Hierarchical Clustering

Hierarchical clustering (sometimes called hierarchical cluster analysis or HCA) is a method of cluster analysis where clusters are built as a tree (a hierarchy) of clusters.

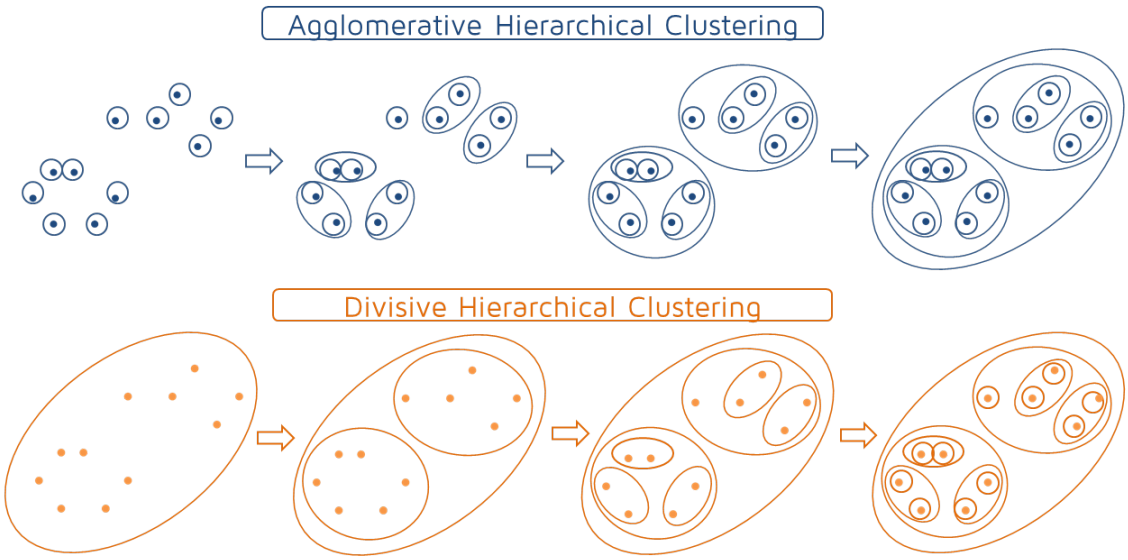
Hierarchical Clustering has these advantages:

- There is **no need to choose the number of clusters** ahead of time (although the number of clusters at each level is preselected a parameter in most cases).
- There is no need to rerun the algorithm to choose a different number or granularity of clusters. The number and size of **clusters can be chosen post-facto** using dendrograms to both visualize and choose clusters.
- Most algorithms **allow a choice of distance metric** (whereas k-means was restricted to Euclidean distance).
- Hierarchical algorithms can **often find more complex shapes** than k-means or Gaussian Mixture models such as those involving clusters surrounded by other clusters.

Hierarchical clustering strategies fall into **two types**:

- **Divisive**: This is a "**top down**" approach where **all observations start in one cluster, and splits are performed recursively** as one moves down the hierarchy. If we apply this approach fully, then we end up with each observation in it's own cluster (each leaf node of the dendrograms is a single observation). Recursive k-means is an example of a divisive algorithm.
- **Agglomerative**: This is a "**bottom up**" approach where **each observation starts in its own cluster, and pairs of clusters are merged** as one moves up the hierarchy. If we apply this approach fully, then we end up with a single root cluster (the root of the dendrogram) that contains all observations. Single-linkage is an example of an agglomerative algorithm.

Below is an illustration, taken from the QuantDare (a blog about applying machine learning algorithms in the area of asset management), that shows how a set of data could hypothetically be clustered using the two approaches. It's worth noting that this is a simple illustration and it should not be taken to suggest that the approaches will end up with exactly the same results; the final clusters depend more of the specific algorithms, metrics and terminating conditions chosen than the high level strategy.



Agglomerative vs Divisive Clustering from QuantDare blog

Divisive clustering

A common example of Divisive clustering is recursive k-means. This is what it sounds like. We apply the k-means algorithm to a complete set of observations, resulting in k clusters. We then apply it again to the k clusters. If we fix k at k = 2, then we end up with a binary tree dendrogram. However, we can choose k to be anything we want at each recursion level and so create more complex structures.

