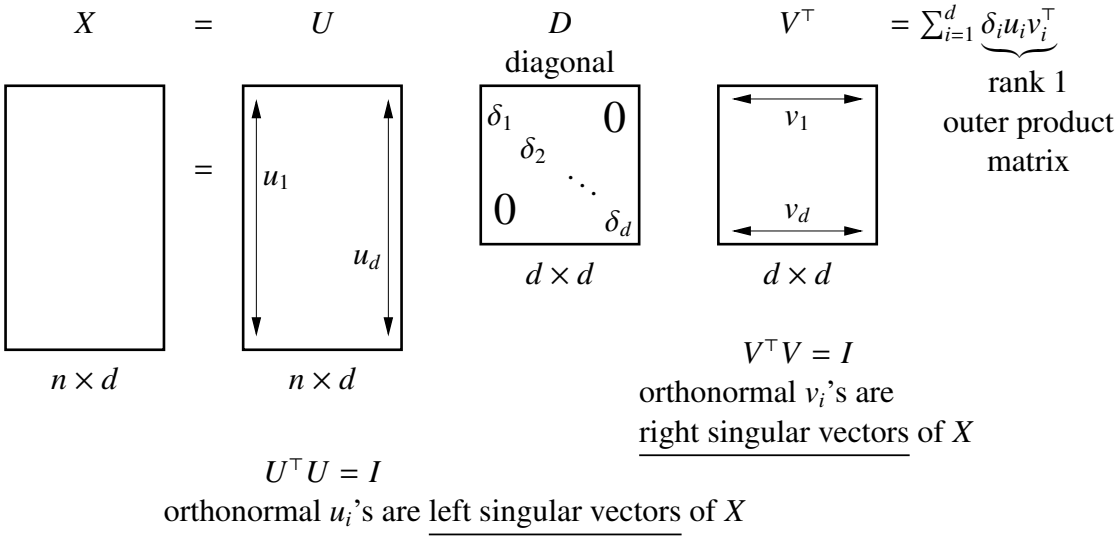
21 The Singular Value Decomposition; Clustering

The Singular Value Decomposition (SVD) [and its Application to PCA]

Problems: Computing $X^\top X$ takes $\Theta(nd^2)$ time.
$X^\top X$ is poorly conditioned $\rightarrow$ numerically inaccurate eigenvectors.
[The SVD improves both these problems.]

[Earlier this semester, we learned about the eigendecomposition of a square, symmetric matrix. Unfortunately, nonsymmetric matrices don't eigendecompose nearly as nicely, and non-square matrices don't have eigenvectors at all. Happily, there is a similar decomposition that works for all matrices, even if they're not symmetric and not square.]

Fact: If $n \geq d$, we can find a singular value decomposition $X = UDV^\top$



$V^\top V = I$
orthonormal v_i's are
right singular vectors of X

$U^\top U = I$
orthonormal u_i's are left singular vectors of X

[Draw this by hand; write summation at the right last. svd.pdf]

Diagonal entries $\delta_1, \ldots, \delta_d$ of D are nonnegative singular values of X.

[Some of the singular values might be zero. The number of nonzero singular values is equal to the rank of the centered design matrix X. If all the sample points lie on a line, there is only one nonzero singular value. If the points span a subspace of dimension r, there are r nonzero singular values.]

[If $n < d$, an SVD still exists, but now U is square and V is not.]

Fact: v_i is an eigenvector of $X^\top X$ w/eigenvalue δ_i^2.
Proof: $X^\top X = VDU^\top UDV^\top = VD^2V^\top$
which is an eigendecomposition of $X^\top X$.

[The columns of V are the eigenvectors of $X^\top X$, which is what we need for PCA. If $n < d$, V will omit some of the eigenvectors that have eigenvalue zero, but those are useless for PCA. The SVD also tells us the eigenvalues, which are the squares of the singular values. By the way, that's related to why the SVD is more numerically stable: the ratios between singular values are smaller than the ratios between eigenvalues.]

