Lecture 6

For deliverable, create a folder with an improved project proposal based on discussions with the client. Schedule a meeting with the client, go over proposal with them, clarify and questions, try to identify any limitations/risks with data or acheiving project goals.

Fork the repo, create a pull request for our project.

Hierarchical Clustering

Last time we looked at Lloyd's clustering, which was a partitioning type of clustering. Today we'll look at a different type of clustering, called hierarchical clustering.

There are two main types:

- Agglomerative: this is one way of building the hierarchy as follows:
  - start with every point in its own cluster.
  - At each step, merge the two closest clusters.
  - Stop when every point is in the same cluster
- Divisive:
  - Start with each point in the same cluster
  - At each step, split until every point is in its own cluster

We are mostly going to focus on the agglomerative methods, since the divisive methods are more computationally expensive.

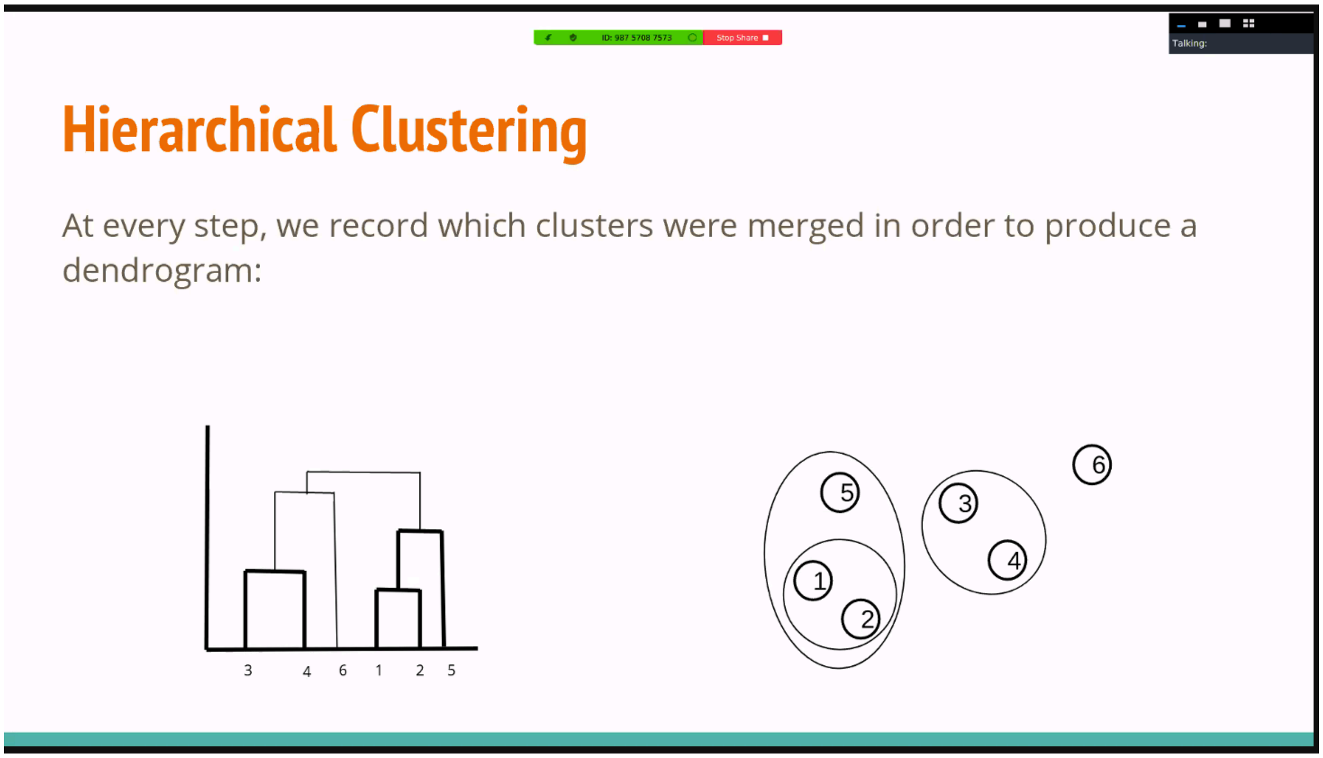
Agglormerative Clustering Algorithm

```
1. Let each point in the dataset be in its own cluster

Repeat:
2. Compute the distance between all pairs of clusters
3. Merge the two closest clusters
until: all points are in the same clusters
```

