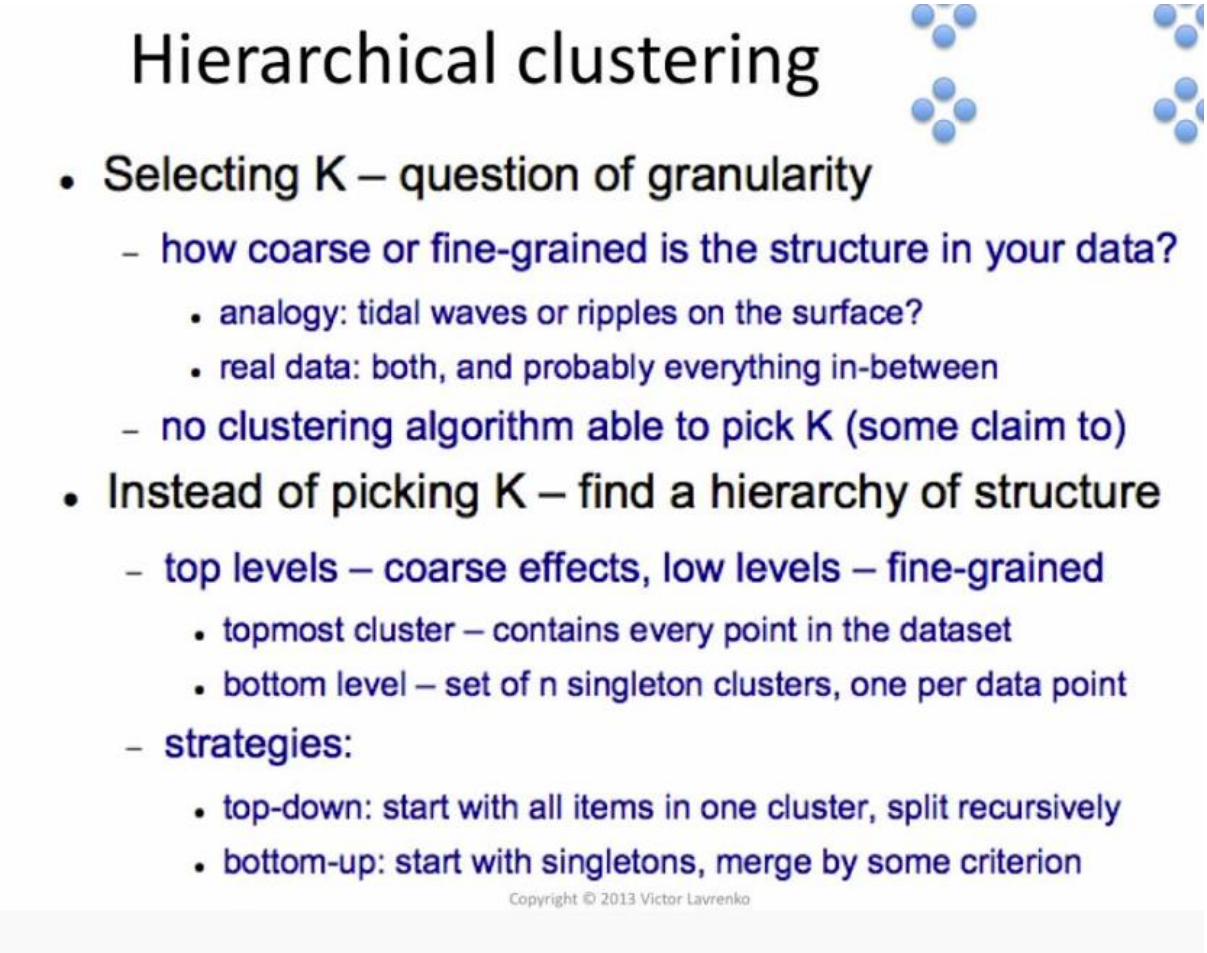
# Hierarchical Clustering




No universally good way to pick number of clusters. The number of clusters within a dataset depends on the context and what you plan to do with it. This is highly ambiguous. So, you are dealing with a real dataset, you are dealing with data that has structure at multiple scales. When you are asking how many clusters there are, you are basically asking the question *what is the right scale* to be looking at the data. Ground scale, fine scale, finer scale, etc. So. It's a question of scale – do you want to model the big effects in your data or do you want to look at the fine-grained things. Hence, there is no good answer for picking the number of clusters within a given (real-world) dataset.

**The idea behind *hierarchical clustering* is that instead of choosing *k* (number of clusters), construct a hierarchy**. So, at the top, you have no cluster and then you are going to have two clusters and so on. **You build a hierarchy of data points and the levels of hierarchy near the top will model (sort of) the coarse-grained effects in your data. And the levels of hierarchy near the bottom, you will see the fine-grained effects in your data.**

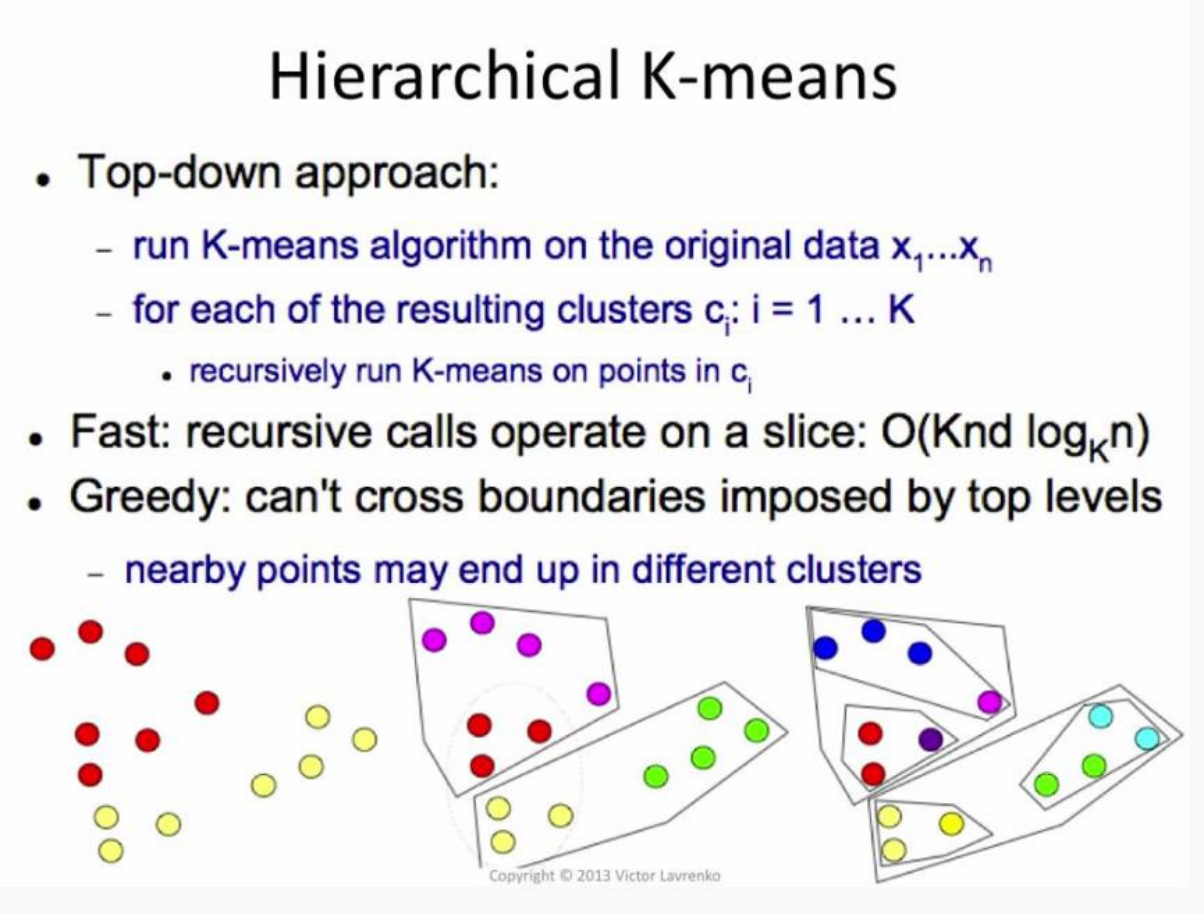
Again, the top cluster will contain all of the data points in the given dataset, and at the bottom level, you would have mini/singleton clusters having one instance/data point each in them. So, that is the idea behind hierarchical clustering.

