21 The Singular Value Decomposition; Clustering

THE SINGULAR VALUE DECOMPOSITION (SVD) [and its Application to PCA]

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

[Earlier this semester, we talked about the eigendecomposition of a square, symmetric matrix. Unfortunately, nonsymmetric matrices don't have nice eigendecompositions, 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: Every $X \in \mathbb{R}^{n\times d}$ has a singular value decomposition $X = UDV^\top$ of the form

$$X = U D V^\top = \sum_{i=1}^{\min\{n,d\}} \underbrace{\delta_i u_i v_i^\top}_{\text{rank 1 outer product matrix}}$$

D: diagonal

$$= \begin{bmatrix} & \\ & \\ & \end{bmatrix} = \begin{bmatrix} \uparrow & & \uparrow \\ u_1 & \cdots & u_n \\ \downarrow & & \downarrow \end{bmatrix} \begin{bmatrix} \delta_1 & & 0 \\ & \delta_2 & \\ & \ddots & \\ 0 & & \delta_d \\ 0 & & 0 \end{bmatrix} \begin{bmatrix} \leftarrow & v_1 & \rightarrow \\ & \vdots & \\ \leftarrow & v_d & \rightarrow \end{bmatrix}$$

$n \times d$ $\quad$ $n \times n$ $\quad$ $n \times d$ $\quad$ $d \times d$

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

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

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

Diagonal entries $\delta_1, \ldots, \delta_{\min\{n,d\}}$ of D are nonnegative singular values of X.

[By convention, singular values are never negative, but some of them might be zero.]

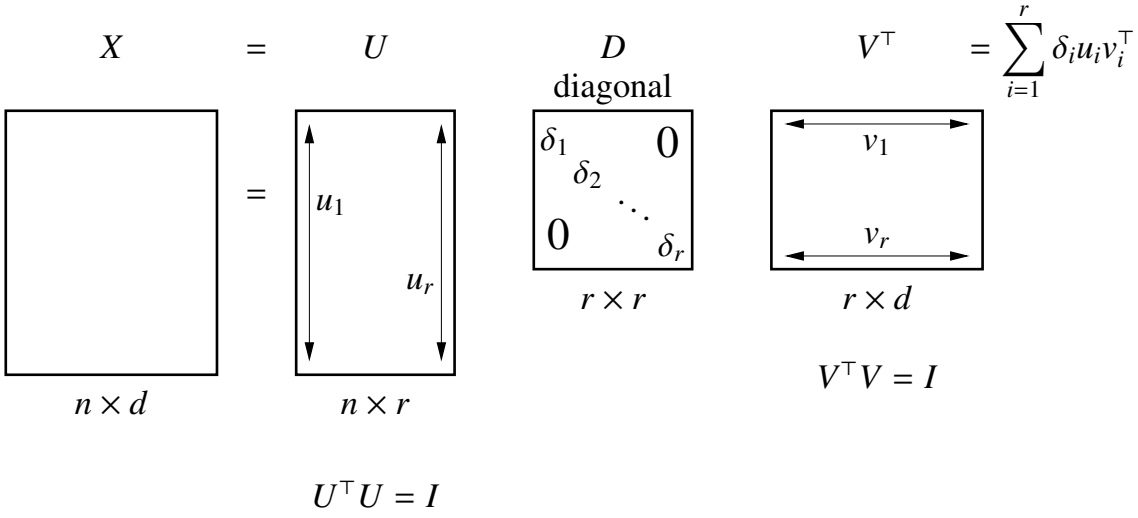
Fact: v_i is an eigenvector of $X^\top X$ w/eigenvalue δ_i^2, so the SVD solves PCA.
Proof: $X^\top X = VD^\top U^\top UDV^\top = VD^\top DV^\top$
which is an eigendecomposition of $X^\top X$, with each δ_i^2 on the diagonal of $D^\top D$.

[The columns of V are the eigenvectors of $X^\top X$, which are the principal components we need for PCA. The SVD also tells us their eigenvalues, which are the squares of the singular values of X. That's related to why the SVD is more numerically stable: the ratios between singular values are smaller than the ratios between eigenvalues, so it's easier to stably compute the singular values of X than the eigenvalues of $X^\top X$.]

