Partitioning Data into Clusters
Finding Categories in Data

36-350: Data Mining

20 September 2006

Reading: Textbook, sections 9.3–9.5.

Why Cluster?

So far, in our discussion of categorization, we have been assuming that we have some initial examples which are divided into sensible classes. That is, on our **training data**, we not only know what the classes are (automobile/motorcycle, flower/tiger/ocean, etc.), but we know which example is in which class, because the examples have been **labeled** by someone who knew what they were doing. The point of classification methods, like the nearest-neighbor method or the prototype method, is to accurate assign new, unlabeled examples (from **test data**) to these known classes. This involves guessing and extrapolation, but we at least have the labeled training data to start from. (This use of the training labels is why the subject is called **supervised** learning.) The point of calculating information was to select features which made classification easier.

All of this depends on having both known categories and labeled examples of the categories. If there are known categories but no labeled examples, we may be able to do some kind of **query**, **feedback**, **reinforcement** or **semi-supervised** learning, if we can check guesses about category membership — Rocchio's algorithm, from lecture 2, takes feedback from a user and learns to classify search results as "relevant" or "not relevant". But we might *not* have known classes to start with. In these **unsupervised** situation, one thing we can try to do is to *discover* categories which are in implicit in the data themselves. These categories are called **clusters**, rather than "classes", and finding them is the subject of **clustering** or **cluster analysis**. (See Table 1.)

(Even if our data comes to us with class labels attached, it's often wise to be skeptical of their use. Official classification schemes are often the end result of a mix of practical experience; intuition; theory; prejudice; ideas copied from somewhere else; compromises among groups which differ in interests, ideas and clout; and people making stuff up because they need *something* by deadline. Moreover, once a scheme gets established, organizational inertia can keep it in place long after whatever relevance it once had has eroded away. The Census

Known classes?	Class labels	Type of learning problem
Yes	Given for training data	Classification; supervised learning
Yes	Hints/feedback	Semi-supervised learning
No	None	Clustering; unsupervised learning

Table 1: Kinds of learning problems

Bureau set up a classification scheme for different jobs and industries in the 1930s, so that for several decades there was one class of jobs in "electronics", including all of the computer industry. The point being, even when you have what you are *told* is a supervised learning problem with labeled data, it can be worth treating it as unsupervised learning problem. If the clusters you find match the official classes, that's a sign that the official scheme is actually reasonable and relevant; if they disagree, you have to decide whether to trust the official story or your cluster-finding method.)

Good clusters

A good way to start thinking about *how* to cluster our data is to ask ourselves *what* properties we want in clusters. First of all, clusters, like classes, should **partition** the data: every possible object should belong to one, and only one, cluster. Beyond that, it would be good if knowing which cluster an object belonged to told us, by itself, a lot about that object's properties. In other words, we would like the expected information in the cluster about the features to be large. If the features are $X_1, X_2, \ldots X_n$, and the cluster is C, we would ideally maximize

$$I[X_1, X_2, \ldots X_n; C]$$

Actually doing this maximization turns out to be very hard. However, we can say some things about what the maximally-informative clusters would look like, and use these properties to guide our search.

A high information value for the clusters means that knowing the cluster reduces our uncertainty about the features. All else being equal, this means that the objects in a cluster should be *similar to each other*, or form a **compact** set of points in feature-vector space. Again, all else being equal, different clusters should have *different* distributions of features, so clusters should be **separated**. If one of the clusters is much more probable than the others, learning which cluster an object belongs to doesn't reduce uncertainty about its features much, so ideally the clusters should be equally probable, or **balanced**. Finally, we could get a partition which was compact, separated and balanced by saying each object was a cluster of one, but that would be silly, because we want the partition to be **parsimonious**, with many fewer clusters than objects.

There are many algorithms which try to find clusters which are compact, separated, balanced and parsimonious. Parsimony and balance are pretty easy to quantify; measuring compactness and separation depends on having a good measure of distance for our data to start with. (Fortunately, similarity search has taught us a lot about distance!) We'll look first at one of the classical clustering algorithms, and try to see how it achieves all these goals.

The k-means algorithm

Recall that in the prototype method, we took the prototype for each class to be its average or mean, and assigned new points to the class with the closest prototype. The **k-means algorithm** is an unsupervised relative of the prototype method for clustering, rather than classification.

1. Guess the number of clusters, k
2. Guess the location of cluster means
3. Assign each point to the nearest mean
4. Re-compute the mean for each cluster
5. If the means are unchanged, exit; otherwise go back to (3)

The **objective function** for k-means, what it "wants" to minimize, is the **sum-of-squares** for the clusters:

$$\begin{aligned} SS &\equiv \sum_{C}\sum_{i \in C} \|x_i - m_C\|^2 \\ m_C &\equiv \frac{1}{n_C}\sum_{i \in C} x_i \end{aligned}$$

m_C is the mean for cluster C, and n_C is the number of points in that cluster.