There are two ways to arrive at a hierarchy of clusters. There are more, but here, we only talk about the two ways. **Strategies of constructing a hierarchy of clusters:**

1. **Top-down:** start with a single cluster containing all of the data points in it and then split it recursively using some strategy.
2. **Bottom-up:** start with as many clusters as the data points in the dataset (a singleton cluster for each individual instance in your population) and then you start merging the nearest clusters according to some strategy. You keep merging them until you merge every data point/cluster(s) into a single cluster.

Doing so creates a structure.

Let's take a brief look into both of these strategies. For *top-down* hierarchical clustering approaches, there are many ways to do that but the simplest way is to use *hierarchical K-Means*.




Say k = 2 (a predefined value) and for each of the resulting clusters, apply K-Means on them recursively. This will produce a hierarchy.

The main advantage is that this is *relatively* fast. At each level of hierarchy or tree, we are going to be creating *k* clusters out of *n* points and each point has d-dimensionality. At the top level, we have *n* elements in a single population. 'd' complexity is just a measure of complexity for comparing a centroid to each data point, that is going to take you 'd' operations because you have 'd' attributes in your data. And 'k' because you have to compare each data point to each one of the 'k' clusters. And the number of iterations is left out, that is constant.

This happens at the top. What is going to happen at the second level? At the second level, suppose that the clusters are approximately equal in size, so one cluster has n/2 items in it and the other cluster has n/2 items. So, you have one run of K-Means over n/2 and another run of K-Means over n/2, so that is (k x (n/2) x d) + (k x (n/2) x d), and over all it's going to work out to be the same (k x n x d).

So, at each level of the hierarchy, you have exactly the same number of computations. And it doesn't matter how the instances are distributed into the cluster, whether the clusters are uniformly sized or whether one cluster is bigger than another, because what you are doing is that you are putting a hard boundary. Once this instance is in this sub-population, you never compare it to any of the clusters from the bottom sub-population. It's a divide-and-conquer type thing.

The complexity is (kxnxd) at each level of the hierarchy and the number of levels you can have is $\log_k n$. If the clusters are roughly balanced, you cannot have more than $\log_k n$ levels in the tree.

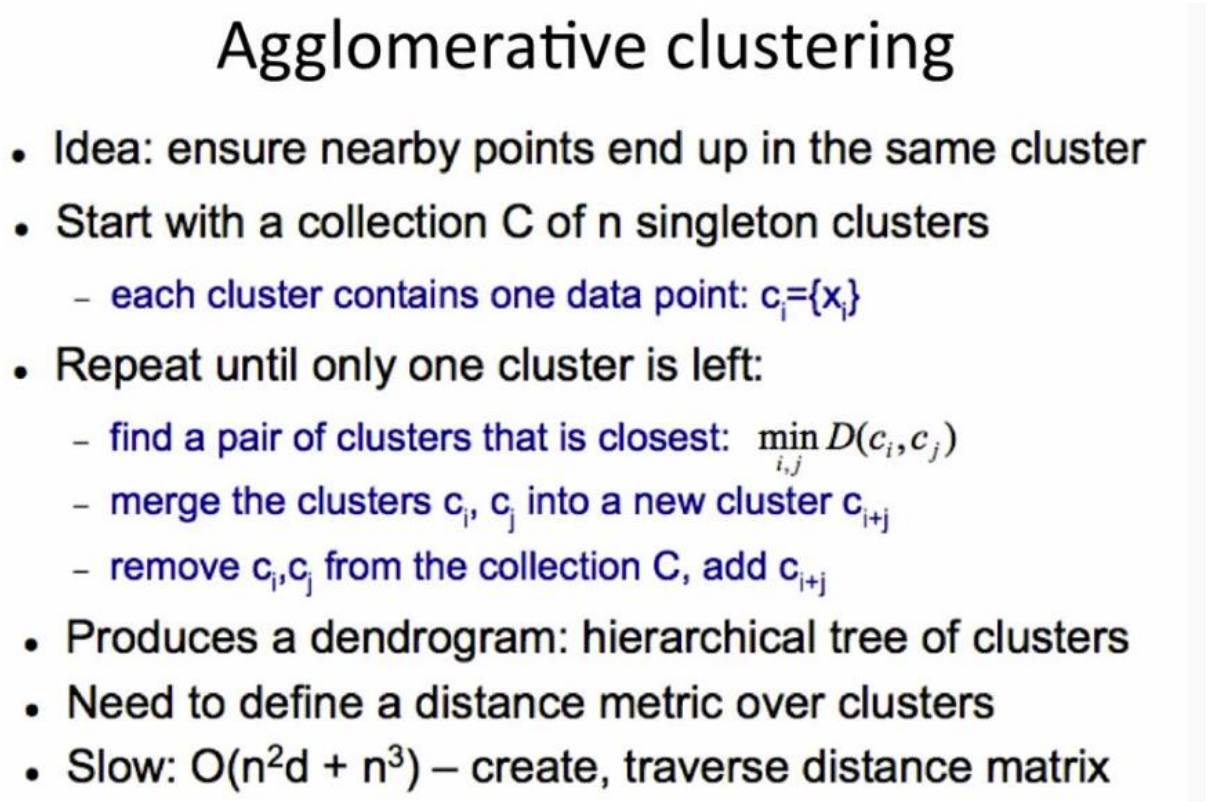
If k = 2, each time you are splitting each cluster into 2 sub-clusters and in $\log_2 n$ steps you will end up with singleton clusters. This is with the assumption that the clusters are roughly equally balanced, but this is a decent bound on performance. And this is not much more expensive than the original K-Means algorithm. Basically, you have the cost for the original K-Means (Knd) times the log of the number of instances 'n' or, $\log_k n$. log is a conservative function and so, this is reasonably fast.

**The downside/con for *hierarchical K-Means* algorithm are:**

1. **this is *greedy***. Remember that one of the things that we did not like about K-Means algorithm is that you can have (say) 2 nearby points that might end up in totally different clusters. And this cannot overcome that. Once these 2 points are in two different clusters, you will never put them into 1 cluster again.
2. And as in the original K-Means algorithm, **depending on where you start, you will end up with a different clustering and in this case, you will end up with a different hierarchy of clusters**. So, **your initial position for the seeds/centroids for the clusters will have a big effect on what you end up with**.

Reference

# Agglomerative Clustering: How it Works




Here, the basic idea is that if you have two nearby points, they end up in the same cluster which is the idea of clustering. Agglomerative clustering is typically bottoms up.

You start with a collection of singleton clusters where each cluster contains a single data point. Then, you iteratively do the following procedure: look at your collection of clusters and find a pair that is *closest* to each other. This is not the distance between the two instances, this is not the distance between two points. It's the distance between two clusters. We will talk about how to define it and it turns out to be a complicated thing.

So, say you have clusters 'i' and 'j', you then measure the distance between them and then you look over all of the pairs, find the pair of clusters that are closest. If $c_i$ and $c_j$ are closest to each other, you merge them which in turn creates a new cluster $c_{i+j}$ and then delete the two clusters $c_i$ and $c_j$ from the collection 'C' and add $c_{i+j}$ to the collection C. You keep repeating these steps until you end up with only one cluster.

The first thing to do is to measure distances between everything. In the beginning, when you have 'n' singletons, you are going to measure distance between each singleton cluster and all of the other singleton clusters, so that is going to be $n^2$ total comparisons. And each comparison is going to take 'd' operations due to 'd' attributes (for each cluster) in dataset. So that is the cost of creating a distance matrix. And then, it turns out that there is an efficient algorithm for doing everything in terms of that one distance matrix. It turns out that you don't need to recompute distances once you create a new cluster by merging two clusters, you can combine individual distances in a clever way. So, you compute the distance

matrix only once which gives you your $n^2 \times d$ term. And then, you will need to traverse it. So, you will have 'n' steps of this algorithm – which is one of the $n^3$ term. And then, each time we will need to find the nearest pair of clusters and that is going to take $n^2$ steps. You can attempt to optimize it further but in reality, it doesn't drop much below $n^2$ steps.