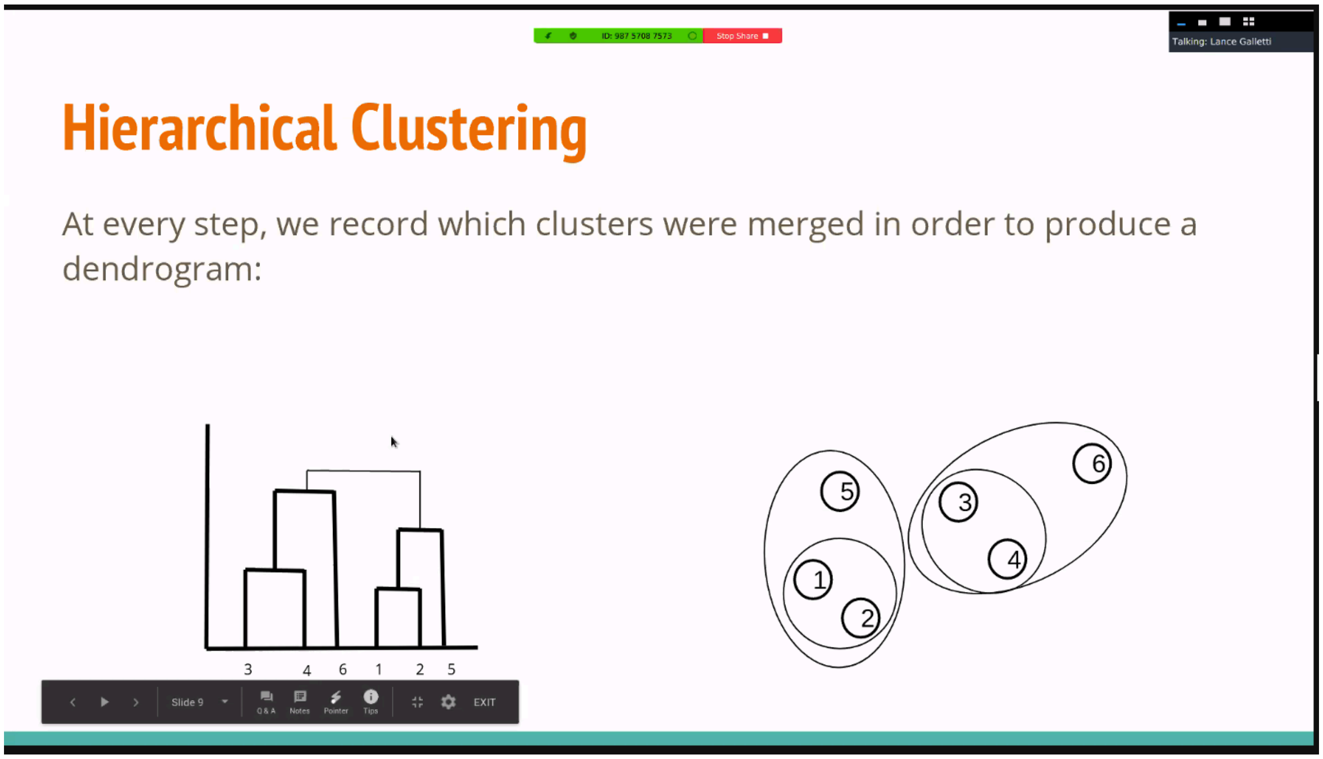
Question: after merging, how do we compute the distance between clusters that each have multiple points? We'll return to this.

At every step, we record which clusters were merged in order to produce a dendrogram:



