Contents

Chapter 7

Clustering and classification methods

Introduction

- **Supervised vs. unsupervised**. In a supervised method, we are given ”training data” for which we know the proper classification, and ”test data” where we want to infer the proper classification. In an unsupervised method, we are given a bunch of examples and asked to classify them de novo. Examples of unsupervised methods include hierarchical clustering or k-means clustering. An example of a supervised classification method is an artificial neural network.

- **Model-based vs. distance-based classification**. If we know enough about a problem to model it with a probabilistic model, we can use EM or other algorithms to solve classification problems; in our introduction to EM, we saw a simple fair versus biased coin classification problem, for example. Sometimes we don't know enough about the problem to model it effectively, especially in an exploratory phase of first looking at the problem, so we use cruder classification methods based on clustering points that seem to be close by some distance criterion.

- **Hierarchical versus flat classification.** We might classify examples into one and only one of k classes, in which case we're assuming a ”flat”, nonhierarchical data structure. (Pfam, for instance, artifically imposes a flat structure on protein domain annotation.) Alternatively, we can assume hierarchical classification, often in the form of a binary tree structure, in which an example is a member of many nested groups, defined by nodes on the tree.

- **Fixed versus unknown class number.** Some methods, such as k-means, will assume that the number of classes is known a priori. Other methods, such as hierarchical clustering, do not have to make an a priori assumption about the number of clusters.

- **Probabilistic versus absolute classification.** Some methods (especially explicit probabilistic modeling methods, but also including neural nets) allow the classification of an element to be probabilistic, e.g. modeling uncertainty about the classification.

Hierarchical clustering

Algorithm: (see Durbin et al. p. 166):
Input: Distance matrix d_{ij}

- Choose i, j with smallest distance d_{ij}
- Join i, j in binary tree with a node at height $\frac{d_{ij}}{2}$.
- Create a new cluster k by joining clusters i, j.
- Calculate new distances d_{kl} to all remaining clusters l. (matrix is now one row/column smaller.)
- Repeat until all elements are joined.

Choice of pairwise distance calculations

Given two feature vectors x, y with N components $1..i..N$:

- Euclidean: $d_{xy} = \sqrt{\sum_i (x_i - y_i)^2}$
- Manhattan: $d_{xy} = \sum_i |x_i - y_i|$
- Chebyshev: $d_{xy} = \max_i |x_i - y_i|$
- Hamming: $d_{xy} = \sum_i \delta(x_i, y_i)$ $\delta(a, b) = 1$ if $a \neq b$
- for sequences, we often use 1-an optimal alignment of x, y.
- In phylogenetic inference, we take the calculation of distances between sequences much more seriously, imposing probabilistic models to convert observed residue dissimilarity to an evolutionary distance.

Choice of rules for calculating distances between clusters

- **Single linkage clustering**.
  $d_{ij} = \min_{x \in C_i, y \in C_j} d_{xy}$
- **Average linkage clustering**
  $d_{ij} = \frac{1}{|C_i||C_j|} \sum_{x \in C_i, y \in C_j} d_{xy}$
  a.k.a. UPGMA, unweighted pair group method with arithmetic averaging

- **Complete linkage clustering**
  $d_{ij} = \max_{x \in C_i, y \in C_j} d_{xy}$

K-means clustering

K-means is an example of *partitional clustering*, as opposed to hierarchical clustering.

We will seek to split the data points into K clusters $C_1..C_K$. We have to specify K *a priori*. Each cluster i is defined by a *centroid* μ_i, which here simply means the average position of all the points in the cluster.

K-means seeks to minimize a least-squares deviation of the data points from their K assigned centroids:

$$\sum_{i=1}^{K} \sum_{x \in C_i} D(x, \mu_i)^2$$

We have to define a function $D(.,.)$ to calculate a distance between a data point and a centroid. This is often just a Euclidean distance, but others are possible, as in hierarchical clustering.

The K-means algorithm is very simple. It proceeds iteratively. To begin, K centroids are chosen at random (perhaps by assigning the data points to K clusters randomly, and calculating centroids). At each iteration, each point is assigned to the nearest centroid from the previous iteration, and centroids are recalculated. The iteration continues until satisfactory convergence is achieved (usually, until no data points are reassigned in an iteration; but infinite cycles are possible.)

K-means clustering is not guaranteed to find a globally optimal solution. The procedure should be repeated with multiple random starting points.