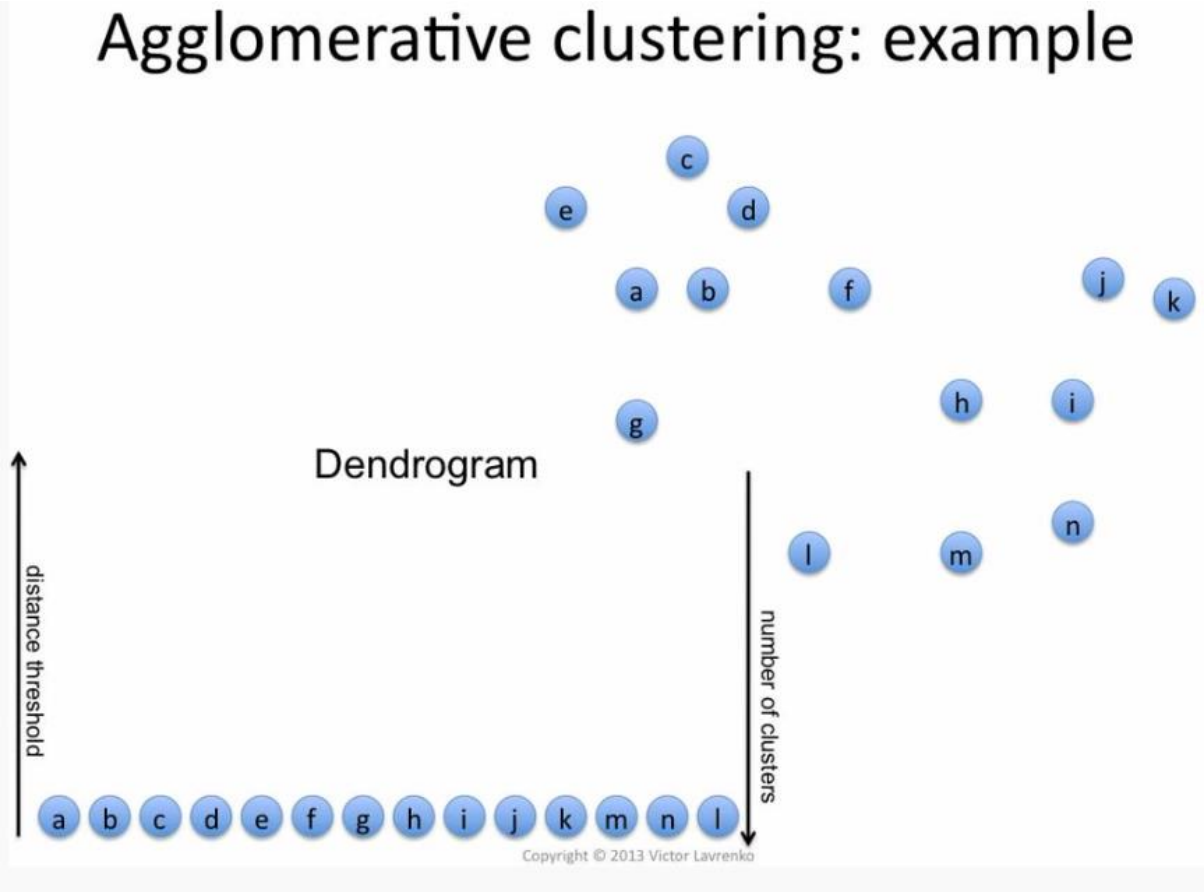
That is the overall complexity: $O(n^2d + n^3)$ and you can see that this is a lot more expensive than the top-down approach of K-Means algorithm.

As an example, consider if you had a billion data points that you wanted to cluster, that is 1 billion cubed! This will run for a long time!

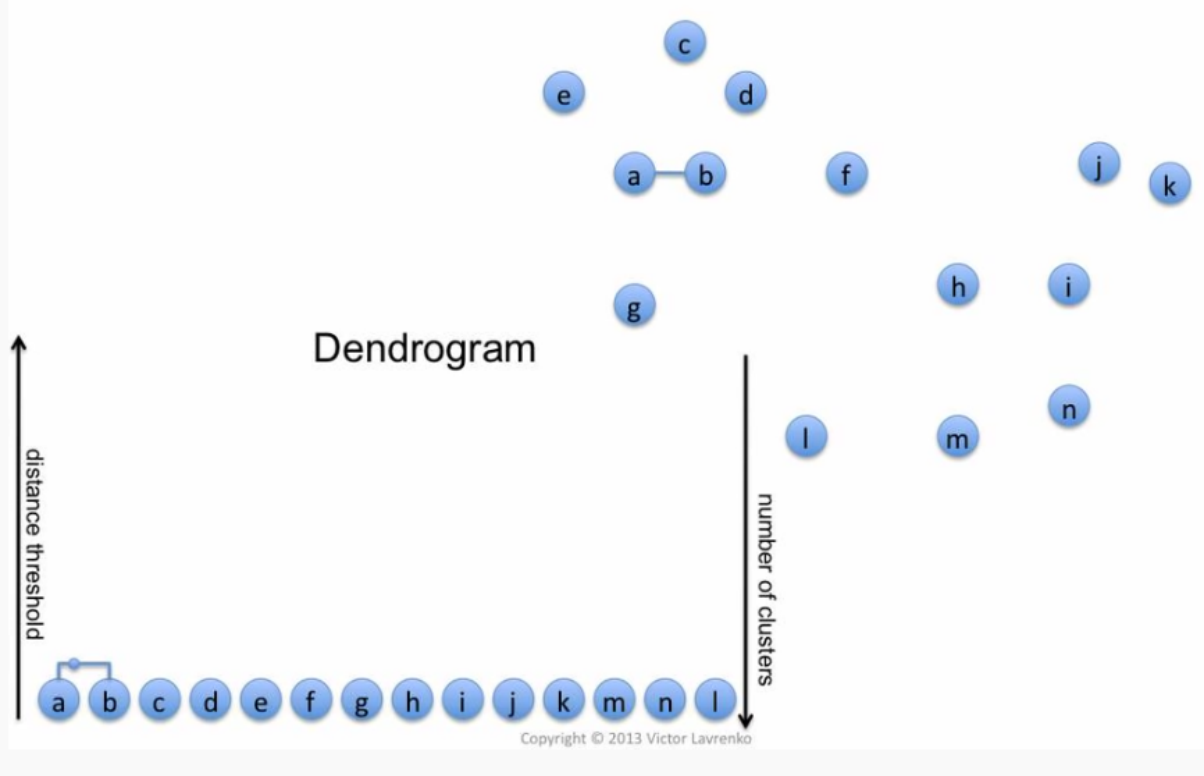
If you do this, you get something called as a ***dendrogram*****. That is a representation of a hierarchical structure**. This basically gives you a tree-like structure of clusters.

You also need to define a *distance metric* for clusters.