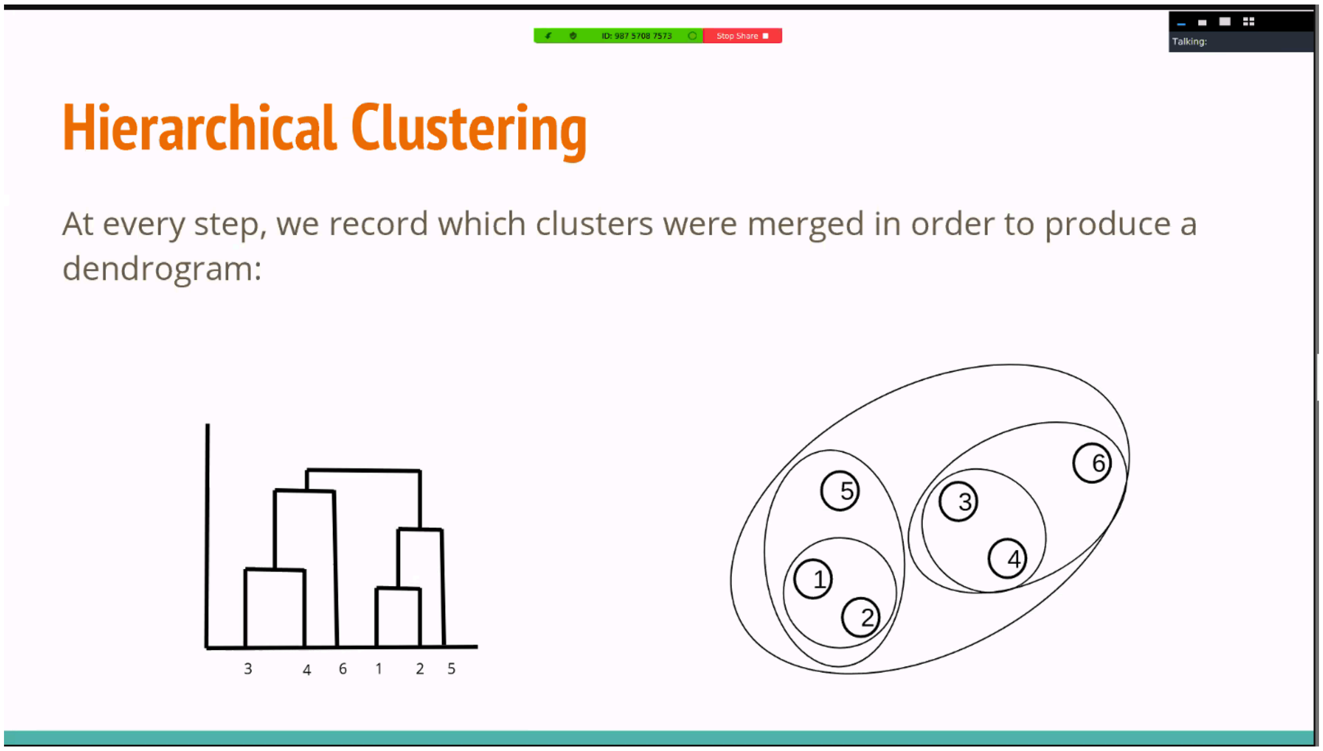
Hierarchical Clustering

At every step, we record which clusters were merged in order to produce a dendrogram:



Hierarchical Clustering

At every step, we record which clusters were merged in order to produce a dendrogram:



Hierarchical Clustering

At every step, we record which clusters were merged in order to produce a dendrogram:

Note the progression of merged clusters in the slides above.

What kind of applications can we see with this kind of graphical presentation of the clusters? One possibility is with global trends in clustering. Another example would be in species similarity - how much DNA do various species share? We can then set thresholds at differing similarities, and we can then look at the different clusters between these thresholds. That might be one way then of separating between two different species.

Can we compute this? Are we missing anything? Well, we need a way of computing distance between different clusters. Here, to compute this, we can think of the distance between clusters as the distance between two sets of points.

Some definitions:

- $d(p_1, p_2)$ denotes the distance between points
- $D(C_1, C_2)$ denotes the distance between clusters

Single Link Distance

This is the minimum of all pairwise distances between a point from one cluster and a point from another cluster.

$$D_{SL}(C_1, C_2) = min\{d(p_1, p_2) | p_1 \in C_1, p_2 \in C_2\} \qquad (1)$$

Basically, generate a set of all possible pairwise distances, and then choose the minimum value. Note that the distance between points here can be various different distance functions - Euclidean, Manhattan, etc. For simplicity, let's assume we're talking Euclidean distance.

What this is good at is that it can hanlde clusters of different sizes - kmeans wasn't very good at this, since it tried to find clusters of about the same size. However, it is quite sensitive to noisy points. It also tends to create elongated clusters.

Complete Link Distance

This is the maximum of all pairwise distances between a point from one cluster and a point from the other cluster. This is pretty much the same as above, except that we take the max instead of the min.

$$D_{CL}(C_1, C_2) = max\{d(p_1, p_2) | p_1 \in C_1, p_2 \in C_2\} \tag{2}$$

What's good about this is that this is less susceptible to noise points. Further, it creates a more balanced, equal diameter clusters. However, it is bad at handling outliers, and it also tends to split up large clusters, as each cluster tends to have the same diameter.