Multidimensional scaling

Another way to do cluster analysis is not to cluster at all, but to render the data in a convenient visualization that allows us to see the clusters in a high-dimensional data set for ourselves, in our restricted world of two or three dimensions.

In multidimensional scaling (MDS), we are given M data points $x_1..x_i..x_M$. Each point is a vector in an N-dimensional space. We can calculate an $M \times M$ distance matrix D_{ij} of the "real" distances between all the points in this N-space. We seek a K-dimensional representation (where $K < N$, and K normally is 2 or, more rarely, 3) of points $y_1..y_i..y_M$. We can calculate an $M \times M$ distance matrix d_{ij} of the "new" distances between all the points in the visualizable K-space. The game is to place the points y in such a way that we best approximate the original pairwise distances D_{ij}.

For instance, a simple MDS algorithm is to choose y_i's that minimize the squared difference between the real D_{ij} and the new d_{ij}, summed over all i, j. This objective function is differentiable, so we can optimize it by standard steepest descents or conjugate gradients methods.

There are a lot of possible nuances to MDS, including the choice of distance functions, and the choice of objective function. For instance, by playing with the exponent (making it a cube instead of a square, for instance), one can force long distances to be satisfied at the expense of closer distances. One can also play with the optimization procedure in various ways, for instance by fitting larger distances first, and progressively fitting closer distances as the y's congeal toward a solution.

Supervised classification with neural networks

- "definition": neurons and connections, where the state of a neuron is a function of the state of its connected neighbors and connection weights.

- A McCulloch/Pitts neuron is either on (1) or off (0):

$$n_i(t+1) = F(\sum_j (w_{ij} n_j(t)) - \mu_i)$$

  where $n_i(t+1)$ is the state of neuron i at time $t+1$; w_{ij} are the connection weights between neuron i and other neurons j; μ_i is a "threshold" term that determines the input level at which neuron i will fire; and $F(.)$ is a step function, $F(x) = 1$ if $x > 0$, $F(x) = 0$ if $x \leq 0$.

- You can build a universal computing device (Turing machine) out of such units.

- Can generalize to continuous function $g(x)$ of the summed inputs instead of $F(x)$: sometimes called a "squashing function".

- Finding appropriate weights = "learning", or "training" the network. Analogies to biology are strained, but interesting.

- Various types of ANNs have been built from this basic concept. A terrific introductory example of the use of Hopfield networks for associative memory is in chapter 2 of Hertz/Krogh/Palmer.

- We'll focus, though, on a specific type of network: *layered feedforward networks*, a.k.a. *perceptrons*, introduced by Rosenblatt.

Perceptrons

- Simple (single layered) perceptron has input neurons and output neurons. (The input layer isn't counted as a layer, since it doesn't compute anything, hence "single layered" perceptron.)

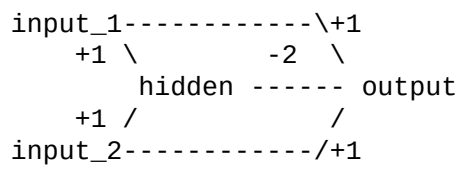
- Can classify any "linearly separable" problem.

- Example of what a simple perceptron can't do: the XOR problem:

input 1	input 2	output
1	1	0
1	0	1
0	1	1
0	0	0

- multilayered perceptron (MLP) one or more "hidden" layers, in addition to the input and output layer.

- Example of how an MLP can compute XOR:



  w/ threshold 1.5 on hidden unit, and threshold 0.5 on output unit. Intuition: hidden unit is an "AND" that switches on when both inputs are on, and suppresses the output unit.

- A simple learning rule (the "delta rule"): iterative bumping of the weights, according to the magnitude of the output error and the magnitude of the input:

$$\Delta w_{ji} = -\delta_i I_j$$

  where δ_i is the difference between the desired (target) output t_i and observed output O_i for output neuron i; I_j is the value on input neuron j.

- For classification problems, outputs typically represent the classes.
  - A *softmax classifier* interprets the outputs as posterior probabilities.

- Minsky and Papert (1969) showed that simple perceptrons couldn't solve XOR and other problems. The Minsky/Papert paper was apparently devastating to the perceptron field, for reasons I don't fully understand (and probably make for a good study of the sociology of science).

- More complex (multilayered) perceptrons can solve XOR but learning algorithms for multilayered perceptrons didn't exist at the time.

Backpropagation: the main MLP learning algorithm

- The delta rule is just steepest gradient descent on the weights.

- Reminder 1: gradient descent optimization, relation to partial differentiation.

- Reminder 2: chain rule in differential calculus.