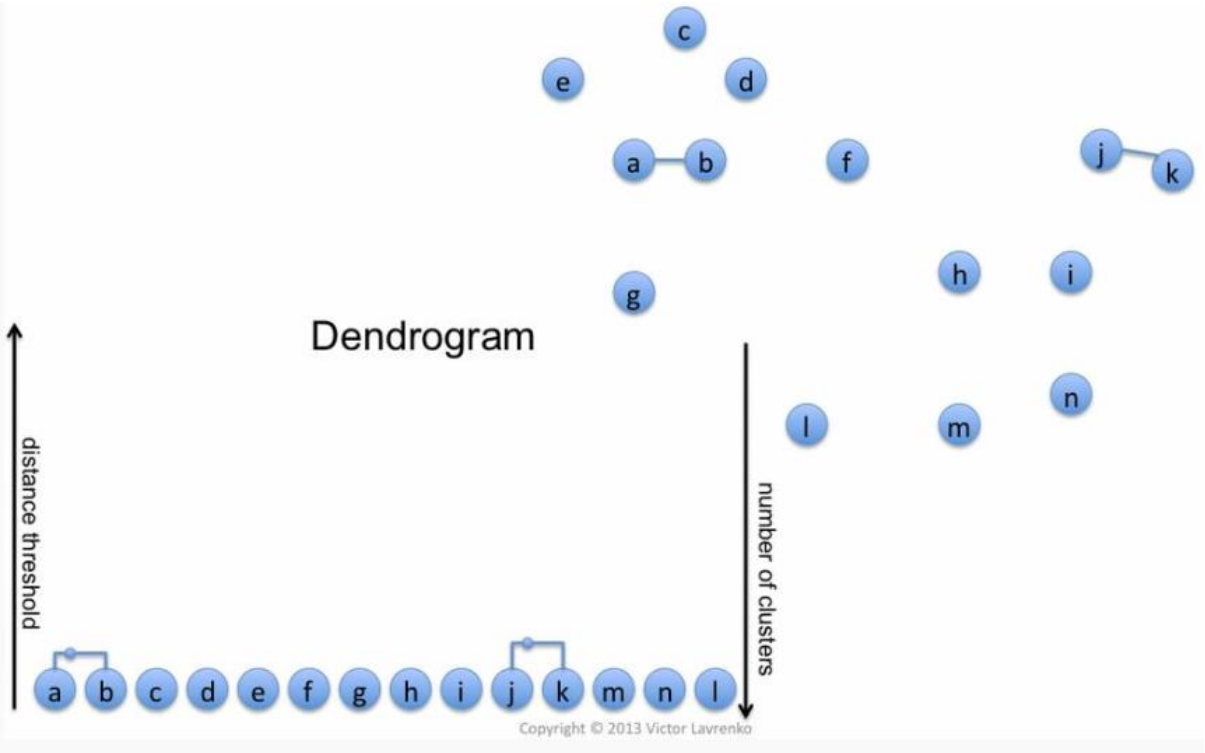
Let's look at one of the simplest examples using **single link clustering algorithm which basically defines the distance between the two clusters as the distance between the two nearest points in those clusters**. The way this works is suppose you have some points in a 2-D space as shown below-



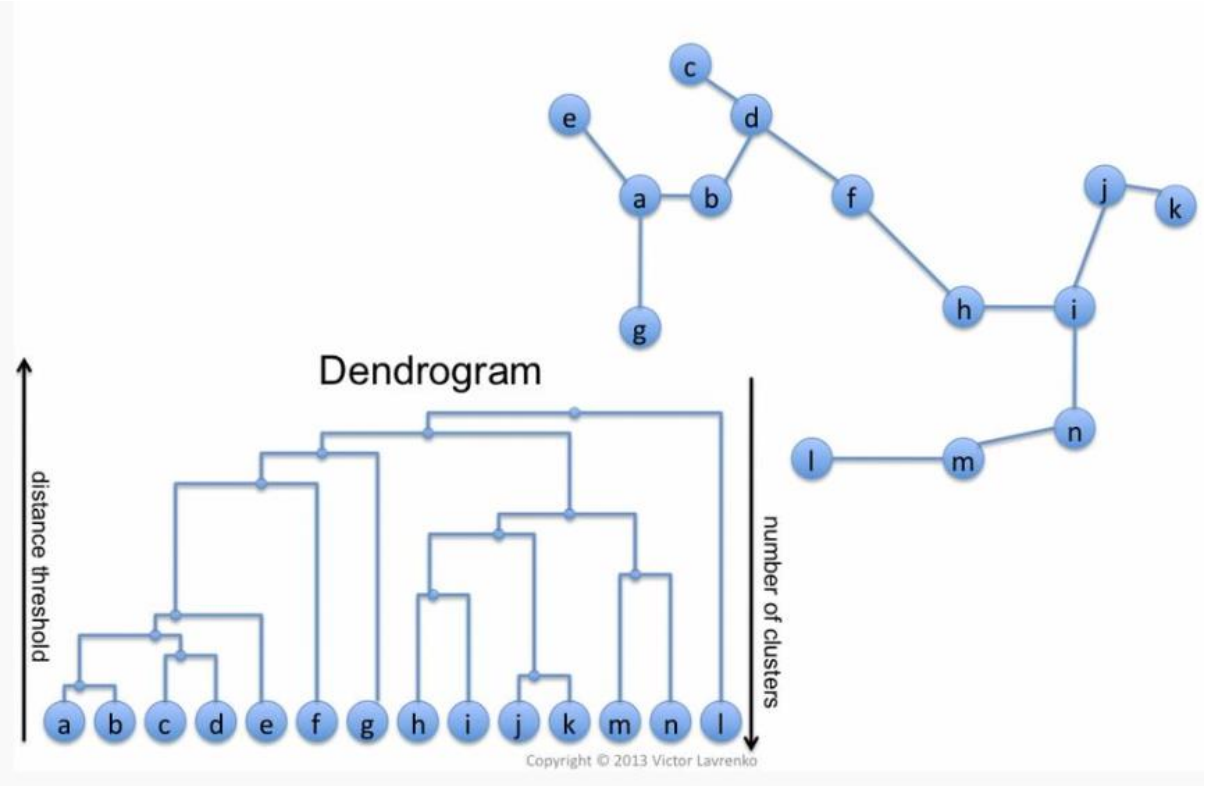
These points are lined-up/arranged in an alphabetical order. How does this algorithm work? Each one of these points is a singleton cluster, so you look for a pair of clusters that is closest and, in this case, say that this pair of closest clusters are ‘a’ & ‘b’. You now put a link between ‘a’ & ‘b’ denoting that this is now a single cluster ‘a+b’ and in the *dendrogram*, you put an edge between nodes ‘a’ & ‘b’. And the *height* at which you put the joining edge corresponds to the distance between ‘a’ & ‘b’ in the attribute space, how far are they from each other.




And then, you look for the next closest pair of clusters, and maybe it’s ‘j’ & ‘k’ and then you put a link between them. And that link in the *dendrogram* will happen slightly higher (as compared to the link between ‘a’ & ‘b’) as the distance is more than the distance between points ‘a’ & ‘b’.