Average Link Distance

The average of all pairwise distances between a point from one cluster and a point from the other cluster.

$$D_{AL}(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{p_1 \in C_1, p_2 \in C_2} d(p_1, p_2) \tag{3}$$

Sum all pairwise distances, and then divide by the number of distances that we have computed.

This is less susceptible to noise and outliers, but it tends to be biased towards globular clusters, which basically means that it behaves like kmeans.

Centroid Distance

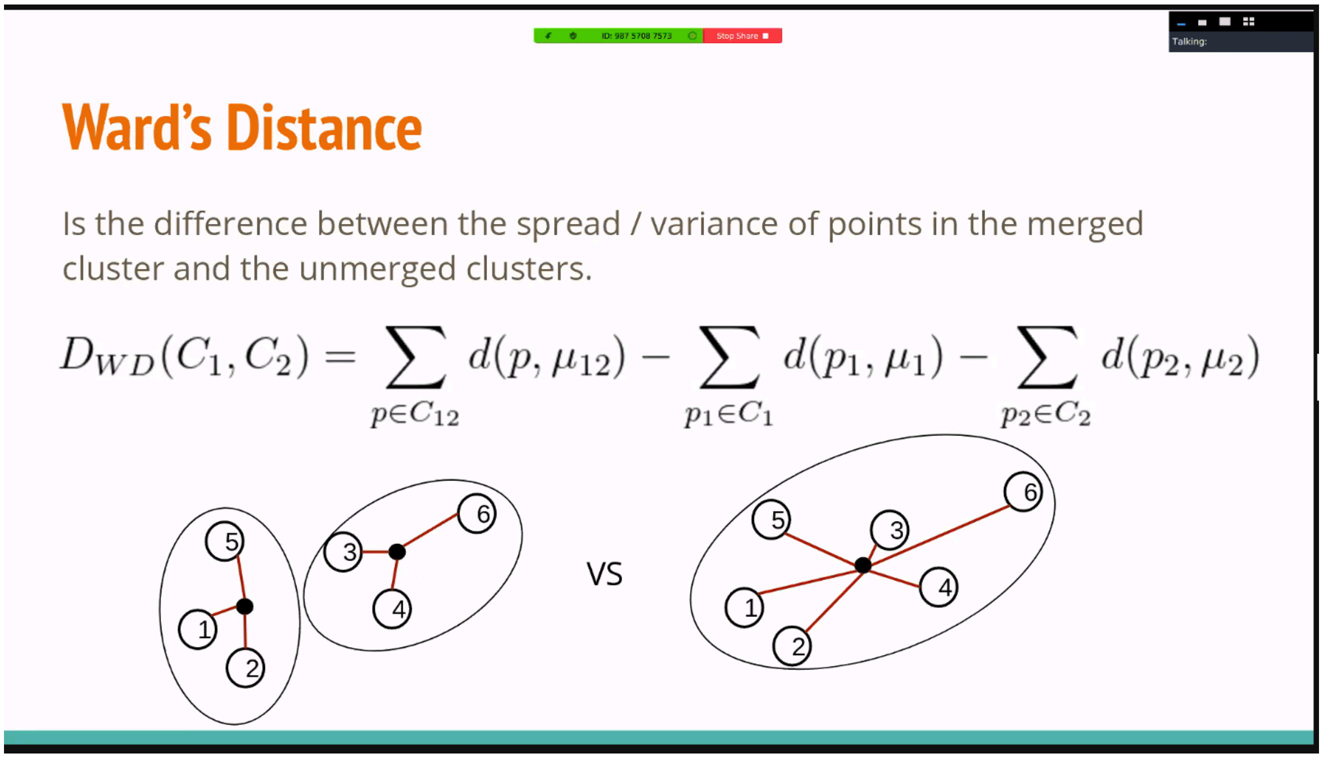
The distance between the centroids of clusters

$$D_C(C_1, C_2) = d(\mu_1, \mu_2) \tag{4}$$

Ward's Distance

"State of the art" distance. This is the difference between the spread/variance of points in the merged cluster and the unmerged clusters. The idea is that we want to know how much do we gain by merging clusters? To understand what would happen if clusters were merged, we can look at the new variance between - if the new variance is high, then the points are rather spread out. If the new variance is low, then the points are close together. Thus, we basically ask, how much can we get the variance to improve by merging clusters?