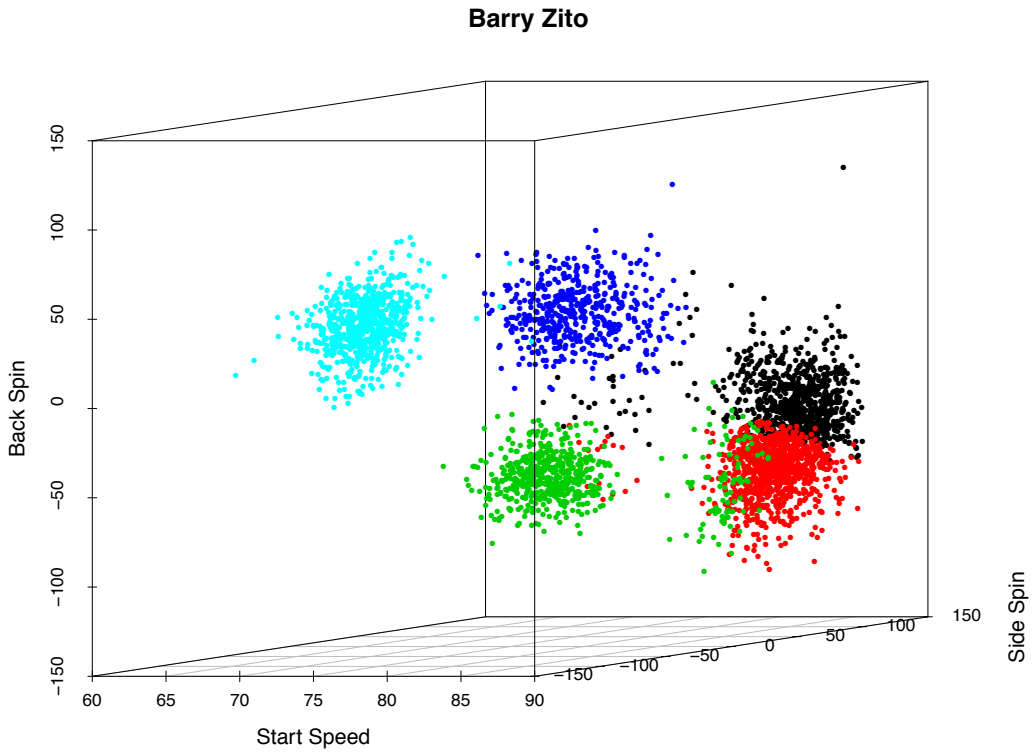
Fact: We can find the k greatest singular values & corresponding vectors in $O(ndk)$ time.
[So we can save time by computing some of the singular vectors without computing all of them.]
[There are approximate, randomized algorithms that are even faster, producing an approximate SVD in $O(nd \log k)$ time. These are starting to become popular in algorithms for very big data.]
[https://code.google.com/archive/p/redsvd/]

Important: Row i of UD gives the coordinates of sample point X_i
in principal components space (i.e. $X_i \cdot v_j$ for each j).
[So we don't need to explicitly project the input points onto principal components space; the SVD has already done it for us.]

CLUSTERING

Partition data into clusters so points in a cluster are more similar than across clusters.
Why?

- Discovery: Find songs similar to songs you like; determine market segments
- Hierarchy: Find good taxonomy of species from genes
- Quantization: Compress a data set by reducing choices
- Graph partitioning: Image segmentation; find groups in social networks



4-Seam Fastball	2-Seam Fastball	Changeup	Slider	Curveball
Black	Red	Green	Blue	Light Blue

zito.pdf (from a talk by Michael Pane) [k-means clusters that classify Barry Zito's baseball pitches. Here we discover that there really are distinct classes of baseball pitches.]

k-Means Clustering aka Lloyd's Algorithm (Stuart Lloyd, 1957)

Goal: Partition n points into k disjoint clusters.
Assign each input point X_i a cluster label $y_i \in [1, k]$.
Cluster i's mean is $\mu_i = \frac{1}{n_i} \sum_{y_j=i} X_j$, given n_i points in cluster i.

Find y that minimizes $\sum_{i=1}^{k} \sum_{y_j=i} |X_j - \mu_i|^2$ [Sum of the squared distances from points to their cluster means.]