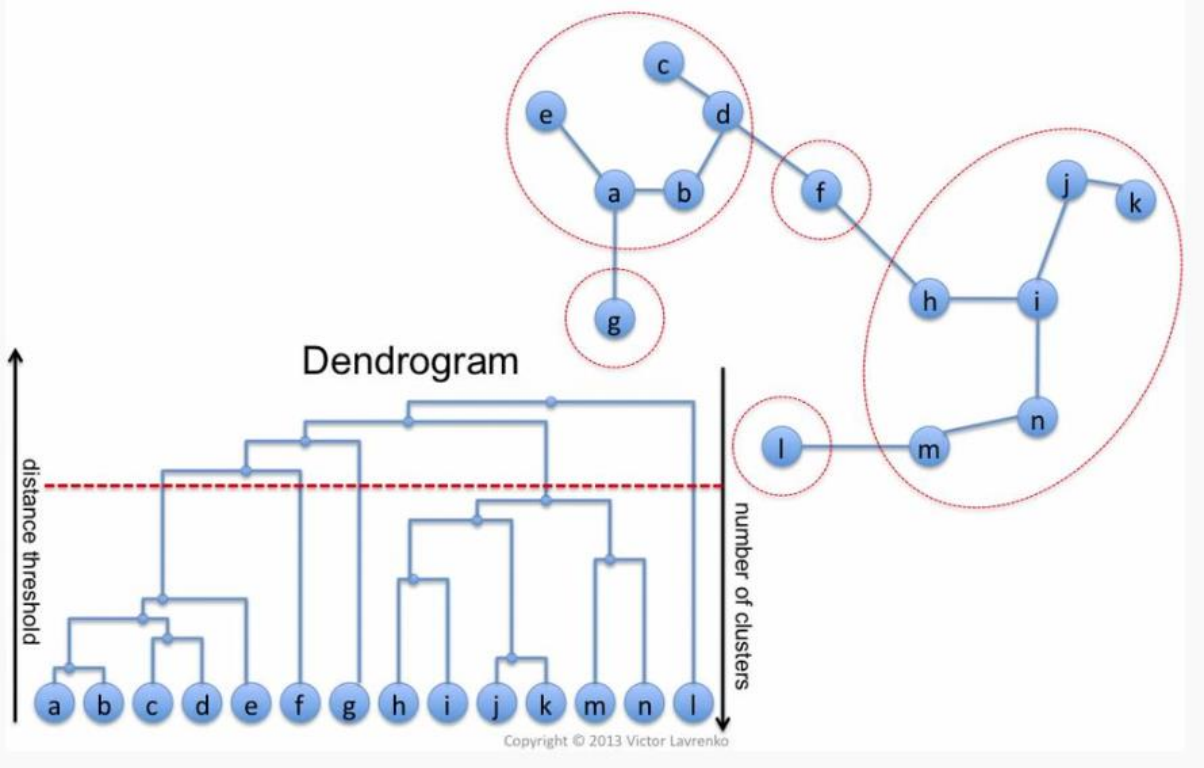

And then, you keep repeating this process- ‘c’ & ‘d’, a cluster merging ‘a’ & ‘b’ with ‘c’ & ‘d’. You keep repeating this. As the distance increases between the clusters, the joining edge’s height increases accordingly. Eventually, everything is connected into one big cluster.




As the merging distance increases, the joining edge is higher and higher.

**The *dendrogram* is the tree which represents the hierarchical structure of the data.** It tells you how the data is organized in space.

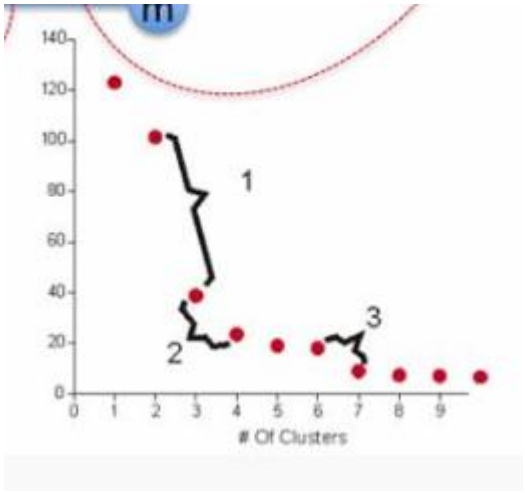
Now, **if you wanted to produce a *flat clustering*, what you do is that you pick a threshold on the distance and you cut the *dendrogram* tree on that distance threshold**.




If in the example above, you decide to cut the *dendrogram* at that distance, every node that comes to this red line becomes a cluster.

Once you have a *dendrogram*, you can actually cut in any place you want and end up with different clusterings. If you got closer to the top, you will get fewer bigger clusters. If you got closer to the bottom, you will get many small clusters.

If you are trying to decide when to stop or, how many clusters you want, then you do the same trick as K-Means. For example, you can plot the distance as a function of the number of clusters and then look for gaps, elbows – maximizing the second derivative, or treat it as a scree plot where you look for the point where the mountains end and the rubble begins. Any of those heuristics are applicable if you really want a fixed number of clusters.

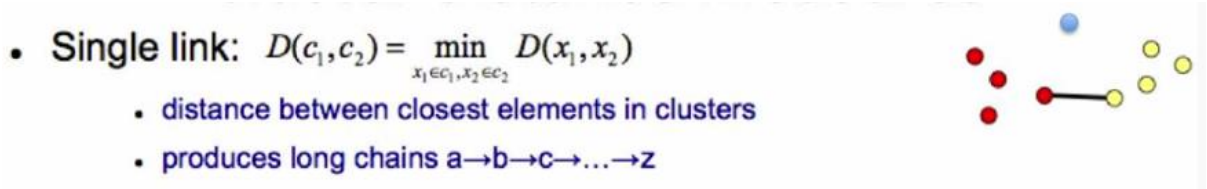



[Reference]

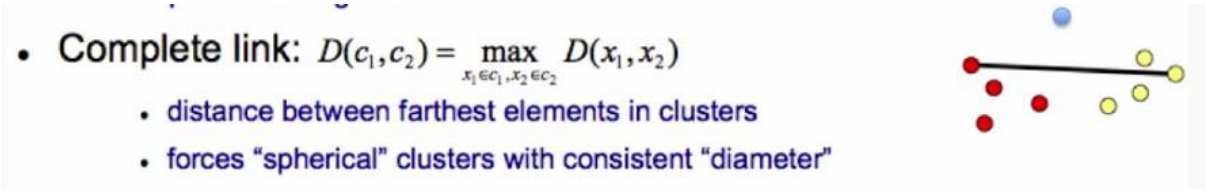
# Hierarchical Clustering: single-link vs. complete-link

## Cluster distance measures

Let's talk about the *distance measures* and **the reason that we are talking about these is that these are the distances between the clusters and not the distances between the individual data points**.




The simple *distance measure* is the **Single link** which we had covered in the previous topic (above). In the example picture, you have 3 clusters denoted by 3 colors – red, yellow and blue. If you measure the distance between the red cluster and yellow cluster, in *single link clustering*, the way you do that is you look for two points such that one point is in the red cluster and the other point is in the yellow cluster and they are as close to each other as they can be. So, **you look for two closest points belonging to two different clusters**. **You are doing a minimum (min) over all of the points in cluster1 and all of the points in cluster2 of the distance between $x_1$ and $x_2$**. This is a nice strategy and, in many domains, this works really well. But one thing that you realize is that **this will produce very long chain because** what you are basically doing is that **you are connecting a point to a nearby point and eventually, you will put two points that are pretty far away from each other into the same cluster.** Whether that is desirable or not depends on the domain on which you are working in. For some domains it might work and for some other domains it might not work.



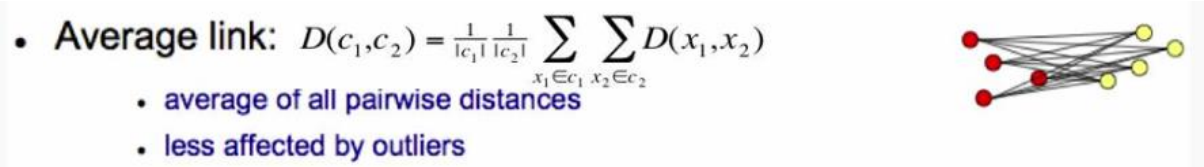

**Complete link** distance measure is the opposite of *single link* distance measure. ***Complete link* tries to produce clusters that are kind of spherical** – this means that it wants all of the points in the cluster to be reasonably close to each other. *Single link* just wants one point in the cluster to be close to another point in another cluster. *Complete link* wants all of the points to be nearby within a certain threshold. And that is why it's spherical. It's defined as

the opposite to *single link*. Here, you have the maximum. For the given example, **when you are measuring the distance between the read and yellow clusters, we are going to look at the pair of points such that one point is in the red cluster and another point is in the yellow cluster and their distance is the largest**. And that is going to be our measure of distance between the red cluster and the yellow cluster. **What this doesn't mean is that it doesn't mean that you are going to merge red and yellow clusters. This is my measure of distance. And once you define this distance, you will have a distance between the red and yellow clusters, a distance between the yellow and blue clusters and a distance between the red and blue clusters and you are going to still pick the *minimum* of those distances!**

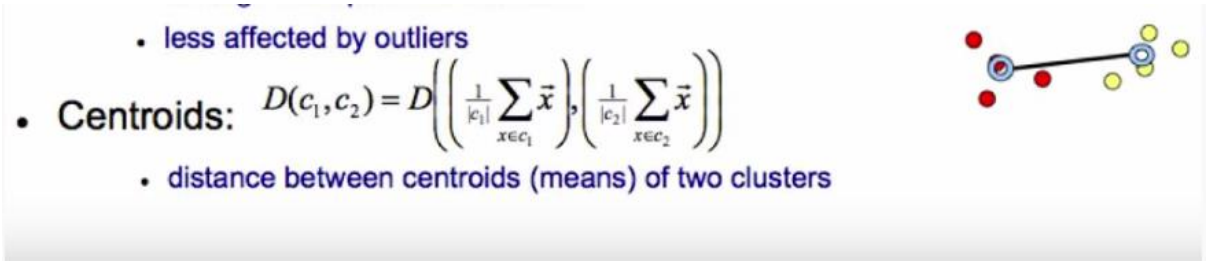
**The distance is now the *max*, but once you compute the distances, you are still looking for the *minimum*, you are still looking for the two closest clusters**.