- objective function to minimize: least squares error.

$$E = 0.5 \sum_i (t_i - O_i)^2$$

where t_i is the target output for neuron i, and O_i is the observed output for neuron i. Remember that O_i is a function of the inputs:

$$O_i = \sum_j w_{ji} I_j$$

- First partial derivative w.r.t. parameter w_{ji}:

$$= -(t_i - O_i) I_j = -\delta_i I_j$$

- That is: backprop is easy to derive from first principles, you're just minimizing an error function by gradient descent.

- Example: write error function of a two-layered MLP as a function of all weights.

$$\begin{aligned} E &= 0.5 \sum_i [t_i - O_i]^2 \\ &= 0.5 \sum_i [t_i - (\sum_j W_{ji} V j)]^2 \\ &= 0.5 \sum_i [t_i - (\sum_j W_j i (\sum_k w_k j I_k))]^2 \end{aligned}$$

- w.r.t $W_j i$ (hidden layer to output layer), derivative looks like simple perceptron:

$$= -\delta_i V_j$$

- w.r.t. $w_k j$ (input to hidden layer):

$$= -\sum_i (t_i - O_i) W_{ji} I_k$$

The $\sum_i (t_i - O_i) W_{ji}$ term is interpreted as a backpropagation of the error $(t_i - O_i)$ in the i layer to the hidden units in the j layer, weighted by W_{ji}.

- This holds up recursively for more complex MLPs with more layers.

- Two properties of interest. First, the backprop algorithm is *local*: corrections to the weight of a neuron only depends on information from its neighbors. This is considered to be biologically attractive. Second, the algorithm is recursive and computationally efficient. Calculating N partial derivatives is $O(N^2)$, but backprop is $O(N)$.

- More complex backprop algorithms (momentum, etc.) can be viewed from the standpoint of standard optimization theory (e.g. conjugate gradient descent, etc.) - see Press et al., Numerical Recipes in C.

Some practical issues in training a neural network

- *Generalization*: the ability of a neural network to properly classify data that weren't in the training set.
- Data are typically split into a *training set* and *test set*, so generalization can be measured on the test set.
- In *cross-validation*, the available data are split multiple ways: for example, in a five-fold crossvalidation experiment, the data are split into five equal chunks abcde; five different networks are trained on abcd, acde, abde, abce, and bcde; tested on the 20was left out in each experiment; and the average of the five performances is reported.
- *Overtraining* If you plot generalization versus the number of training iterations, ANNs typically show a peculiar behavior: although their performance on the training set always increases with more iterations, after a certain point, their performance on the test set will start to degrade after a certain point. The intuition is that the network has started to memorize peculiarities specific to the training data. The best way to avoid overtraining is to keep the networks as simple as possible. (There is also a large literature on Bayesian NNs, where priors, "regularizers", are used to reduce overfitting.) However, an easy way to minimize overtraining is to split the available data into three sets, not two: a training set, a validation set (used for *early stopping*, stopping training when performance on the validation set starts degrading), and a test set (used to evaluate generalization.
- This raises a general point about training and testing machine learning methods. The test set can only be used *once*. The moment you peek at test results and react to them, improving and tweaking your algorithm, you've effectivly used the test set for training, and you can no longer use it fairly to predict generalization.
- Another issue, particularly important in biological applications: we are often interested in training on some data and measuring our ability to generalize to *non-homologous* data. For example, if I train a genefinder, I want to know how well it will predict new genes. But if my available data include all known genes, and I split the data *randomly* into cross-validation training and test sets, odds are very good that I'll have many homologous sequences in the test and training sets. But I don't want to be measuring the ability of my algorithm to predict the gene structure of a mouse gene that's 100Cross-validation experiments on non-independent data (like biological sequences) should be done not randomly, but by a procedure that minimizes the nonindependence: for example, by clustering the data and choosing different clusters for training and testing.

Additional reading

For a practical introduction to the use of neural networks, see *Introduction to the Theory of Neural Computation*, by Hertz, Krogh, and Palmer [2]. Anderson and Rosenfeld

have compiled a collection of classic papers in the field of "neurocomputing" [1]. On the Internet, the http://www-2.cs.cmu.edu/Groups/AI/html/faqs/ai/neural/faq.html collects together a great deal of useful information.

Bibliography

[1] J. A. Anderson and E. Rosenfeld, editors. *Neurocomputing: Foundations of Research*. MIT Press, 1989.

[2] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, Reading, Massachusetts, 1991.