

# Machine learning for busy computational biologists

Miler T. Lee

11 May 2005

## 1 Preliminaries

The crux of machine learning is to assign a “property” (or properties) to an object based on a set of measurable “features” that the object has. The simplest formulation would be a 0/1 categorical property (the object is either in category 0 or category 1) based on similarly binary features (has or does not have feature  $i$  for some  $i$ ). Both the property and the features can be arbitrary, but obviously these will be meaningfully defined if it is a well-structured machine learning problem. The particular property of interest will be motivated by the biological problem; the features ususally by intuition. In all likelihood, not all the features will be equally informative, but you won’t know this ahead of time.

### 1.1 Supervised vs unsupervised learning

The majority of this document will handle so-called supervised learning – that is, learning tasks that consist of “training” the learning system on instances where you know ahead of time what property they possess (these are often called “labeled” instances because they have already been labeled according to the category they belong to). Unsupervised learning dispenses with the prelabeling, and can be thought of as discovering the “natural” categories that the objects fall into; clustering is a good example of this.

### 1.2 Generative vs discriminatory models

Some people use this distinction to contrast techniques such as hidden Markov models (generative) to techniques such as support vector machines (discriminatory). A generative model is based on being able to “construct” the object and in that way determine whether it belongs in a particular category of objects. If we take HMMs as an example, each of the states represent atomic units, say motifs, and an object, say a nucleotide sequence, is made up of a series of such units. The ability for an object to be legally constructed from a series of states at high probability is the criterion for assigning category in the HMM. So as you can tell, HMMs are particularly suited for objects that can be decomposed into a series of informative units. In contrast, a discriminatory model views objects solely as a collection of features that have no inherent ordering or constructive property (this does not necessarily mean that they’re not suited for sequence objects though). All of the techniques discussed here can be considered discriminatory models.

### 1.3 Geometric intuition

Imagine your objects are points represented by vectors based on your feature values. So in two dimensions, your objects are an  $x,y$  coordinate pair ( $x$  is feature 1,  $y$  is feature 2) and thus lie somewhere on the  $x-y$  plane. If your data are well-separable, it should be possible to draw a line somewhere on the graph, such that points lying on one side of the line correspond to category 0, and points lying on the other side of the line correspond to category 1. The goal of machine learning is to determine this line. Once you have the separating line defined, you can classify new objects by graphing them and seeing where they are relative to the separating line. This is easy to extend to more than 2 features – each axis represents the range of one feature, so if you have  $d$  features your graph is in  $d$ -dimensional space, and you will be seeking a separating hyperplane. Similarly, if you have more than two categories, you can

have more than one separating hyperplane, the goal being to divide up the space into as many parts as you have categories.

One caveat worth noting is that throughout this discussion we are assuming that the coordinate space defined by the features has reasonable Euclidean properties – that is, a distance between any two points is a meaningful and correct concept. This is not always the case, since biological intuition does not always correspond to metric distance between features. To some extent, you can make sure ahead of time that your feature space makes sense, and if not, change the way you are defining your features – for instance, changing a numerical feature to a binary presence/absence feature.

Finally, data are rarely well-separable. Sometimes this is due to noise or errors in data collection, but more likely it's because it's just not an easy classification problem. To some extent this can be alleviated by choice of machine learning technique – if you can't separate your points using straight lines, can you use a quadratic function? – but most of the time you'll have to accept and account for some degree of error.

## 2 Training supervised learners

Let's say you have a set of labeled data. Traditionally, you would randomly separate these data into two subsets, a training set and a test set. You train your learning algorithm using the training set, and you assess performance based on classification of objects in the test set.

There is not much science involved in deciding how to partition a training set and test set, except that the partition must be random, otherwise you're introducing bias into your learner – remember, the learning algorithm obtains a hypothesis based on the training instances, so if the training instances are skewed in some particular way, it will affect classification accuracy of future objects. Usually the training set is as large as possible, to maximize number of instances seen during the training process, but this must be balanced with leaving enough instances in the test set to meaningfully evaluate performance.

The test set is meant to be an independent assessment of your trained learner. As such, it is a faux pas to, based on the performance of the test set, go back and tweak the learner. Occasionally you will find examples of papers that use the entire data set as a training and test set, usually because there is a very small amount of data to begin with. This is frowned upon, because there is no way to obtain an independent assessment of the learning algorithm – as an extreme example, my learner could simply memorize the exact feature values for each of the training examples, such that when it sees them again during the testing round, it knows exactly how to classify them. This issue is addressed in more detail in the Overfitting section.