Aspects of a divisive approach:

- **Which algorithm** to we recurse over the data? In the previous example, we talked about k-means, but it could be any of the the clustering algorithms we have talked about (and any we haven't!)
- **How many clusters per split**. Basically, what is the value of k at each recursion level?
- What is our **termination condition**; do we continue splitting or do we stop. There a number of common metrics to use:
    - **Max cluster size**: stop recursing when a cluster is encountered that falls below a threshold. So when a cluster is small enough, we are done.
    - **Max cluster spread**: stop recursing when the cluster radius falls below a threshold. So when the cluster is 'compact' enough, we are done.
    - **Number of clusters**: split until a specified number of clusters is reached. So when we have enough clusters, we are done. Of course, this means we have to pre-specify the number of clusters, but we may choose a higher k than we really want, then we can choose our final clusters post-facto (see the discussion below).

Agglomerative clustering

A very common algorithm using the Agglomerative approach is called Single Linkage clustering. The algorithm requires that two important metrics are chosen:

- **The distance function**
  This measures the pairwise distance (dissimilarity) between two observations.
- **The linkage function**
  The linkage function uses the pairwise distance function to determine the distance between sets of observations (clusters).

In single linkage clustering, the linkage function returns the distance between the two observations (one from each set) that are closest to each other; it is the intuitive notion of close edges. The algorithm must

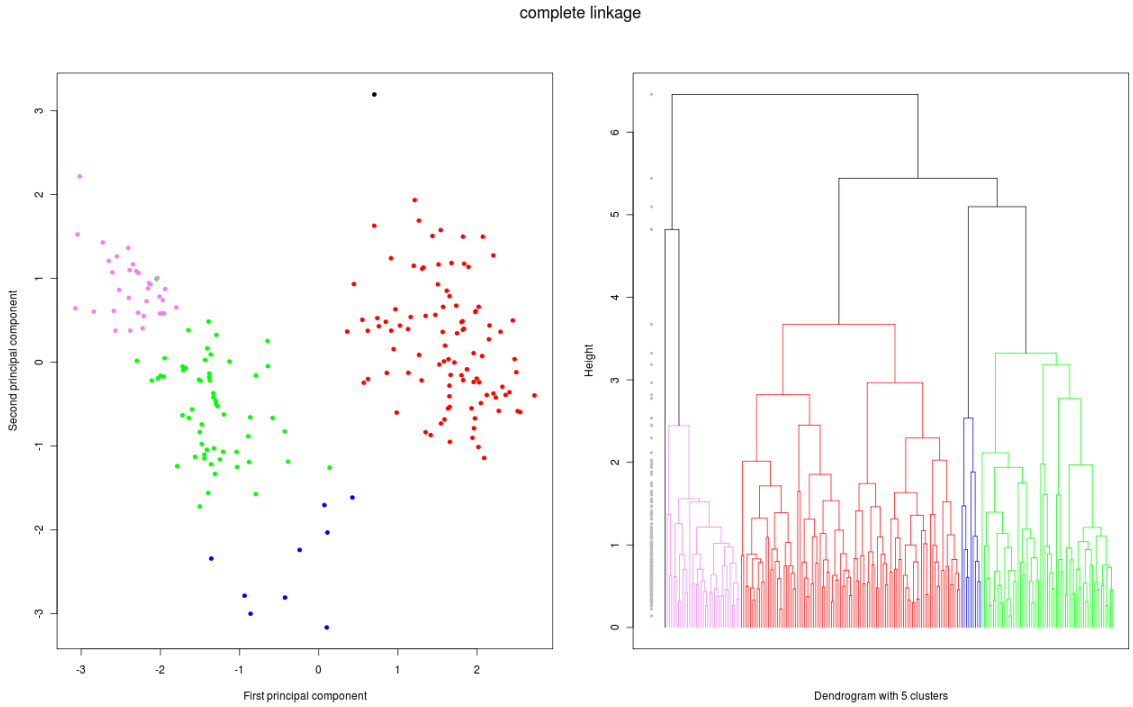
take each point in the first set and compare it against each point in the second set to find the minimum distance between the two sets.