In the example above, *single link* will first merge red and yellow clusters before merging the blue cluster. Why? Because the distance between the red and yellow clusters is smaller than the distance between red and blue clusters. *Complete link* would not. Why? Because the distance between red and yellow clusters is the distance between their farthest points as they are the most far away points. That is a big distance. Whereas, the distance between the red and blue clusters and the distance between the red and yellow clusters are still comparatively smaller. So, *complete link* will definitely not merge red and yellow clusters first. It will either merge red and blue or yellow and blue clusters.

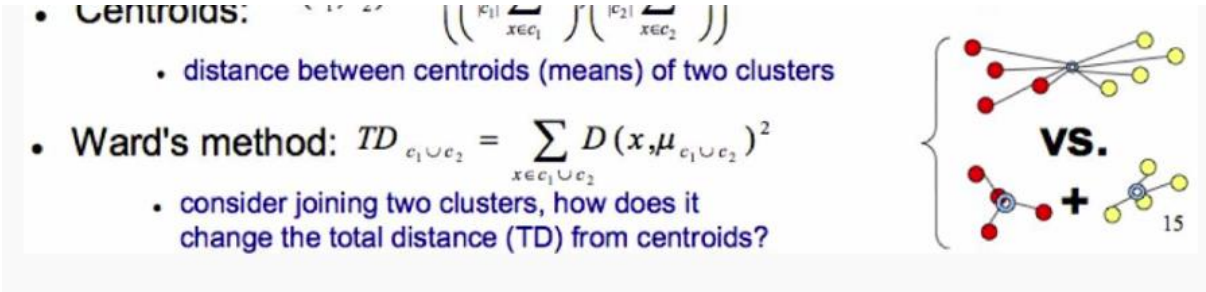
***Single link* looks for chains while *complete link* looks for spherical units**.



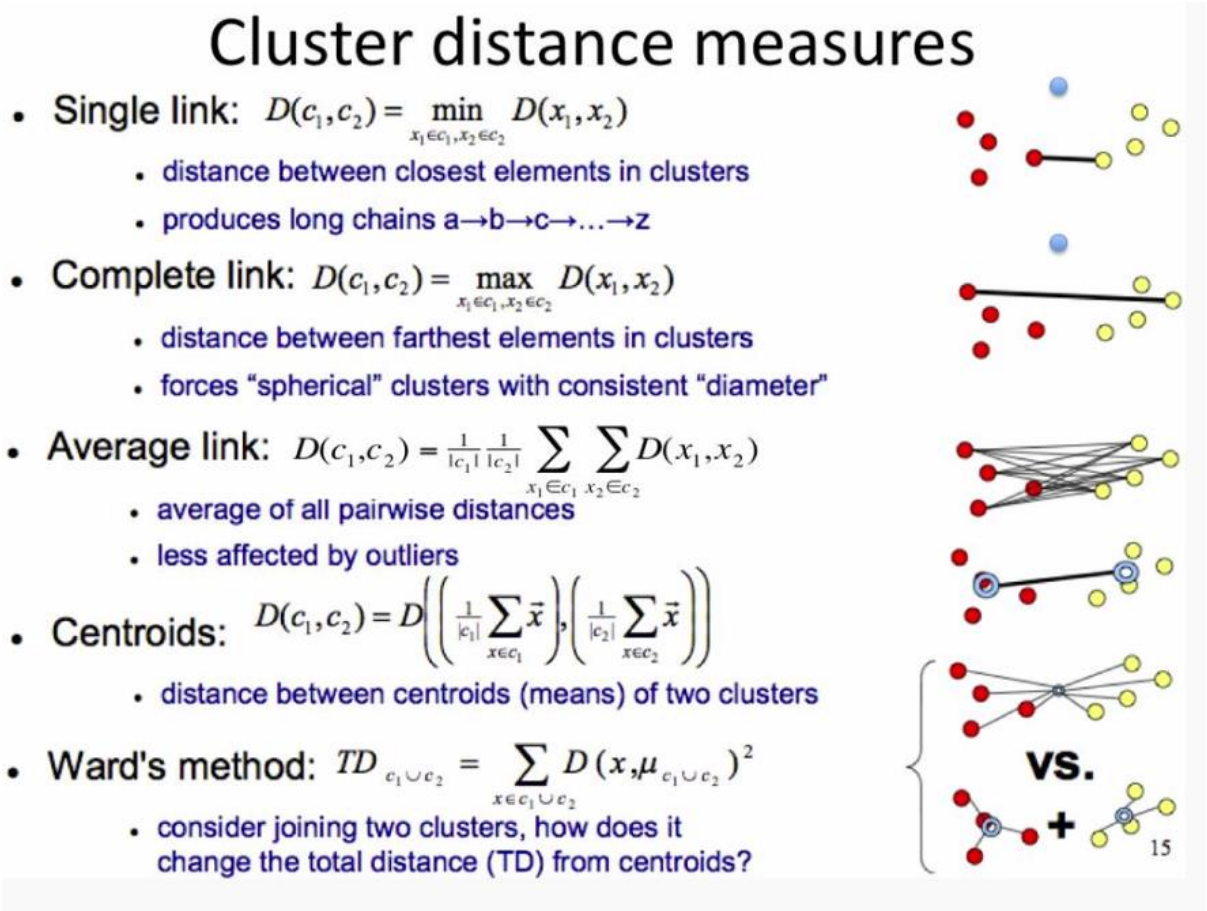

*Average link* looks at the pairwise average of all the distances between elements/data points in a cluster. So, for all elements in cluster1, c1 and for all elements in cluster2, c2, you will add up the distances and then divide by the total number of pairs. It sort of does a middle ground between the *single link* and *complete link*.




In **Centroids**, **you construct a centroid representation for both the clusters** shown as blue dots above **which is just the mean**. **The distance between the clusters becomes the distance between the centroids**.




**Ward's method is using the same objective function as *K-Means***. Remember that the objective function in *K-Means* – you are looking at the total variance around the centroids. So, for each centroid, you get all of the points assigned to that centroid and look at the squared deviation of points from the centroid. And that is what *ward's method* is doing. *Ward's method* is saying that before I do the merge, I have the red and yellow clusters, each cluster has a centroid and I have certain deviations from the centroid to the points in the clusters. And the total deviation is just the sum of the deviations. Now, suppose that you merge red and yellow clusters, if you did that, you will end up with one cluster and that one cluster will now have one centroid and then I will have a deviation between the new centroid and every point within the new cluster. So that is another number, and **what *ward's method* looks at is that it looks at how much the total deviation will change between the two. Is it going to go up or down? It always has to go up! Any time you merge things, points are going to end up further away from the centroid**. It's never going to happen the other way. But the question is how much larger this total deviation is. So, ***ward's method* picks the pair of clusters that results in the smallest increase of the variance and that is the pair of clusters that you merge and you advance to the next iteration**.




There are more *distance measure* but these are the dominant ones and they will produce very different results. **If you have a set of data points and you run hierarchical clustering with different inter-cluster distances, you will end up with very different results**.

Using *Euclidean distance*, you can use any of these above methods. But *Euclidean distance* assumes that your data attributes is numeric and sometimes they are not. Sometimes you are working with a data for which *Euclidean distance* does not make sense. For example, in the case of categorical attributes, *Euclidean distance* does not make sense.