Important: Row i of UD gives the principal coordinates of sample point X_i (i.e., $\forall i, \forall j, X_i \cdot v_j = \delta_j U_{ij}$).
[So we don't need to explicitly compute the inner products $X_i \cdot v_j$; the SVD has already done it for us.]
Proof: $XV = UDV^\top V = UD$, so $(XV)_{ij} = (UD)_{ij}$.

The Compact SVD

Singular vectors with singular value zero are useless for PCA [and many other applications], motivating the compact SVD. Let r be the rank of X.



[Draw this by hand. compsvd.pdf]

$\delta_1, \ldots, \delta_r$ are nonzero singular values of the full SVD, & have the same left/right singular vectors.

[There are r nonzero singular values, and we can express X with their singular vectors alone. If X is a centered design matrix for sample points that all lie on a line, then X has rank 1 and there is only one nonzero singular value. If the centered sample points span a subspace of dimension r, X has rank r and there are r nonzero singular values.]

[We might save a fair amount of time by not computing the left and right singular vectors with singular value zero. Observe that the columns of U are still orthogonal, but it is no longer true that $UU^\top = I$. The same goes for V.]

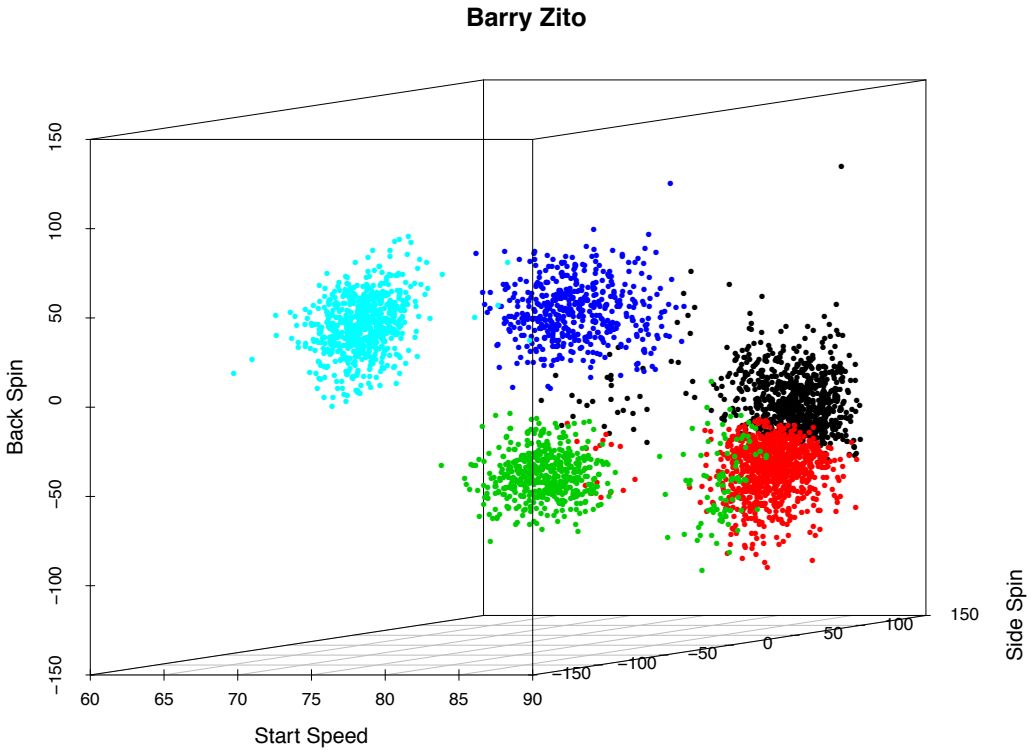
Fact: We can find the k greatest singular values & corresponding vectors in $O(ndk)$ time.
[So we can save time by computing some of the singular vectors without computing all of them.]
[There are approximate, randomized algorithms that are even faster, producing an approximate SVD in $O(nd \log k)$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

[In the next lecture, we will use the compact SVD to help us understand the Moore-Penrose pseudoinverse and its application to least-squares linear regression.]

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 sample 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.]