Basic Single Linkage Clustering

1. Initialize each point to be it's own cluster
2. For each cluster, use the linkage cluster to find the closest cluster.
3. Choose the two clusters that are closest, and merge them.
4. Repeat steps 2,3

There are alternatives to single linkage that use the same basic agglomerative algorithm, but use a different linkage function.

- Complete linkage clustering uses the maximum distance between points the two sets. This means that the two sets with the closest maximum distance are merged.
- Centroid linkage uses the minimum distance between the cluster centroids. This means that the two clusters with the closest centroids are merged.
- Ward's minimum variance criterion minimizes the total within-cluster variance at each merge.

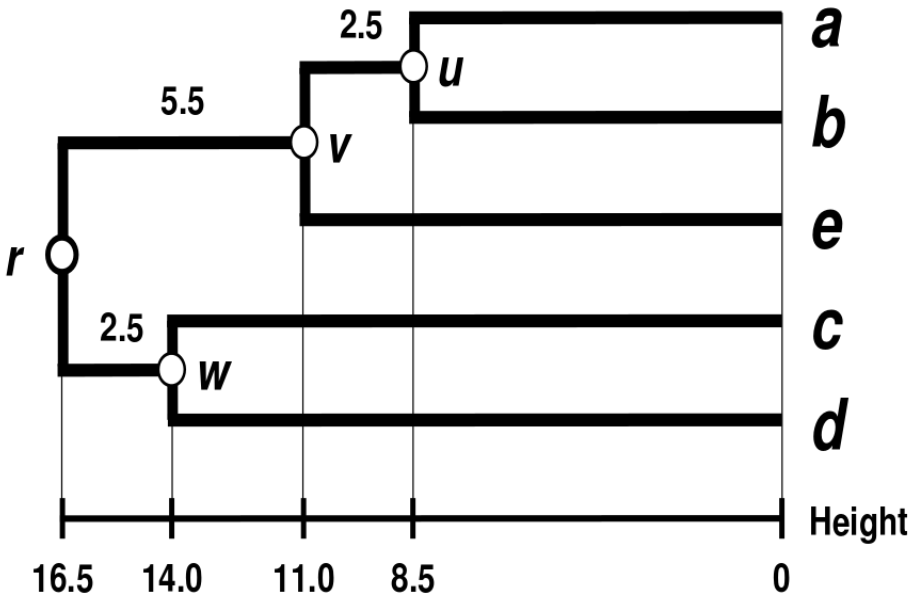


Example of dendrogram created during agglomerative clustering taken from Wikimedia

The dendrogram

A dendrogram is a tree diagram frequently used to illustrate the arrangement of the clusters produced by hierarchical clustering.

- Individual obserations are leaf clusters (a cluster of one). These are ordered on an axis by cluster (so that the visual looks good)
- The connections between clusters (called clades) are positioned to indicate the distance between the two clusters.