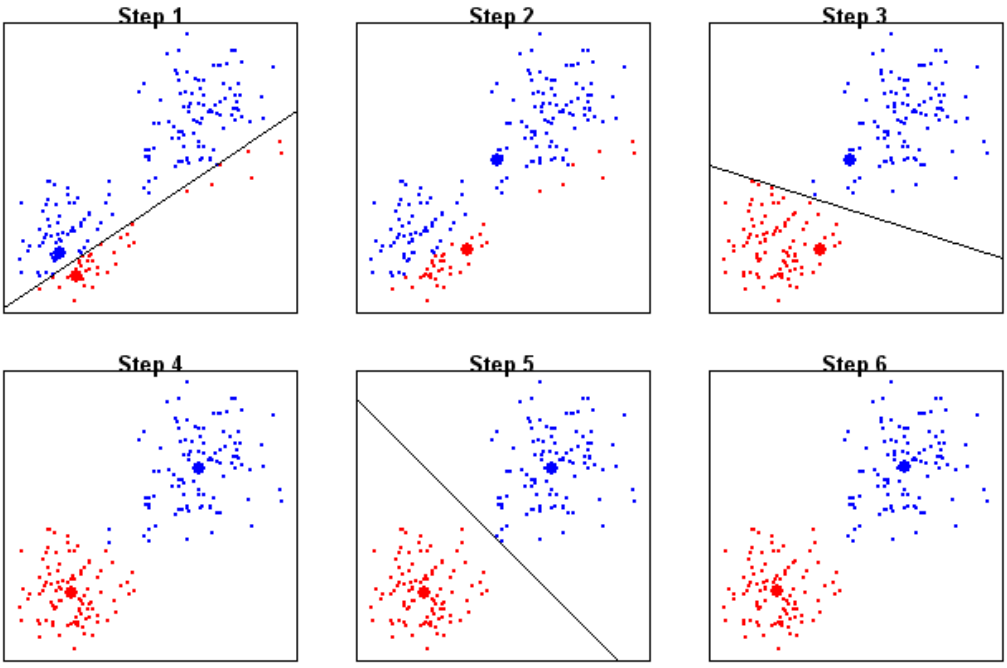
NP-hard. Solvable in $O(nk^n)$ time. [Try every partition.]

k-means heuristic: Alternate between
(1) y_j's are fixed; update μ_i's
(2) μ_i's are fixed; update y_j's
Halt when step (2) changes no assignments.

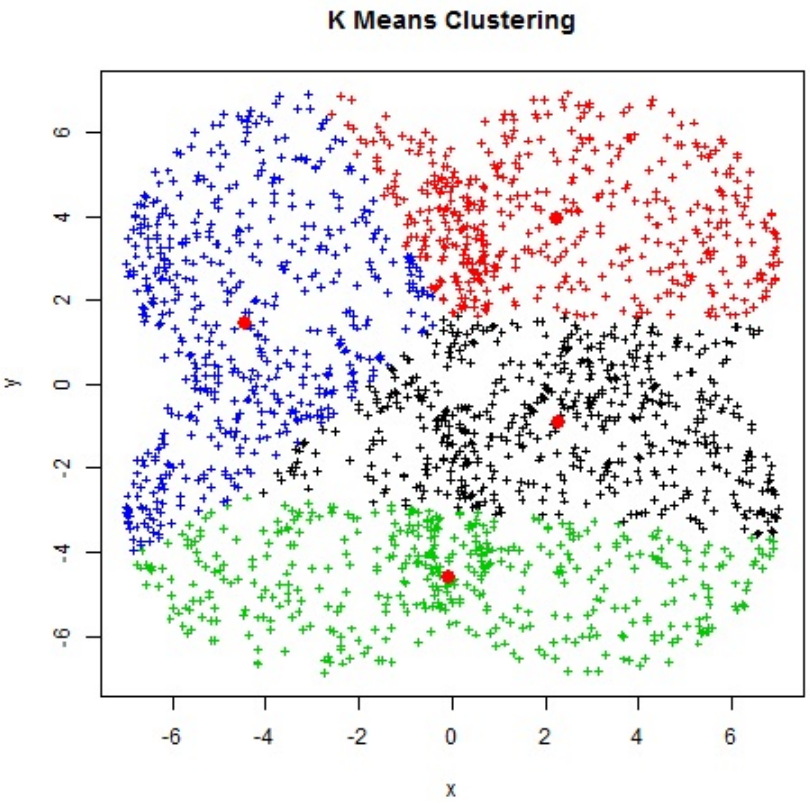
[So, we have an assignment of points to clusters. We compute the cluster means. Then we reconsider the assignment. A point might change clusters if some other's cluster's mean is closer than its own cluster's mean. Then repeat.]