The **within-cluster variance** for cluster C is

$$V_C \equiv \frac{1}{n_C}\sum_{i \in C} \|x_i - m_C\|^2$$

so

$$SS = \sum_{C} n_C V_c$$

In words, the sum of squares is the within-cluster variance times the cluster size, summed over clusters. If each cluster is compact, they will have a small within-cluster variance, so V_C and SS will be small, so this objective function favors compactness. It also favors balance, because big clusters are more "costly" than small ones of equal variance.

Each step of k-means reduces the sum-of-squares. The sum-of-squares is always positive. Therefore k-means must eventually stop, no matter where it was started. However, it may not stop at the best solution.

K-means is a **local search** algorithm: it makes small changes to the solution that improve the objective. This sort of search strategy can get stuck in **local**

minima, where the no improvement is possible by making small changes, but the objective function is still not optimized.

It's often helpful to think of this in terms of a **search landscape**, where the height of the landscape corresponds to how good a solution the algorithm has found. (So *minimizing* the objective function is the same as *maximizing* the height on the landscape.) Local search is also called **hill climbing**, because it's like a short-sighted climber who tries to get to the top by always going uphill. If the landscape rises smoothly to a central peak, this will get to that peak. But if there are local peaks, it can get stuck at one, and which one it reaches depends on where the climb starts.

For k-means, the different starting positions correspond to different initial guesses about the cluster centers. Changing those initial guesses will change the output of the algorithm. These are typically randomized, either as k random data points, or by randomly assigning points to clusters and then computing the means. Different runs of k-means will thus generally give different clusters, but you can actually make use of this: if some points end up clustered together in many different runs, that's a good sign that they really do belong together.

Hierarchical clustering

k means is what's sometimes called a *simple* or *flat* partitioning algorithm, because it just gives us a single set of clusters, with no particular organization or structure within them. But it could easily be the case that some clusters could, themselves, be closely related to other clusters, and more distantly related to others. (If we are clustering images, we might want not just to have a cluster of flowers, but roses and marigolds within that. Or, if we're clustering patient medical records, we might want "respiratory complaints" as a super-cluster of "pneumonia", "influenza", "SARS", "sniffles".) So sometimes we want a **hierarchical** clustering, which is depicted by a **tree** or **dendrogram**.

War'ds method

Ward's method is another algorithm for finding a partition with small sum of squares. Instead of starting with a large sum of squares and reducing it, you start with a small sum of squares (by using lots of clusters) and then increasing it.

1. Start with each point in a cluster by itself (sum of squares = 0).
2. Merge two clusters, in order to produce the smallest increase in the sum of squares (the smallest merging cost).
3. Keep merging until you've reached k clusters.

The **merging cost** is the increase in sum of squares when you merge two clusters (A and B, say), and has a simple formula:

$$\begin{aligned}\Delta(A,B) &= \sum_{i \in A\cup B} \|x_i - m_{A\cup B}\|^2 - \sum_{i\in A}\|x_i - m_A\|^2 - \sum_{i \in C}\|x_i - m_B\|^2 \\ &= \frac{n_A n_B}{n_A + n_B}\|m_A - m_B\|^2\end{aligned}$$