### 2.1 Quantifying performance

Performance metrics of the trained learner are usually reported for the test set. They can also be reported for the training set as sort of a litmus test to see if anything was actually learned.

Let's say you're classifying objects as belonging to category X or not belonging to category X. Then there are two types of errors one can make: a false positive occurs when you classify an object as being in category X when in reality it is not in that category; and a false negative, when you classify an object as not being in category X when in reality it actually is (the definitions of true positive and true negative follow naturally from this). You generally want to minimize both sorts of errors, but usually decreasing the number of false positives increases the number of false negatives, and vice versa. For example, you could classify everything as belonging to category X. You would yield zero false negatives, but probably a whole lot of false positives.

In bioinformatics, two performance metrics based on the number of false positives and negatives are reported: sensitivity and specificity. Sensitivity is defined as the number of true positives divided by the number of true positives plus false negatives ( $TP/(TP+FN)$ ). This always confuses me, so the easiest way to think about it is to say that sensitivity is concerned with accuracy only on the test instances that are in category X in real life. That is, the number you got right out of the number of things you should have classified as belonging to category X. Specificity is similarly  $TN/(TN+FP)$ , or the number you got right out of the number of things you should have classified as not belonging to category X. Sensitivity plus specificity will not add up to anything in particular, but each ranges from 0 to 1.

In the computer science literature, you will also see the metrics precision and recall. Recall is identical to sensitivity. Precision is equal to  $TP/(TP+FP)$ , so it's equal to the number correct out of everything you classified to be positive.

It is worth noting that in most cases, accuracy as defined by number of correct divided by total number of test instances is not an appropriate metric to report, unless you have equal numbers of test instances from all of the different categories (and even so, it's a bit dodgy). Note that you can get 99% accuracy from a gene finder that classifies every nucleotide in the human genome as belonging to a non-gene.

## 2.2 Overfitting

In generating a hypothesis (separating hyperplane) for your data based on your training set, you can end up with a more general hypothesis or a more specific hypothesis. It's usually better to end up with the most general hypothesis possible that still explains your data; the alternative would be a hypothesis too specific for your training set, and thus incapable of performing well on unseen test data. This phenomenon is known as overfitting, and is very easy to do if you're not careful.

Overfitting can be subtle. Sometimes it's the realization that you have too many parameters in your model as compared to number of features and amount of training data. It can even result from something as innocent as making a tweak to your learner parameters based on poor performance on a particular class of test examples, and not reevaluating on a new unseen test set. The solution to this would be to partition your data set into three parts – a training set, a validation set, and a test set. The learner can iterate through a training phase on the training set and a test phase on the validation set, allowing modifications to the training parameters based on the test results, until performance (sens and spec) reaches a plateau. Continuing to train past the plateau is a good way to overfit. Then final performance can be assessed on the test set, which consists of totally unseen data; this will indicate how well the learner will do in “real world” situations.

This general validation scheme can be modified so that the training and validation sets are different during each iteration. A particular instantiation of this is n-fold cross validation. For example, 10-fold cross validation divides the training set into 10 equal-sized parts, at any given iteration trains on 9 parts and validates on the remaining part.

## 3 Supervised learning algorithms

The following is a sampling of various machine learning techniques one uses on labeled data. It is not meant to be overly-mathy – for mathematical details, refer to J. Kim's notes from BIOL 536, Mitchell's Machine Learning, Duda & Hart (& Stork)'s Pattern Classification (ignoring the chapter on Bayesian networks, which is wrong), and Hastie et al's The Elements of Statistical Learning.

### 3.1 Linear discriminant analysis

LDA with two categories amounts to assuming a probability distribution on the data points in each of the two classes and defining a mathematical function (the so-called Fisher's discriminant function) on the feature variables, such that the value of the function, given particular feature inputs, defines the class that that datapoint belongs to. For most purposes, the probability distribution is a multivariate Gaussian, and the two distributions have equal covariance matrices but possibly (hopefully) different means.

What this amounts to is taking the ratio of the likelihoods that a particular point belongs to one category's probability distribution or the other, and assigning category based on whether that ratio exceeds a particular threshold. The discriminant function is

$$d(\vec{x}) = (\mu_1 - \mu_2)' \Sigma^{-1} \vec{x} \quad (1)$$

which is the result of taking the log ratio between two multivariate Gaussians with means  $\mu_1$  and  $\mu_2$ . These means are estimated using the sample means from the training data. The covariance matrix is estimated using the pooled sample variance-covariance. The threshold value is given by

$$c = \frac{1}{2} (\mu_1 - \mu_2)' \Sigma^{-1} (\mu_1 + \mu_2) \quad (2)$$

assuming equal prior probability of being from either category.