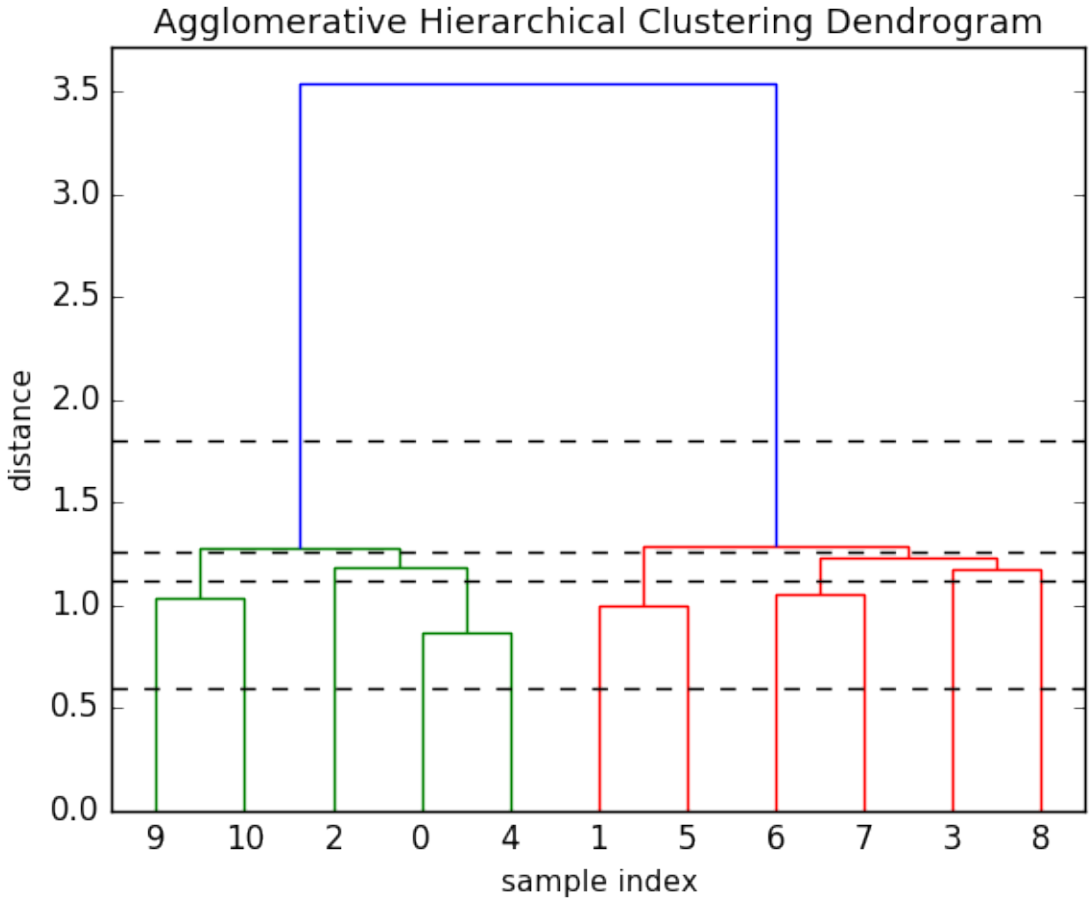


Dendrogram with clades at 8.5, 11, 14, 16 labeled u, v, w, r respectively; taken from Wikimedia

The path along the dendrogram indicates the clusters that each point belongs to and the orders of the merges. In the above illustration, we can see

- cluster b (which is a single point, since it is a leaf) was merged with cluster a to form cluster u. The clade at this point indicates the distance between clusters a and b as 8.5.
- Then cluster u was merged with cluster e to form cluster v. The clade indicates that the distance between cluster u and cluster e as 11.0.
- Finally, cluster v was merged with cluster w to form cluster r, which contains all points.

The fact that the clades are placed to indicate the distance between the two connected clusters is important; the fact can be used when partitioning the dendrogram into final clusters. One simple way to partition into final clusters is to simply choose a distance D* and use this to 'cut' across the dendrogram. We can see this in the illustration below.



Dendorgram taken from QuantDare

In the cut at D* = 0.5, we are left with each leaf node as a cluster (which isn't useful, since that is just our original observations). The cut at D* = 1.8 creates two clusters; the green subtree (9,10,2,0,4) and the red subtree (1,5,6,7,3,8). The number of clusters will be the number of vertical lines that are crossed by the cut. The cut at D* = 1.1 will produce 7 clusters; (9),(10),(2),(0,4),(1,5),(6,7)(3),(8).

Varying D* results in different possible clusterings of the dataset through different granularities, going from really, really fine granularity (where D* is close to zero) all the way up to very coarse granularities. For the resulting clusters produced by this cut, there are no pair of clusters at a distance less than D* that have not already been merged. So that means that D* is the minimum distance between our clusters at this level of the clustering – we can choose our set of clusters by choosing the desired minimum distance between clusters.

More on 'cutting' the dendrogram

- The prior examples shows cuts at a constant D*, but that does not have to be the case. We can choose to cut different clusters at different D*.
- a smaller number of clusters are often preferable (so larger D*) for visualizations.
- For applications such as outlier detection, we might calculate a metric;
    - use a calculated distance threshold D* (as in previous examples)
    - make cuts based on an 'Inconsistancy Coefficient' rather than distance
        - Compare the height of a merge to the average heights of the merge points below it.
        - If the merge being considered is substantially higher than it's children, then it is joining two subsets that are relatively far apart compared to the members of each subset.

This then may indicate that the subsets should be separate clusters (so we should cut below the merge point).