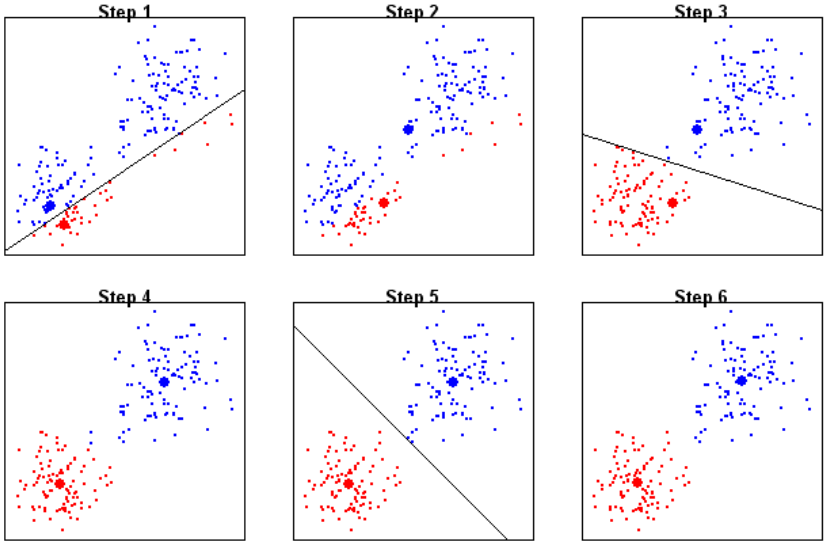
k-means heuristic: Alternate between
(1) y_j's are fixed; update μ_i's
(2) μ_i's are fixed; update y_j's
Halt when step (2) changes no assignments.

[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.]

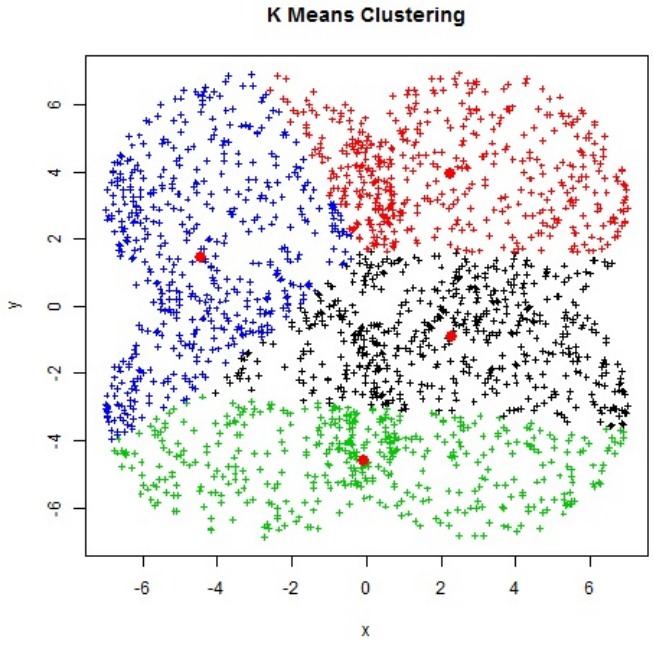
Step (1): One can show (calculus) the optimal μ_i is the mean of the points in cluster i.
[This is easy calculus, so I leave it as a short exercise.]

Step (2): The optimal y assigns each point X_j to the closest mean μ_i.
[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.]

[. . . so both steps minimize the cost function, but they don't optimize all the variables at once.]

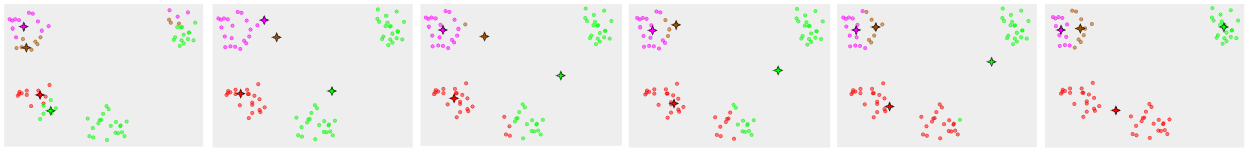


2means.png [An example of 2-means. Odd-numbered steps reassign the data points. Even-numbered steps compute new means.]



4meansanimation.gif [This is an animated GIF of 4-means with many points. Unfortunately, the animation doesn't work 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 says that Lloyd's algorithm never loops forever. But it 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.]