Step (1): One can show (calculus) the optimal μ_i is the mean of the points in cluster i.
[This is easy calculus, so I leave it as a short exercise.]

Step (2): The optimal y assigns each point X_j to the closest center μ_i.
[This should be even more obvious than step (1).]
[If there's a tie, and one of the choices is for X_j to stay in the same cluster as the previous iteration, always take that choice.]

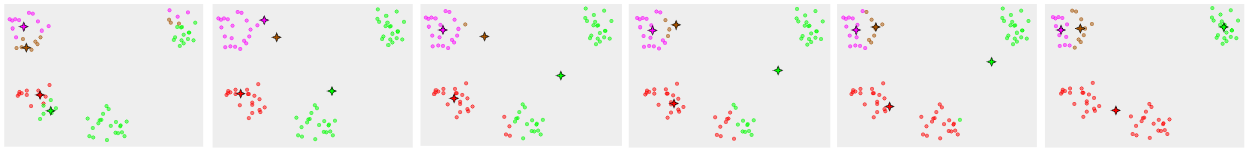


2means.png [An example of 2-means. Odd-numbered steps reassign the data points. Even-numbered steps compute new means.]



4meansanimation.gif [This is an animated GIF of 4-means with many points. Unfortunately, the animation can't be included in the PDF lecture notes.]

Both steps decrease objective fn *unless* they change nothing.
[Therefore, the algorithm never returns to a previous assignment.]
Hence alg. must terminate. [As there are only finitely many assignments.]
[This argument doesn't say anything optimistic about the running time, because we might see $O(k^n)$ different assignments before we halt. In theory, one can actually construct point sets in the plane that take an exponential number of iterations, but those don't come up in practice.]
Usually very fast in practice. Finds a local minimum, often not global.
[. . . which is not surprising, as this problem is NP-hard.]



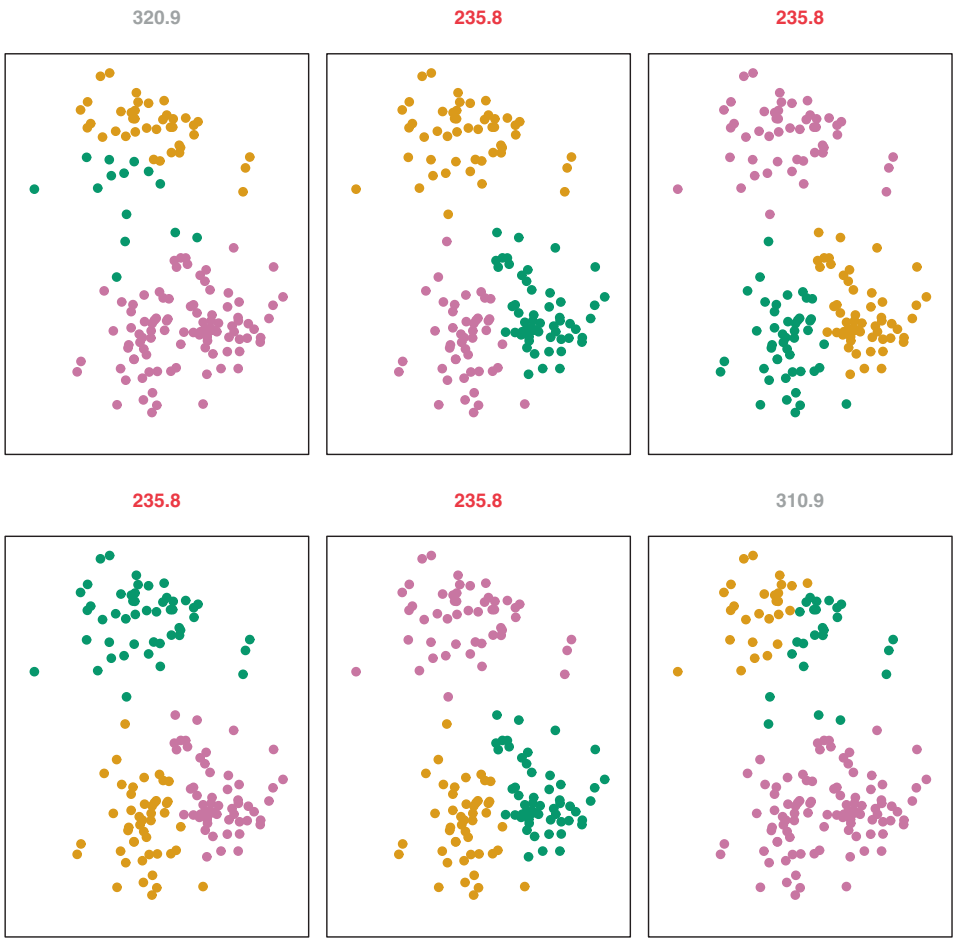
4meansbad.png [An example where 4-means clustering fails.]