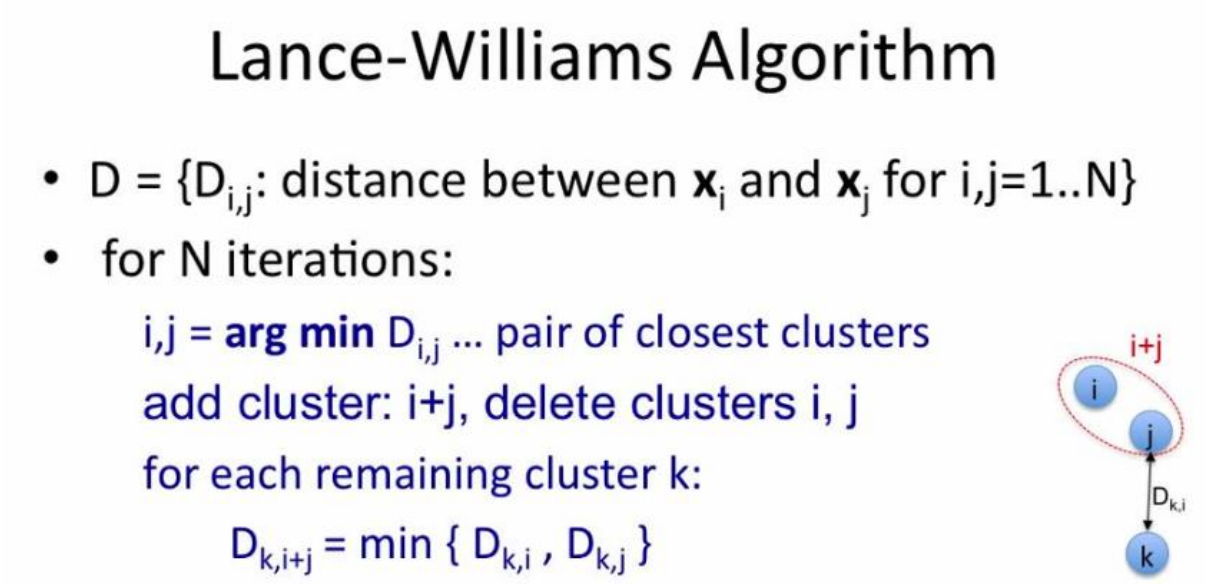
*Single, Complete* & *Average* link algorithms can be used with whatever attributes you have, as long as you can compute the distance (any distance metric). Whereas, *Centroids* & *Ward's method* presume that you have numeric attributes because you are going to be computing centroids/means. And you cannot compute mean for categorical or non-numeric data.

Reference

# Lance-Williams Algorithm

It turns out that there is a nice algorithm called the ***Lance-Williams algorithm*** which **allows you to implement all of the above-mentioned *distance measures* and a few more using one algorithm**. We **start looking at it from the *single-link* perspective**. So, the distance is just the distance between the individual points. How does this work?

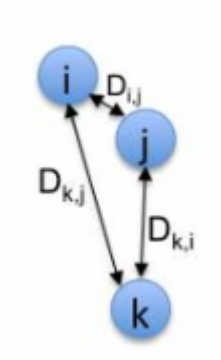
You start with a distance matrix 'D' where for each instance $x_i$ and $x_j$, you have a distance between them defined in whatever way. The way that this algorithm works is:



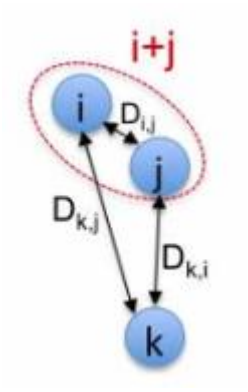

1. You will have 'N' iterations because in each iteration you merge two clusters
    a. Find the pair of closest clusters to each other; 'i' & 'j' having the minimum distance indices
2. Take the two closest clusters, merge them together to create a new cluster 'i+j' which is added to the collection of existing clusters, delete the individual clusters 'i' & 'j'
3. The clever part of the algorithm is – for each remaining cluster k (which wasn't merged), do:
    a. Update the distance from cluster 'k' to the newly created cluster 'i+j'. So, you will throw away the distance from k to i and the distance from k to j and you will create a new distance between k and the newly created cluster i+j. And the way you do that **for *single link*** is simple – **take the minimum of the distance from k to i and the distance from k to j**. This makes sense because there might be some element in cluster j or some element in cluster i which is closer to cluster k. So, you take the minimum of the two inter-cluster

distances $D_{k,i}$ and $D_{k,j}$. And this will be the new distance from k to i+j. Or, $\mathbf{D_{k,i+j} = \min\{ D_{k,i}, D_{k,j}\}}$

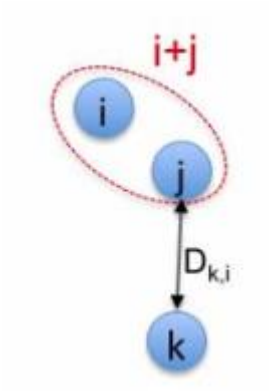
In the given example, the clusters ‘i’ & ‘j’ are closest to each other which gets merged together




to create a new cluster ‘i+j’




get rid of the individual clusters ‘i’ & ‘j’; instead, you just have a single cluster ‘i+j’




here, you are updating the distance from cluster ‘k’ to the newly created cluster ‘i+j’.

The good thing about this algorithm is that you can make a very small change – you are not changing the code, you are changing the update for the distance to the following quantity:

$$D_{k,i+j} = \alpha_i D_{k,i} + \alpha_j D_{k,j} + \beta D_{i,j} + \gamma |D_{k,i} - D_{k,i}|$$

And with that, you can implement any of the algorithms discussed earlier (above).

## Lance-Williams Algorithm

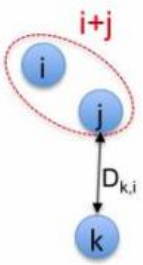
- $D = \{D_{i,j}$: distance between $\mathbf{x}_i$ and $\mathbf{x}_j$ for i,j=1..N$\}$
- for N iterations:

  i,j = **arg min** $D_{i,j}$ … pair of closest clusters

  add cluster: i+j, delete clusters i, j

  for each remaining cluster k:

  $$D_{k,i+j} = \alpha_i D_{k,i} + \alpha_j D_{k,j} + \beta D_{i,j} + \gamma |D_{k,i} - D_{k,i}|$$




So, you can use the following *inter-cluster distance measures*:

1. Single link
2. Complete link
3. Group average link
4. Weighted Group Average
5. Centroid link
6. Ward link

Let's look at that.

Before, we had the minimum of the distance from k to i ($D_{k,i}$) and distance from k to j ($D_{k,j}$). You just take the minimum of them or the closest one-

$$D_{k,i+j} = \min \{ D_{k,i} , D_{k,j} \}$$

And what are we doing now? Now, we are taking the distance from k to i or, $D_{k,i}$, $D_{k,j}$, $D_{i,j}$, and you are putting some weights on these distances. And the weights are given by some constants. So, you can take these constants, plug them into the given equation and get the different clustering algorithms.

$$D_{k,i+j} = \alpha_i D_{k,i} + \alpha_j D_{k,j} + \beta D_{i,j} + \gamma |D_{k,i} - D_{k,i}|$$

# Lance-Williams Algorithm

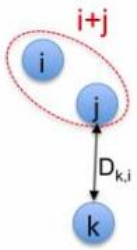
- D = {$D_{i,j}$: distance between $\mathbf{x}_i$ and $\mathbf{x}_j$ for i,j=1..N}
- for N iterations:

  i,j = **arg min** $D_{i,j}$ … pair of closest clusters

  add cluster: i+j, delete clusters i, j

  for each remaining cluster k:

  $$D_{k,i+j} = \alpha_i D_{k,i} + \alpha_j D_{k,j} + \beta D_{i,j} + \gamma |D_{k,i} - D_{k,i}|$$




| Method | $\alpha_i$ | $\alpha_i$ | $\beta$ | $\gamma$ |
|---|---|---|---|---|
| Single linkage | 0.5 | 0.5 | 0 | -0.5 |
| Complete linkage | 0.5 | 0.5 | 0 | 0.5 |
| Group average | $\frac{n_i}{n_i+n_j}$ | $\frac{n_j}{n_i+n_j}$ | 0 | 0 |
| Weighted group average | 0.5 | 0.5 | 0 | 0 |
| Centroid | $\frac{n_i}{n_i+n_j}$ | $\frac{n_j}{n_i+n_j}$ | $\frac{-n_i \cdot n_j}{(n_i+n_j)^2}$ | 0 |
| Ward | $\frac{n_i+n_k}{(n_i+n_j+n_k)}$ | $\frac{n_j+n_k}{(n_i+n_j+n_k)}$ | $\frac{-n_k}{(n_i+n_j+n_k)}$ | 0 |



Let's go through an example for how this works for ***single link*** inter-cluster distance measure.

