Categorizing Data Vectors
Types of Categorization, Basic Classifiers, Finding Simple Clusters in Data

36-350: Data Mining

15 September 2008

READING: Textbook, sections 9.3–9.5.

Categorization and Classifiers

Dividing data into discrete categories is one of the most common kinds of data-mining task. Often the categories are things which are given to us in advance, by some kind of background knowledge (cells: cancerous or not?), or the kind of decision we are going to make (credit applicant: will they pay back the loan or not?), or simply by some taxonomy which our institution has decided to use (text: politics or religion? automobile or motorcycle?; pictures: flower, tiger or ocean?). This case, of assigning new data to pre-existing categories, is called **classification**, and the categories are called **classes**.

When we have some examples, or **training data**, which have been **labeled** by someone who knew what they were doing, we have a **supervised** learning problem. The point of classification methods is to accurately assign new, unlabeled examples, from the **test data**, to these classes. This is "supervised" learning because we can check the performance on the labeled training data. The point of calculating information was to select features which made classification easier.

We have already seen two algorithms for classification:

1. In **prototype classification**, we represent each class by a single vector, its *prototype*, and assign new data to the class whose prototype is closest. This uses little memory or computation time, but implicitly assumes that each class forms a compact (in fact, convex) region in the feature space.

2. In **nearest neighbor** classification, we assign each new data point to the same class as the closest labeled vector, or **exemplar** (or **example**, to be slightly less fancy). This uses lots of memory (because we need to keep track of many vectors) and time (because we need to calculate lots of distances), but assumes a lot less about the geometry of the classes. In fact, in a sense it assumes *nothing* about the geometry of the classes.

Known classes?	Class labels	Type of learning problem
Yes	Given for training data	Classification; supervised learning
Yes	Given for some but not all training data	Semi-supervised learning
Yes	Hints/feedback	Feedback or reinforcement learning
No	None	Clustering; unsupervised learning

Table 1: Kinds of learning problems

We will come back to trade-offs between methods which make strong assumptions and methods which make weak assumptions, especially later when we look at how to evaluate and compare predictive models.

Why Cluster?

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 learning, if we can check guesses about category membership — Rocchio's algorithm 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 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.