To calculate this, we need the distance between all points in the merged cluster, minus what we actually had in the beginning.



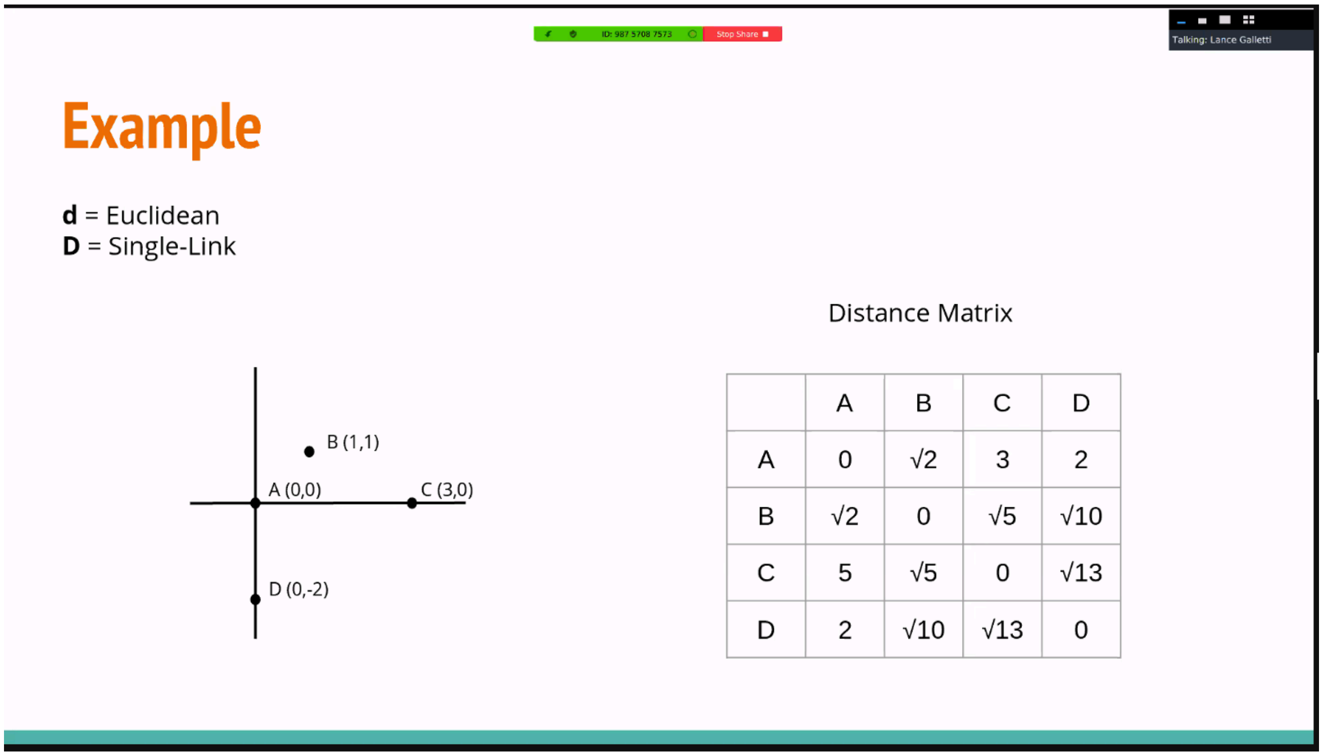
Ward's Distance

Is the difference between the spread / variance of points in the merged cluster and the unmerged clusters.

$$D_{WD}(C_1, C_2) = \sum_{p \in C_{12}} d(p, \mu_{12}) - \sum_{p_1 \in C_1} d(p_1, \mu_1) - \sum_{p_2 \in C_2} d(p_2, \mu_2)$$

Essentially, we want to pick the cluster that picks the smallest variance. Then, we'd merge clusters with the lowest Ward's distance.