Getting started:

- Forgy method: choose k random sample points to be initial μ_i's; go to (2).
- Random partition: randomly assign each sample point to a cluster; go to (1).

[Forgy seems to be better, but Wikipedia mentions some variants of k-means for which random partition is better.]

For best results, run k-means multiple times with random starts.



kmeans6times.pdf (ISL, Figure 10.7) [Clusters found by running 3-means 6 times on the same sample points. The algorithm finds three different local minima.]

[Why did we choose that particular objective function to minimize? Partly because it is equivalent to minimizing the following function.]

Equivalent objective fn: the within-cluster variation

Find y that minimizes $\sum_{i=1}^{k} \frac{1}{n_i} \sum_{y_j=i} \sum_{y_m=i} |X_j - X_m|^2$

[At the minimizer, this objective function is equal to twice the previous one. It's a worthwhile exercise to show that—it's harder than it looks. The nice thing about this expression is that it doesn't include the means; it's a function purely of the input points and the clusters we assign them to.]

Normalize the data? [before applying k-means]
Same advice as for PCA. Sometimes yes, sometimes no.
[If some features are much larger than others, they will tend to dominate the Euclidean distance. So if you have features in different units of measurement, you probably should normalize them. If you have features in the same unit of measurement, you usually shouldn't, but it depends on context.]

k-Medoids Clustering

Generalizes k-means beyond Euclidean distance. [Means aren't optimal for other distance metrics.]
Specify a distance fn $d(x, y)$ between points x, y, aka dissimilarity.
Can be arbitrary; ideally satisfies triangle inequality $d(x, y) \leq d(x, z) + d(z, y)$.
[Sometimes people use the ℓ_1 norm or the ℓ_∞ norm. Sometimes people specify a matrix of pairwise distances between the input points.]
[Suppose you have a database that tells you how many of each product each customer bought. You'd like to cluster together customers who buy similar products for market analysis. But if you cluster customers by Euclidean distance, you'll get a big cluster of all the customers who have only ever bought one thing. So Euclidean distance is not a good measure of dissimilarity. Instead, it makes more sense to treat each customer as a vector and measure the *angle* between two customers. If there's a large angle between customers, they're dissimilar.]
Replace mean with medoid, the sample point that minimizes total distance to other points in same cluster.
[So the medoid of a cluster is always one of the input points.]

[One difficulty with k-means is that you have to choose the number k of clusters before you start, and there isn't any reliable way to guess how many clusters will best fit the data. The next method, hierarchical clustering, has the advantage in that respect. By the way, there is a whole Wikipedia article on "Determining the number of clusters in a data set."]

Hierarchical Clustering

Creates a tree; every subtree is a cluster.
[So some clusters contain smaller clusters.]

Bottom-up, aka agglomerative clustering:
start with each point a cluster; repeatedly fuse pairs.
[Draw figure of points in the plane; pair clusters together until all points are in one top-level cluster.]

Top-down, aka divisive clustering:
start with all pts in one cluster; repeatedly split it.
[Draw figure of points in the plane; divide points into subsets hierarchically until each point is in its own subset.]

[When the input is a point set, agglomerative clustering is used much more in practice than divisive clustering. But when the input is a graph, it's the other way around: divisive clustering is more common.]

