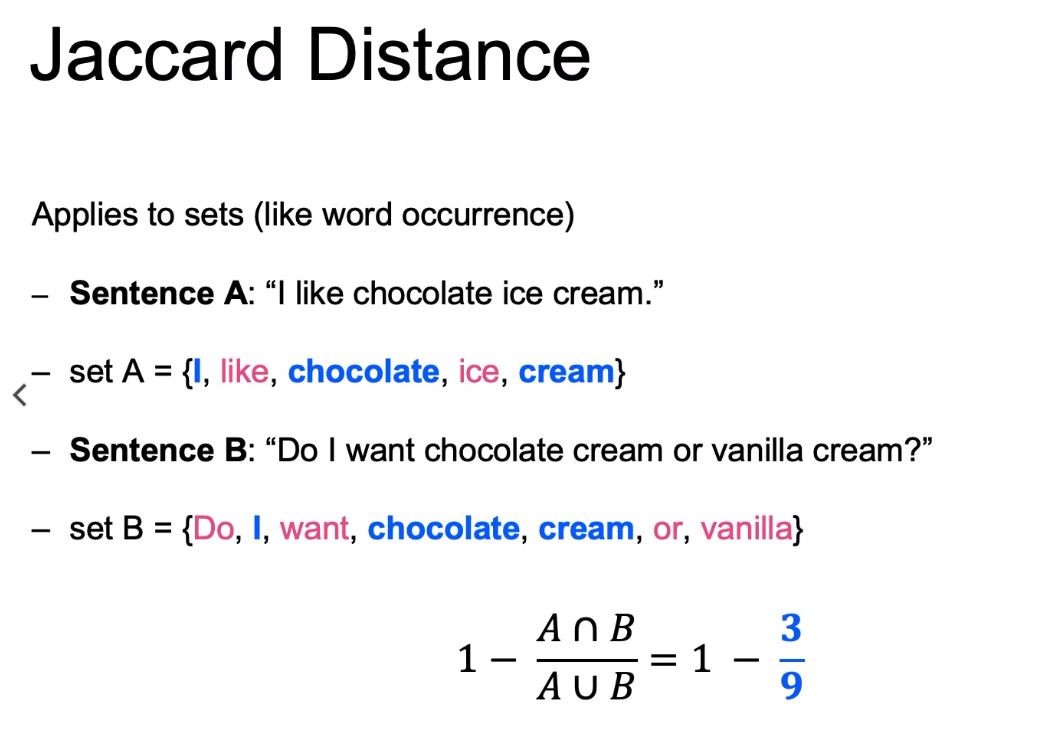
**Jaccard Distance**

This distance is useful for texts and is often used to word occurrence.

Consider the following example:



In this case, the Jaccard Distance is going to be one minus the amount of value shared. So, the intersection over that union. This intersection means, the shared values of the two sentences over the length of the total unique values between sentecnes A and B.



It can be useful in cases you have text documents and you want to group similar topics together.

## Hierarchical Clustering

This clustering algorithm, will try to continuously split out and merge new clusters successively until it reaches a level of convergence.

This algorithm identifies first the pair of points which has the minimal distance and it turns it into the first cluster, then the second pair of points with the second minimal distance will form the second cluster, and so on. As the algorithm continues doing this with all the pairs of closest points, we can turn our points into just one cluster, which is why HAC also needs a stopping criterion.

There are a few linkage types or methods to measure the distance between clusters. these are the most common:

**Single linkage:** minimum pairwise distance between clusters.

It takes the distance between specific points and declare that as the distance between 2 clusters and then it tries to find for all these pairwise linkages which one is the minimum and then we will combine those together as we move up to a higher hierarchy.

Pros: It helps ensuring a clear separation between clusters.

Cons: It won’t be able to separate out cleanly if there is some noise between 2 different clusters.

**Complete linkage:** maximum pairwise distance between clusters.

Instead of taking the minimum distance given the points within each cluster, it will take the maximum value. Then from those maximum distances it decides which one is the smallest and then we can move up that hierarchy.

Pro: It would do a much better job of separating out the clusters if there’s a bit of noise or overlapping points of two different clusters.

Cons: Tends to break apart a larger existing cluster depending on where that maximum distance of those different points may end up lying

**Average linkage:** Average pairwise distance between clusters.

Takes the average of all the points for a given cluster and use those averages or clusters centroids to determine the distance between the different clusters.

Pros: The same as the single and complete linkage.

Cons: It also tends to break apart a larger existing cluster.

Ward linkage: Cluster merge is based on inertia.

Computes the inertia for all pairs of points and picks the pair that will ultimately minimizes the value of inertia.

The pros and cons are the same as the average linkage.

### **Syntax for Agglomerative Clusters**

First, import AgglomarativeClustering

**From sklearn.cluster import AgglomerativeClustering**

then create an instance of class,

**agg = AgglomerativeClustering (n\_clusters=3, affinity=‘euclidean’, linkage=‘ward’)**

and finally, fit the instance on the data and then predict clusters for new data

agg=agg.fit(X1)

y\_predict=agg.predict(X2)