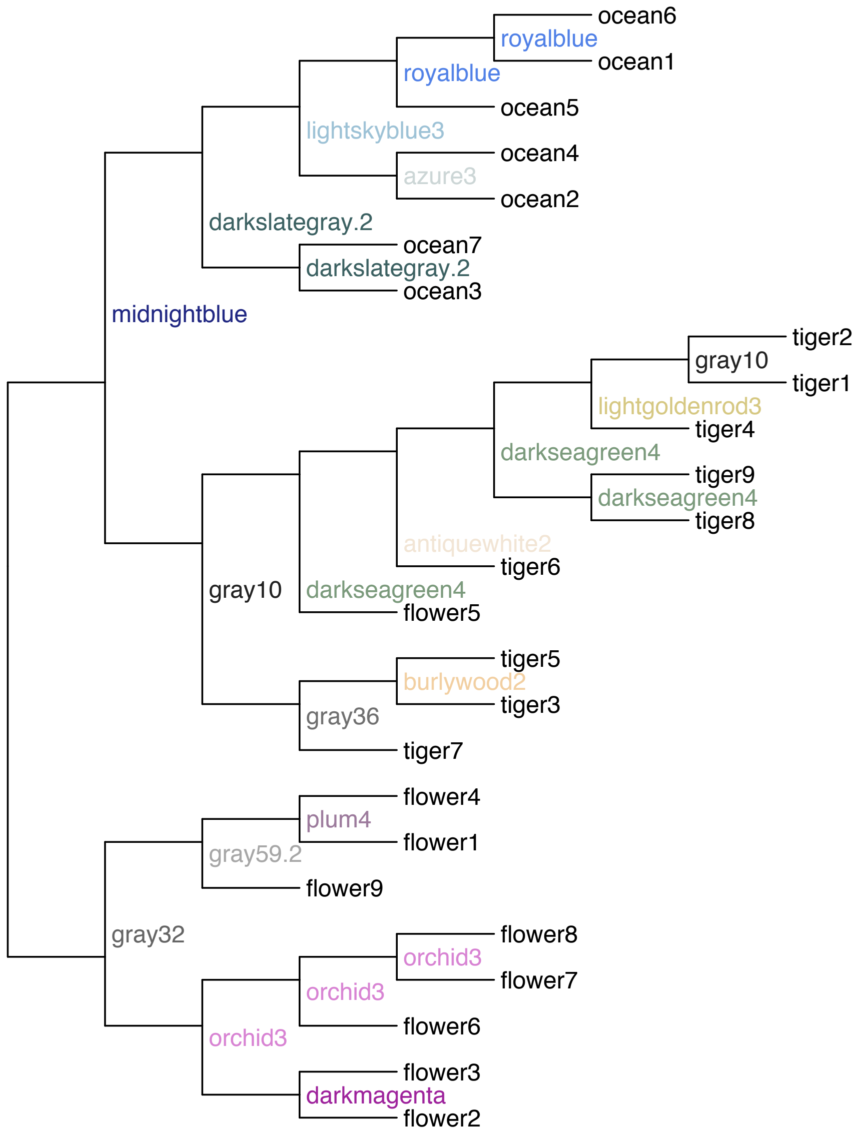
(You may recall having seen a similar formula for t-tests of the difference in means.) Note that Ward's method does not rely on a random starting guess, so its answer is unique.

The Δ formula tells us that there is a trade-off between separation and balance. If clusters are equally far apart (separated), it's better to merge the smaller ones. This means that Ward's algorithm will sometimes merge clusters which are further apart but smaller.

The k-means algorithm gives no guidance about what k should be. Ward's algorithm, on the other hand, can give us a hint through the merging cost. If the cost of merging increases a lot, it's probably going too far, so a reasonable rule of thumb is to keep reducing k until the cost jumps, and then use the k right before the jump.

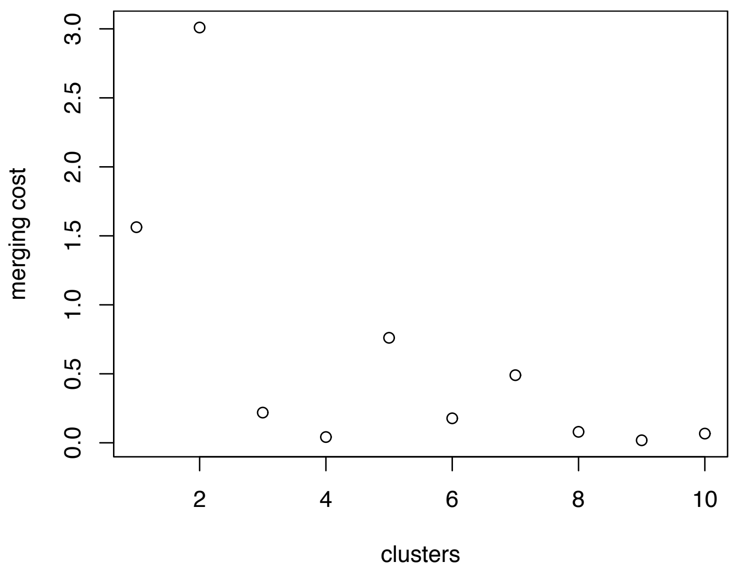
The partitions produced by Ward's method are **nested**: the partition of size k is contained within the partition of size $k + 1$. Ward's method also does not do local search. These two properties mean that Ward's method generally does not produce a sum-of-squares as small as k-means. However, we can run the k-means search starting from the Ward's method solution, to get a competitive sum-of-squares.

Here's what Ward's method does with the flower/tiger/ocean images (represented, as usual, by bags of colors):



Only one mistake is made (flower5). The clustering was computed using all colors, but to enhance interpretation, each cluster has been labeled with a "joining" color (one which both subgroups have in common but is rare in the rest of the data). This is similar to finding colors with high information content.

The merging cost suggests that there are 3 clusters (also 6 or 8):



The sum of squares measures distance equally in all directions, so it wants the clusters to be round. This is not always very sensible (see Figure 1).