For ***single link***, we need $D_{k,i+j}$-

$$D_{k,i+j} = \alpha_i D_{k,i} + \alpha_j D_{k,j} + \beta D_{i,j} + \gamma |D_{k,i} - D_{k,i}|$$


| Method | $\alpha_i$ | $\alpha_i$ | $\beta$ | $\gamma$ |
|---|---|---|---|---|
| Single linkage | 0.5 | 0.5 | 0 | -0.5 |
| Complete linkage | 0.5 | 0.5 | 0 | 0.5 |
| Group average | $\frac{n_i}{n_i+n_j}$ | $\frac{n_j}{n_i+n_j}$ | 0 | 0 |
| Weighted group average | 0.5 | 0.5 | 0 | 0 |
| Centroid | $\frac{n_i}{n_i+n_j}$ | $\frac{n_j}{n_i+n_j}$ | $\frac{-n_i \cdot n_j}{(n_i+n_j)^2}$ | 0 |
| Ward | $\frac{n_i+n_k}{(n_i+n_j+n_k)}$ | $\frac{n_j+n_k}{(n_i+n_j+n_k)}$ | $\frac{-n_k}{(n_i+n_j+n_k)}$ | 0 |

**Single link:**

$D_{k,i+j}$
= ½ ($D_{ki}$ + $D_{kj}$ - |$D_{ki}$-$D_{kj}$|)
= min {$D_{ki}$ , $D_{kj}$}

The claim is that the formula reduces to $\mathbf{D_{k,i+j} = \min\{ D_{k,i}, D_{k,j}\}}$. Why is that? Look at $D_{k,i}$ and $D_{k,j}$. Out of them, one is smaller and the other is larger, you just don't know which one without the *minimum*. This is the difference between the minimum and the maximum: $|D_{ki} - D_{kj}|$. **What if you take the maximum and subtract from it the difference between the maximum and minimum?** What do you get? **You get the minimum**.

$$D_{k,i+j} = \alpha_i D_{k,i} + \alpha_j D_{k,j} + \beta D_{i,j} + \gamma |D_{k,i} - D_{k,i}|$$


| Method | $\alpha_i$ | $\alpha_i$ | $\beta$ | $\gamma$ |
|---|---|---|---|---|
| Single linkage | 0.5 | 0.5 | 0 | -0.5 |
| Complete linkage | 0.5 | 0.5 | 0 | 0.5 |
| Group average | $\frac{n_i}{n_i+n_j}$ | $\frac{n_j}{n_i+n_j}$ | 0 | 0 |
| Weighted group average | 0.5 | 0.5 | 0 | 0 |
| Centroid | $\frac{n_i}{n_i+n_j}$ | $\frac{n_j}{n_i+n_j}$ | $\frac{-n_i \cdot n_j}{(n_i+n_j)^2}$ | 0 |
| Ward | $\frac{n_i+n_k}{(n_i+n_j+n_k)}$ | $\frac{n_j+n_k}{(n_i+n_j+n_k)}$ | $\frac{-n_k}{(n_i+n_j+n_k)}$ | 0 |

**Single link:**

$D_{k,i+j}$
$= ½ (D_{ki} + D_{kj} - |D_{ki}-D_{kj}|)$
$= \min \{D_{ki}, D_{kj}\}$

$\min_{a,b} = \max_{a,b} - |a-b|$



One of the two ($D_{ki}$, $D_{kj}$) is the maximum. For example, say that $D_{kj}$ is the bigger one. You are taking $D_{kj}$ and you are subtracting the difference between $D_{ki}$ & $D_{kj}$, so you are going to end up with the smaller one. Here, the smaller one is $D_{ki}$ which gives you twice the minimum or, 2x $D_{ki}$ and you take the half of it.

You can make the same argument for ***complete link***. **Here, the only difference is that instead of having a minus sign, you have a plus sign**. You have a minimum + a maximum + the difference between the minimum and the maximum. If you take the minim and add ot it the difference, this gives you the maximum. Now, you have twice the maximum, divided by 2, you get just the maximum.

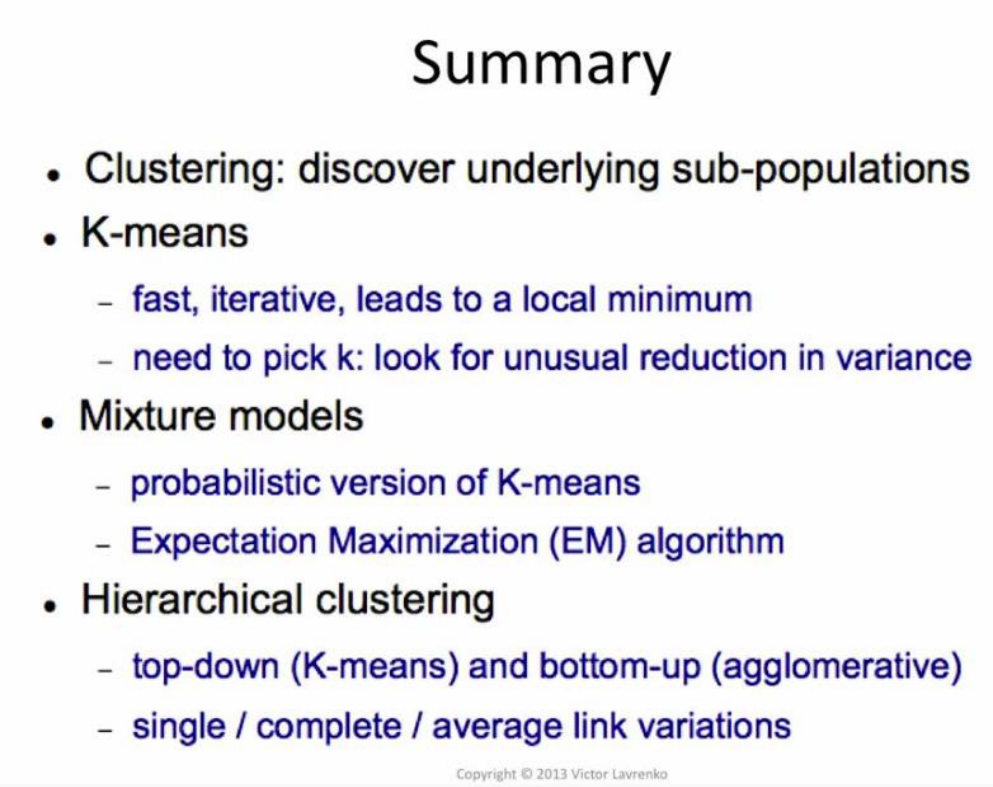
This is a nice way to implement all kinds of *aggregation* functions. So, *single link, complete link* & *average link* are *aggregation* functions. This is how we take individual distances and aggregate them into one. And it turns out that with *Lance-William* algorithm, you can model a bunch of different *aggregation* functions just by substituting different constants.

This is also practical because if you do this, then it should be obvious that **the cost of the entire algorithm is *cubic***. You are going to have 'N' iterations at the top, merging two clusters at each step, then, you will find the pair of clusters having the smallest inter-cluster distance. This is going to take you $n^2$ steps. And then for each remaining cluster, 'n' steps, you will have a constant time update.

Naïve implementations of hierarchical clustering usually end up having horrendous complexity.

Reference

# Hierarchical Clustering: Summary




One way to think about *Mixture models* is that they are almost like K-Means, only it's *soft clustering*. So, an instance or data point can belong to multiple clusters and you have a degree of association between the clusters and a data point(s). And another cool thing about *mixture* models is that It (kind of) changes the definition of distance as it goes along because each cluster ends up having its own variance or covariance matrix and this will change the idea of distance.

In *hierarchical clustering* you are not picking/choosing a fixed number of clusters, rather you are looking for an entire spectrum. And, we had different variations for that in terms of different algorithms.

Reference