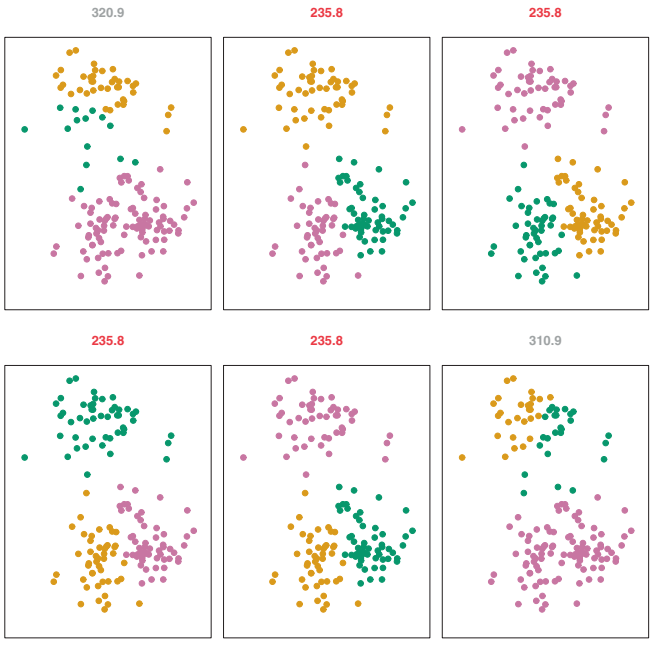
4meansbad.png [An example where 4-means clustering fails.]

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).
- k-means++: like Forgy, but biased distribution.
  [Each μ_i is chosen with a preference for points far from previous μ_i's.]

[k-means++ is a little more work, but it works well in practice and theory. Forgy seems to be better than random partition, 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, each time starting with a different random partition. 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 $\displaystyle\sum_{i=1}^{k} \frac{1}{n_i} \sum_{y_j=i} \sum_{y_m=i} \|X_j - X_m\|^2$

[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 sample points and the clusters we assign them to. So it's more compelling.]

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.]

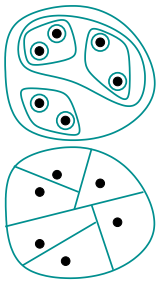
[One difficulty with k-means is that you have to choose the number k of clusters before you start, and there isn't any reliable way to guess how many clusters will best fit the data. The next method, hierarchical clustering, has the advantage in that respect. By the way, there is a whole Wikipedia article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

Top-down, aka divisive clustering:
start with all pts in one cluster; repeatedly split it.



[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.]

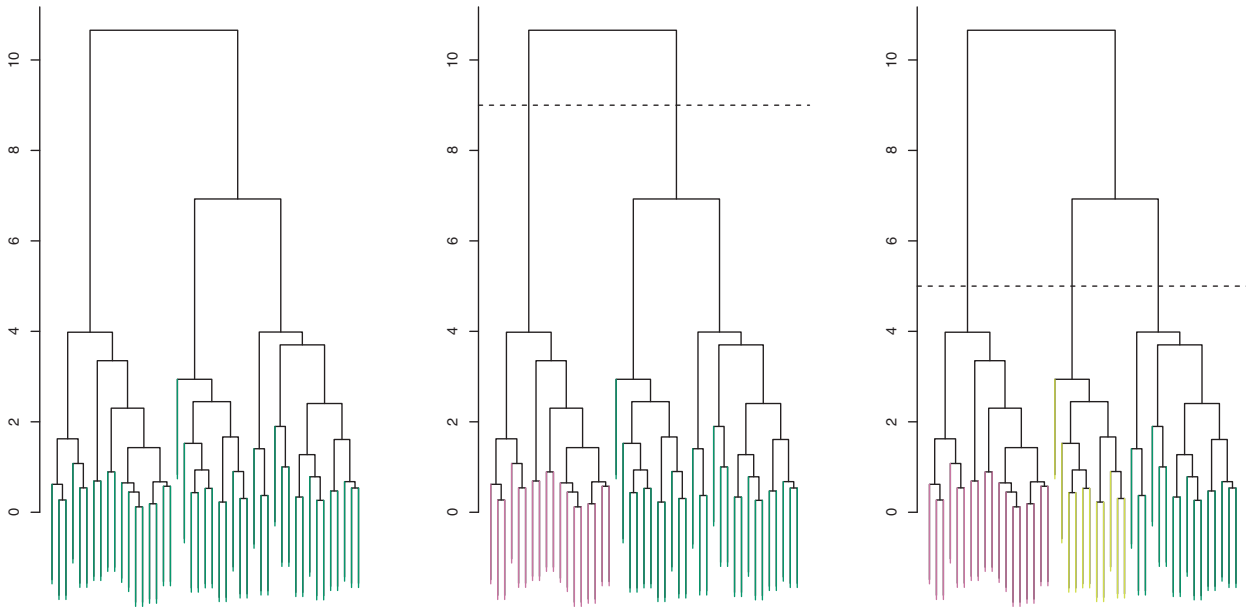
We need a distance fn for clusters A, B:

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 sample points. The centroid linkage only really makes sense if we're using the Euclidean distance.]

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.]

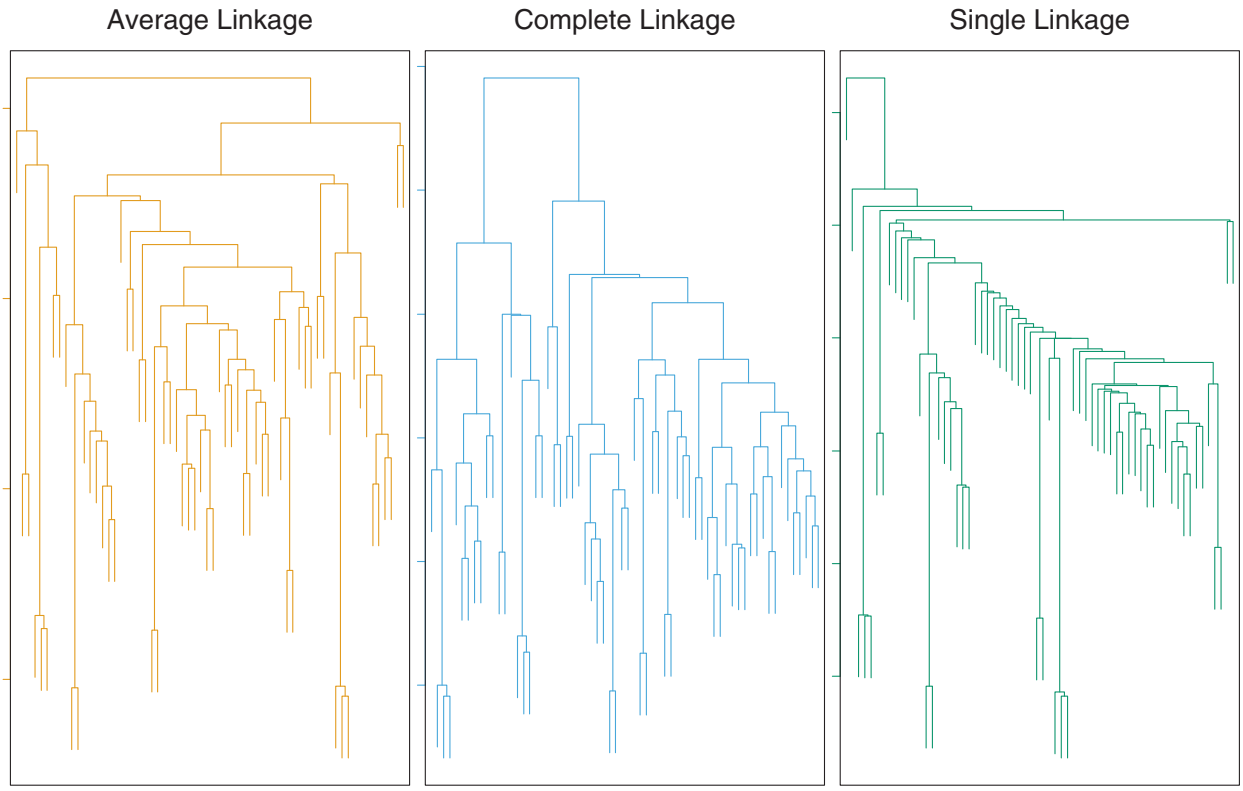
Dendrogram: Illustration of the cluster hierarchy (tree) in which the vertical axis encodes all the linkage distances.



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 treenode's left subtree and right subtree 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 sample point. 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 farthest 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 applications 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 sample 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.]