Single-link algorithm

Single-link clustering can handle any cluster shape:

1. Start with each point in a cluster by itself (sum of squares = 0).
2. Merge the two clusters with smallest gap (distance between the two closest points)
3. Keep merging until you've reached k clusters.

It's called "single link" because it will merge clusters so long as *any* two points in them are close (i.e., there is one link).

This algorithm only wants separation, and doesn't care about compactness or balance. This can lead to new problems, as shown in Figure 2.

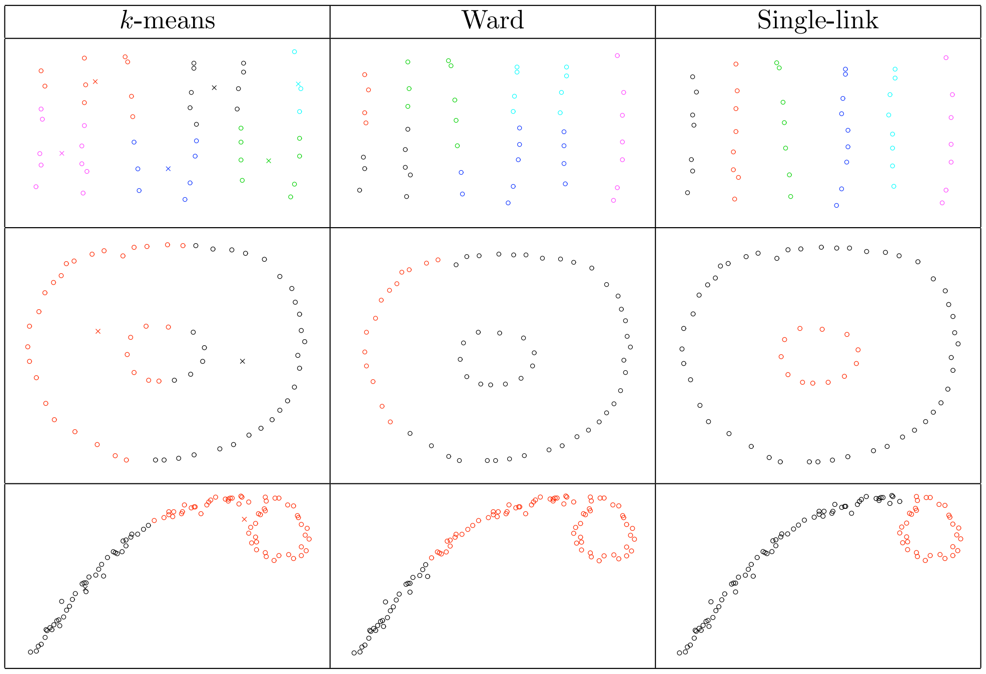


Figure 1: Some clustering problems where the single-link method does better than *k*-means or Ward's method. (In the *k*-means plots, the cluster means are marked $\times$.)

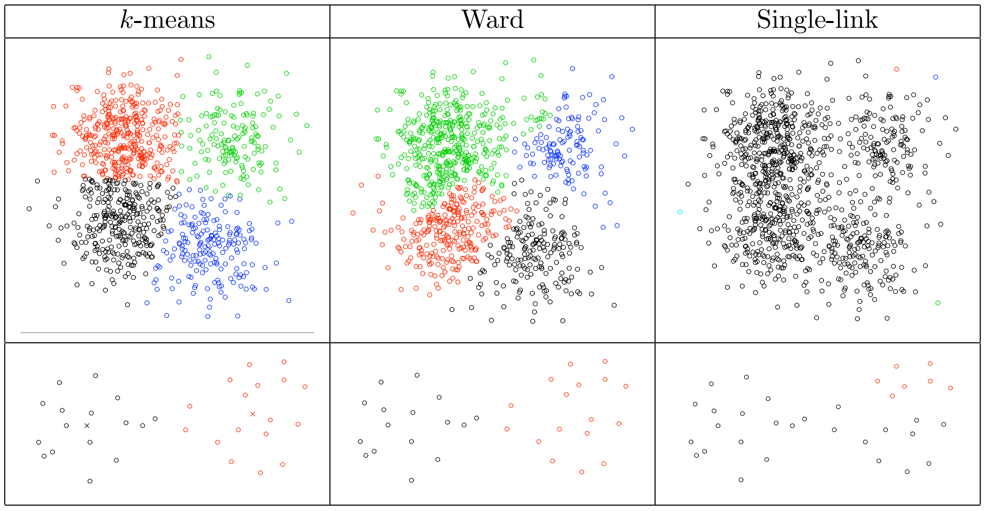


Figure 2: Some cases where *k*-means or Ward's algorithm does better than the single-link method.