- There is no 'correct' or 'incorrect' method for choosing the clusters based on the dendrogram. Some ways are more useful than others, but it will depend on the data and what you are attempting to accomplish. It is better to think of hierarchical clustering as a way to produce different possible clusters.

Computational Considerations for Agglomerative Clustering

- Computing all pairs of distances is expensive. The brute force approach is $O(N^2 \log(N))$ complexity where N is the number of data points.
- Smart implementations use the triangle inequality to rule out some candidate pairs.
- The best known algorithm is still $O(N^2)$, which is very expensive for large N.

Chaining in Single Linkage Clustering

In single linkage, we can get situations where points that are very far apart may end up in the same cluster if there is a chain of pairwise close points between them. For instance, we could get a long line of points put into the same cluster, so the end are very far apart. This is not necessarily bad; this may be a cluster shape inherent in the data. However, in some cases it is an unwanted artifact of the linkage function.

Other linkage functions (like Complete linkage or Ward Criterion) are more robust against such cluster shapes, but add restrictions to the shapes that are allowed. So choosing the linkage function is critical to the final result.

Hidden Markov models

In the clustering we've looked at so far, the index of a given data point plays no role in the resulting clustering of that data point. We could permute all our data indices, and cluster that permuted data, and we would get out exactly the same results. But what if the order of the data points actually mattered, like in time series where the label, in particular the time stamp associated with each data point, is critical to the analysis of that data. So, for example, a univaried time series where the value of the observation is along the y axis and the time stamp of the observation is along the x axis. The goal might be to parse this time series into different dynamic states. We can think of this as different clusters appearing in the data set over time. The structure of this data across time can actually help us uncover patterns. In the context of time series, we can think of this as a segmentation task.

This notion of switching between different dynamic states appears in lots of different applications. So for example, given conference audio of people taking turns speaking in a meeting, it could be useful to be able to segment that audio into who is speaking at various times during the meeting. Or another example; given Fitbit observations of a person doing a set of exercise routines, it would be useful to segment the person's activities given that data over time.

A hidden Markov model (HMM) can be considered a generalization of a mixture model where the hidden variables (or latent variables), which control the mixture component to be selected for each observation are related through a Markov process rather than independent of each other. For instance, in time series data, the state at a point in time is dependent in part on the previous point's state. So the series of states becomes part of the model.

An HMM is very similar to the type of mixture models described earlier in this course (EM of Gaussian Mixtures). So just like in a mixture model, every observation is associated with a cluster indicator. When talking about HMMs the cluster is described as a state. The critical difference is the fact that the

probability of a given cluster assignment depends on the value of the cluster assignment for the previous observation. This is how the time dependency is captured.

With an HMM:

- You can **compute the maximum likelihood estimate** of your HMM parameters using an algorithm that is very similar to what we talked about for Mixture models. But in this context it's called the Baum Welch algorithm for historical reasons..
- You can **compute the most likely state sequence** given your model parameters by fixing the model parameters and looking at the most likely sequence of states or cluster assignments using the **Viterbi algorithm**. This is an example of something called **dynamic programming**, which allows you to efficiently do this search over a set of states.
- You can also use dynamic programming to **form soft assignments** of the cluster variables using something called the **forward-backward algorithm**. And these soft assignments play a role in the **Baum Welch algorithm** just like the soft assignments played a role in standard Algorithm for mixture models we described earlier in the course.

Other Clustering and Retrieval topics:

Retrieval

- Other distance metrics
- Distance metric learning

Clustering

- Nonparametric Clustering
- Spectral Clustering

Related Ideas

- **Density Estimation**
  From Wikipedia: In probability and statistics, density estimation is the construction of an estimate, based on observed data, of an unobservable underlying probability density function. The unobservable density function is thought of as the density according to which a large population is distributed; the data are usually thought of as a random sample from that population.
- **Anomaly Detection**
  From Wikipedia: In data mining, anomaly detection (also outlier detection) is the identification of items, events or observations which do not conform to an expected pattern or other items in a dataset. **Unsupervised anomaly detection** techniques detect anomalies in an unlabeled test data set under the assumption that the majority of the instances in the data set are normal by looking for instances that seem to fit least to the remainder of the data set.