Example: we walked through an example using the Euclidean distance and single-link



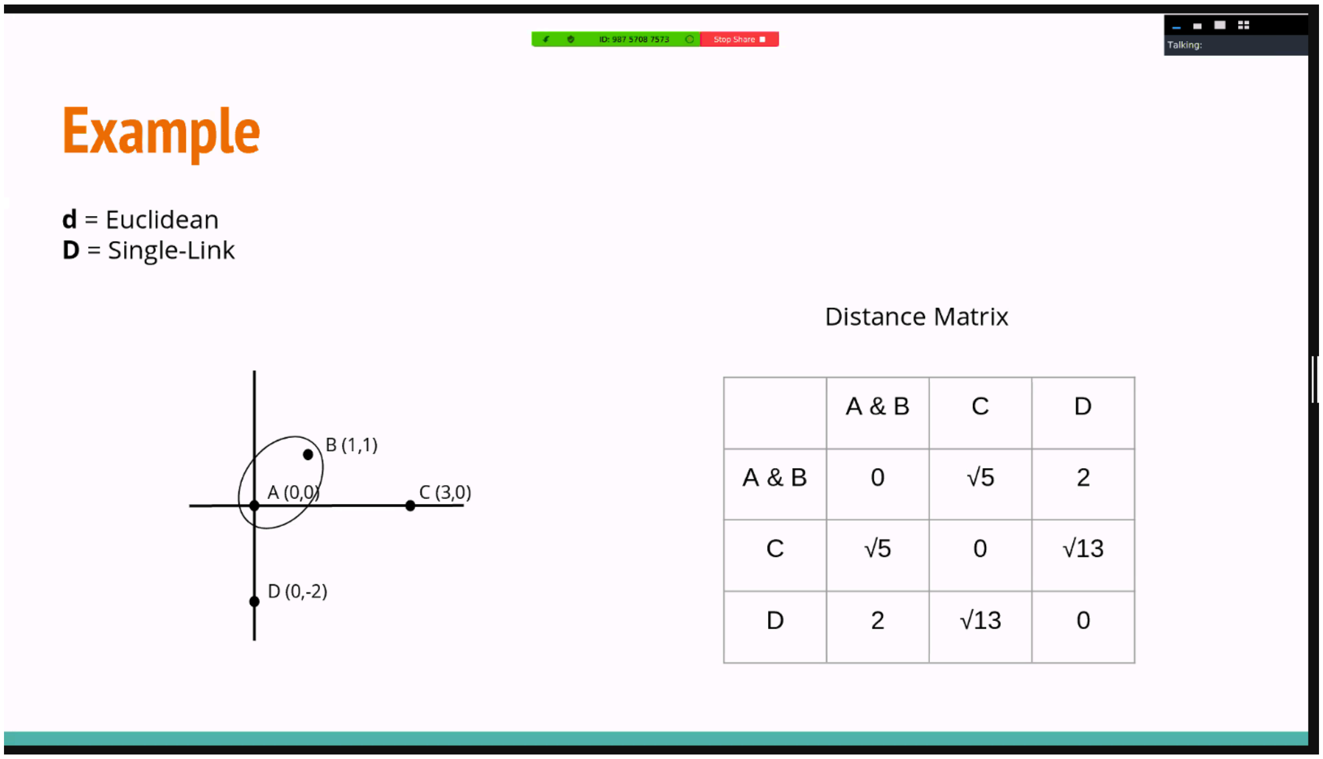
Example

d = Euclidean
D = Single-Link

Distance Matrix

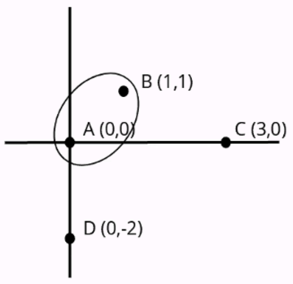
	A	B	C	D
A	0	√2	3	2
B	√2	0	√5	√10
C	5	√5	0	√13
D	2	√10	√13	0

After step one, we merge A and B, because the square root of two is the min. Now, we need to update the distance matrix:



Example

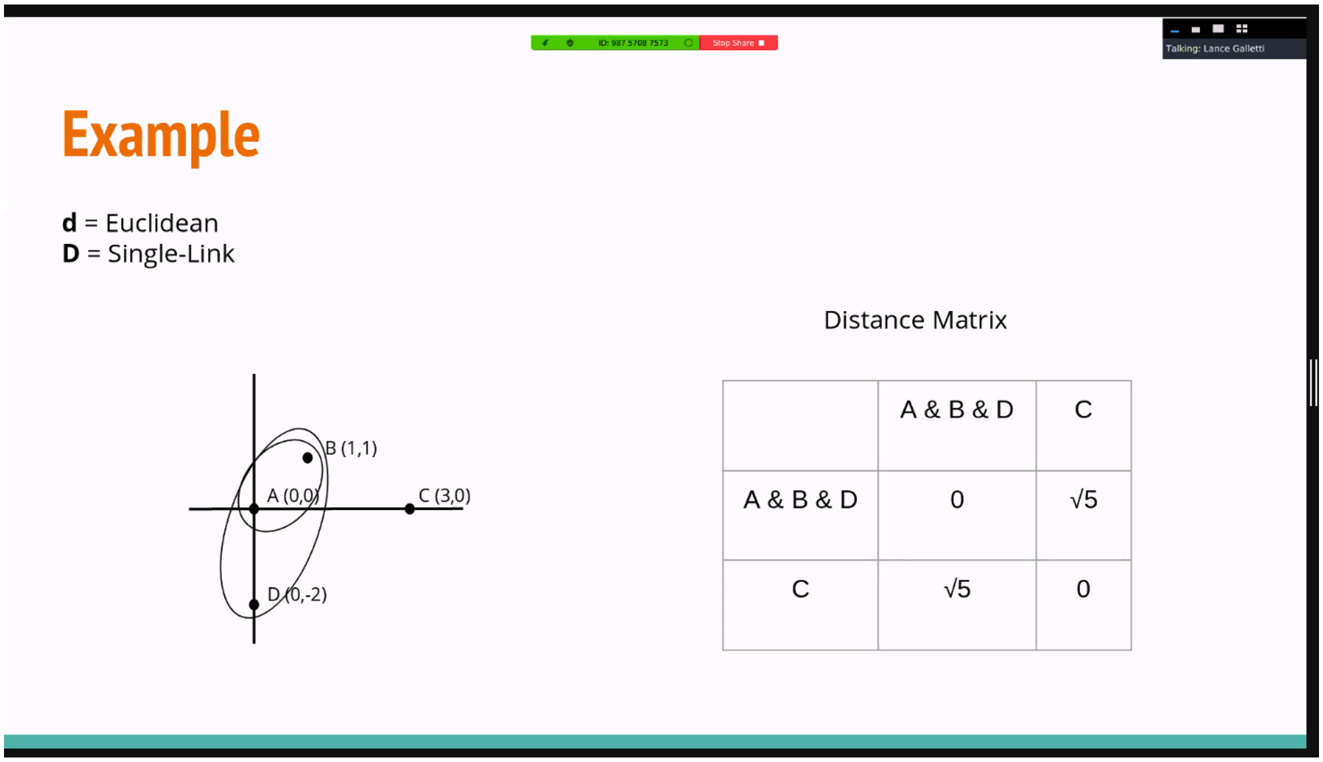
d = Euclidean
D = Single-Link



Distance Matrix

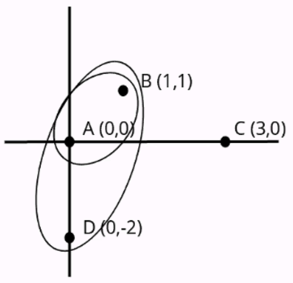
	A & B	C	D
A & B	0	√5	2
C	√5	0	√13
D	2	√13	0