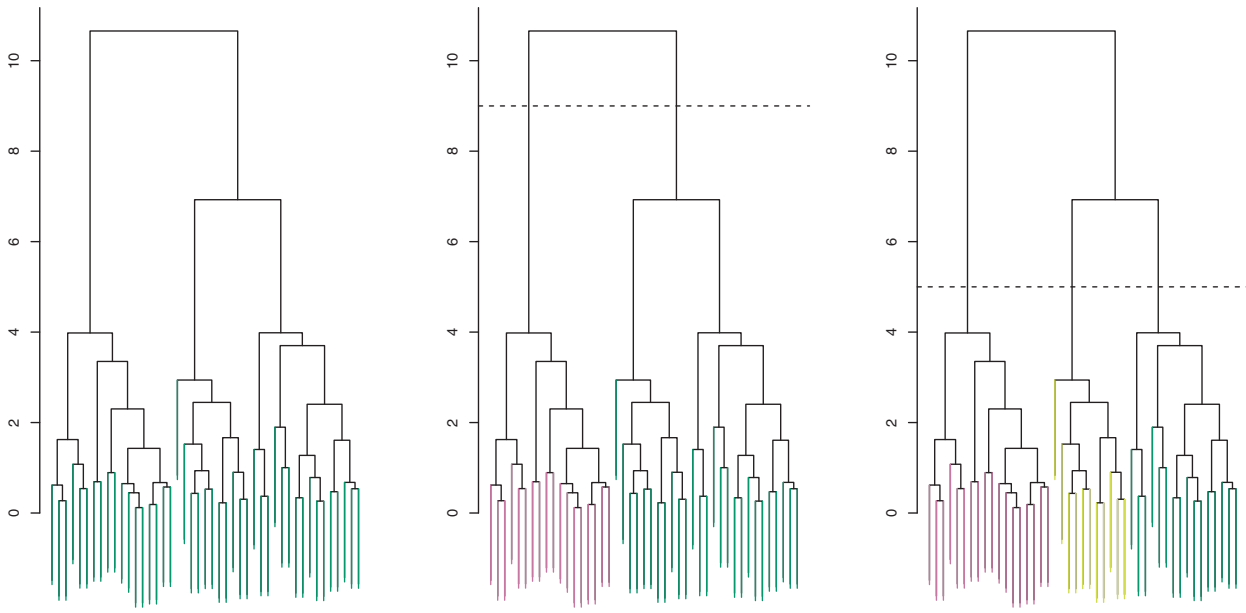
We need a distance fn for clusters A, B:

complete linkage: $d(A, B) = \max\{d(w, x) : w \in A, x \in B\}$
single linkage: $d(A, B) = \min\{d(w, x) : w \in A, x \in B\}$
average linkage: $d(A, B) = \frac{1}{|A|\,|B|} \sum_{w \in A} \sum_{x \in B} d(w, x)$
centroid linkage: $d(A, B) = d(\mu_A, \mu_B)$ where μ_S is mean of S
[The first three of these linkages work for any distance function, even if the input is just a matrix of distances between all pairs of points. The centroid linkage only really makes sense if we're using the Euclidean distance. But there's a variation of the centroid linkage that uses the medoids instead of the means, and medoids are defined for any distance function. Moreover, medoids are more robust to outliers than means.]

Greedy agglomerative alg.:
Repeatedly fuse the two clusters that minimize $d(A, B)$

Naively takes $O(n^3)$ time.
[But for complete and single linkage, there are more sophisticated algorithms called CLINK and SLINK, which run in $O(n^2)$ time. A package called ELKI has publicly available implementations.]