At this step, we merge AB with D, since 2 is the min. Repeat one more time



Example

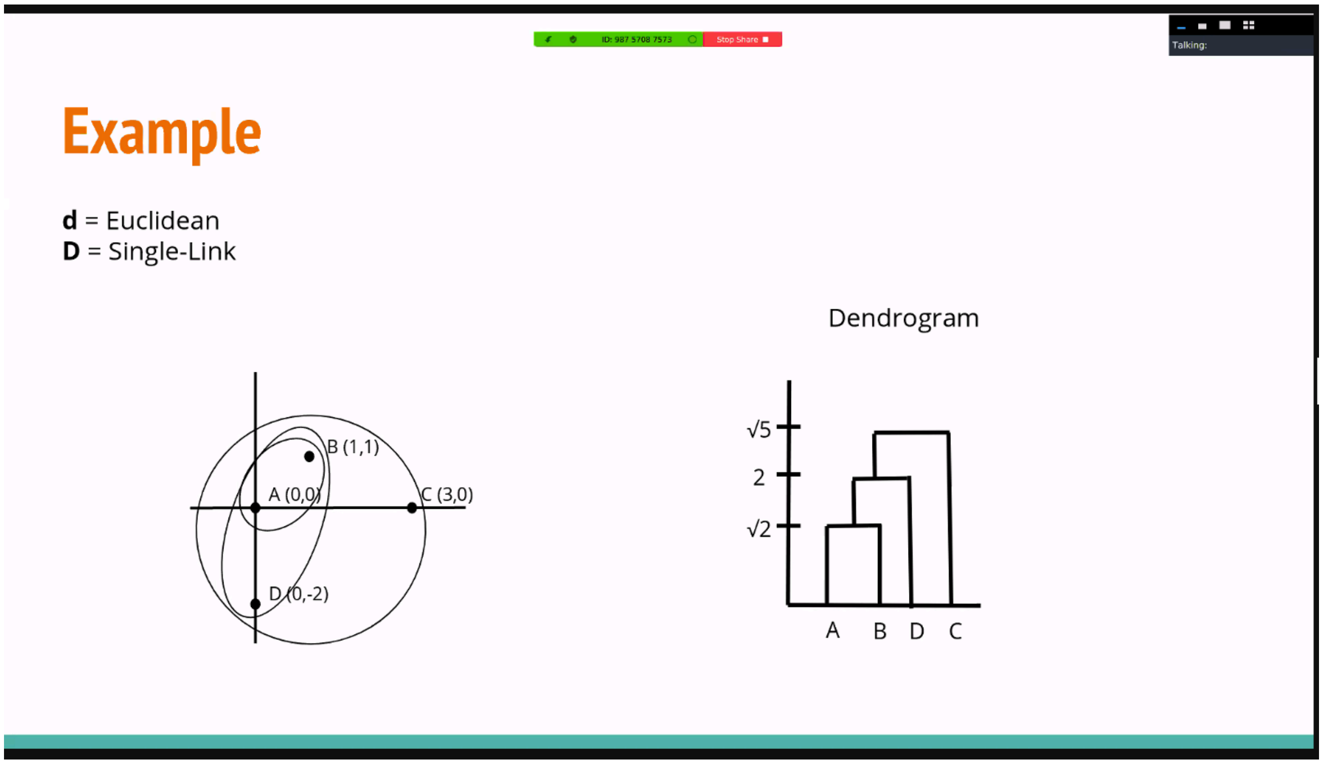
d = Euclidean
D = Single-Link



Distance Matrix

	A & B & D	C
A & B & D	0	√5
C	√5	0

And finally, we can generate the full dendrogram with the distances at which we merged:



Example

d = Euclidean
D = Single-Link

Some Remarks

Finding the threshold with which to cut the dendrogram requires exploration and tuning. But in general, hierarchical clustering is used to expose a hierarchy in the data (ex: finding/defining species via DNA similarity)

To capture the difference between clusterings you can use a cost function, or methods that we will discuss later when we look at clustering aggregation.

Density-Based Clustering

Here, the goal is cluster together points that are densely packed together. However, we need to define density somehow.

Given a fixed radius ϵ around a point, if there are at least **min_pts** number of points in that area, then this section is dense. We'll refer to this ball of ϵ radius as the epsilon neighborhood. Note that one point may not define a dense area, but could be in the epsilon neighborhood of a different point. Thus, we want some differentiation between the core of a dense region vs. being at the edge of a dense region.

- Core point: if its epsilon beighborhood contains at least min_pts.
- border point: if it is in the epsilon neighborhood of a core point.
- Noise point: anythine else - that is, any point that is not a core point nor a border point.

What's great about this, is that we can essentially label the entire set as being core, border, or noise. From there, it's simple enough to create clusters by connecting core points that are both within the same epsilon neighborhood. Note then that noise points don't actually get assigned to clusters!

Question: how do we decide epsilon and min_pts? Well, these are kind of tuning parameters - we need to tweak them to get a sense of what works well for our data.

DBScan Algorithm

1. Find the epsilon neighborhood of each point
2. Label the point as core if it contains at least min_pts
3. Label points in its neighborhood that are not core as border
4. Label points as noise if they are neither core nor border
5. For each core point, assign to the same cluster all core points in its neighborhood.
6. Assign border points to nearby clusters

Benefits of DBScan

- Can identify clusters of different shapes and sizes
- Resistant to noise

Limitations: however, it tends to prefer clusters of similar densities - it won't do so well with clusters of different densities.

Live Coding of DBScan Algorithm

Did some live coding of this algorithm - code is in the repo, not taking notes here.