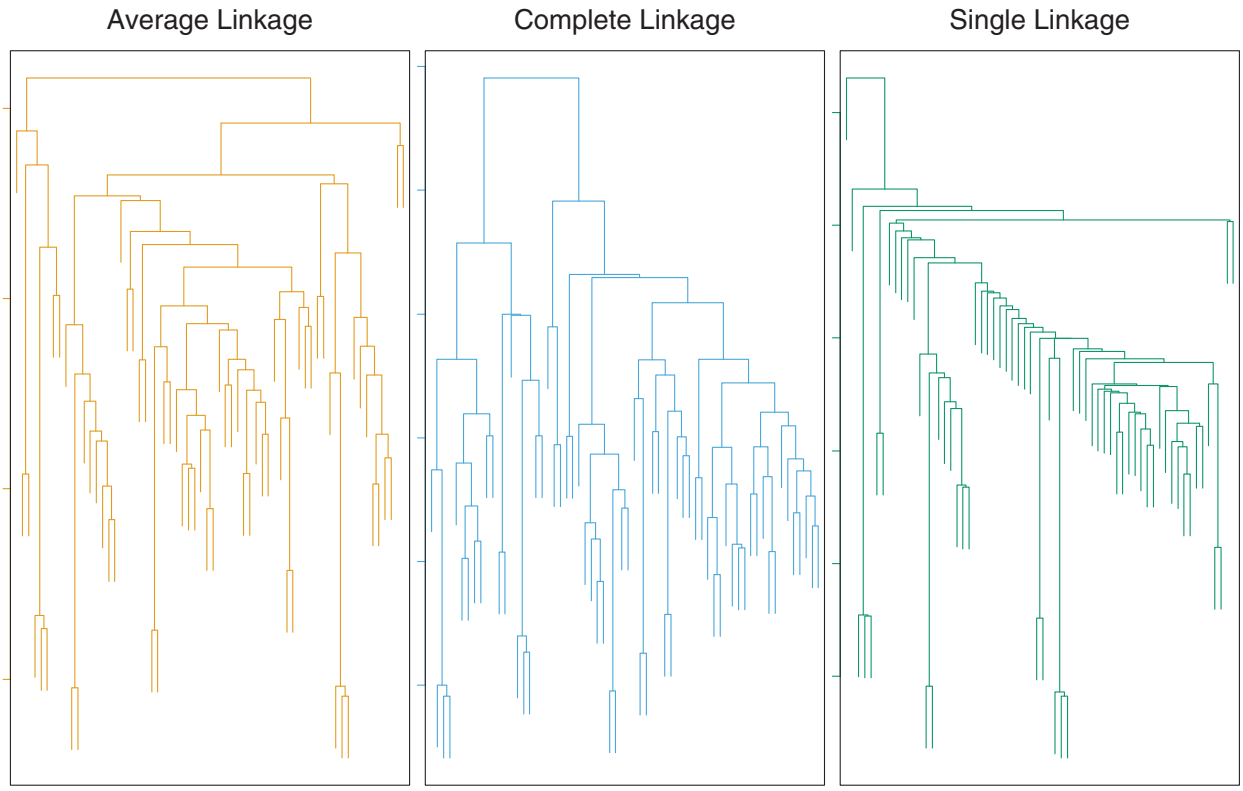
Dendrogram: Illustration of the cluster hierarchy (tree) in which the vertical axis encodes all the linkage distances.



dendrogram.pdf (ISL, Figure 10.9) [Example of a dendrogram cut into 1, 2, or 3 clusters.]

Cut dendrogram into clusters by horizontal line according to your choice of # of clusters OR intercluster distance.

[It's important to be aware that the horizontal axis of a dendrogram has no meaning. You could swap some node's left children and right children and it would still be the same dendrogram. It doesn't mean anything that two leaves happen to be next to each other.]



linkages.pdf (ISL, Figure 10.12) [Comparison of average, complete (max), and single (min) linkages. Observe that the complete linkage gives the best-balanced dendrogram, whereas the single linkage gives a very unbalanced dendrogram that is sensitive to outliers (especially near the top of the dendrogram).]

[Probably the worst of these is the single linkage, because it's very sensitive to outliers. Notice that if you cut this example into three clusters, two of them have only one node. It also tends to give you a very unbalanced tree.]

[The complete linkage tends to be the best balanced, because when a cluster gets large, the furthest point in the cluster is always far away. So large clusters are more resistant to growth than small ones. If balanced clusters are your goal, this is your best choice.]

[In most cases you probably want the average or complete linkage.]

Warning: centroid linkage can cause inversions where a parent cluster is fused at a lower height than its children.

[So statisticians don't like it, but nevertheless, centroid linkage is popular in genomics.]

[As a final note, all the clustering algorithms we've studied so far are unstable, in the sense that deleting a few input points can sometimes give you very different results. But these unstable heuristics are still the most commonly used clustering algorithms. And it's not clear to me whether a truly stable clustering algorithm is even possible.]