According to J. Kim, geometrically the discriminant function defines the separating hyperplane between the two distributions, such that error is minimized – in this case, error means the ratio of the sum of squares difference within one class (category) over the sum of square differences between the two classes. Sum of square differences are calculated with respect to the scalar values of the projection of each point to the normal vector defined by the separating hyperplane.

### 3.2 The perceptron and artificial neural networks

When you don't have a probability density function to guide placement of the separating hyperplane (which is most of the time), you can turn to methods that have increasingly fewer assumptions about the structure of the data. Rosenblatt's perceptron is a simple machine that can learn a linear separator by sequential iteration through training data. The artificial neural network is simply a collection of interconnected perceptrons that can therefore create more complex partitions of the feature space.

#### 3.2.1 Perceptrons

The perceptron is simply a box. Inputs go in, an output comes out. Let's say that the output is a binary value, -1 or +1. The inputs could be features for a particular data point. Then the perceptron performs computation on those features and assigns a category based on the output value.

If you think of your input features as comprising a d-dimensional vector in your feature space, then the perceptron stores a d-dimensional weight vector and takes the dot product of the two vectors – i.e.,  $\sum w_i x_i$ . If the result is greater than a particular threshold value, +1 is returned; otherwise, -1 is returned. This formulation of this perceptron function is known as a sign function. For convenience, the threshold value is incorporated into the input vector, so you actually have a d+1 dimensional input, where the d+1th component is 1 and the d+1th weight is the negative threshold. Then you can simply return the sign of the sum.

Training a perceptron amounts to finding optimal weights (and threshold, which is now just a weight). Starting with random weights, you iterate through the training set. For each input vector, if the output is correct, you do nothing. If it is wrong, then you want to nudge the weights in the appropriate direction, which can be accomplished by adding the input vector times the correct sign to the weights.

Geometric intuition: the perceptron takes the dot product of your vector with the weight vector, and reports how similar your vector is to the weight vector. Recall that the dot product of two vectors is 0 when they are orthogonal. Then if your vector has a positive dot product, this means it forms an angle of less than 90 degrees with the weight vector – i.e., your vector is on the “left” side of the separator (in 2 dimensions). Conversely, if your vector has a negative dot product, the angle is greater than 90 degrees and your vector is on the “right” side of the weight vector. Hence, a separating hyperplane.

### 3.2.2 ANNs

You can create layers of perceptrons, such that the outputs of the perceptrons in one layer are the inputs to the perceptrons in the next layer; this is an ANN. What this allows you to do is to partition your feature space into complex patterns of linear separators, so you can have arbitrarily many pieces of your subspace belonging to one category or another. Each additional perceptron adds a linear separator, and where the perceptron is with respect to the architecture of the ANN determines how much of the feature space is further partitioned.

It turns out having perceptrons output only +/-1 introduces undesirable properties when you start chaining perceptrons together. So the sign function, which is not continuously differentiable, is replaced by a sigmoid function, which is. This is essentially an s-shaped curve, which at the limits behaves like the sign function, but close to 0 gradually switches sign. So the output of a perceptron becomes a value in a continuous range, between -0.5 and 0.5 for example (depending on how you formulate it). The final output node is usually still a sign function, since it returns the final category assignment for the input.

Training a multilayer neural network involves an algorithm called back propagation, which is an extension to the training algorithm for the perceptron. Since only the node in the final layer can be verified as being correct or incorrect, this result must be propagated backward through the network so that all of the nodes’ weights can be updated accordingly.

ANNs are suited to complicated classification problems, since they can very easily partition the feature space into very complicated regions. That said, this also makes them very easy to overfit. There is not much science in choosing the number of nodes and number of layers, but fewer is generally better, since that induces a simpler partition of the feature space.

## 3.3 Support vector machines

Given two separable classes of points, a separating hyperplane can be defined in terms of dot products (inner products): define some unit vector  $\phi$ , and a threshold value  $c$ . Then for any point  $\vec{x}_i$ , decide whether it is in class I and class II by the following inequalities

$$\vec{x}_i \in I \quad \text{if} \quad \vec{x}_i \cdot \phi > c \tag{3}$$

$$\vec{x}_i \in II \quad \text{if} \quad \vec{x}_i \cdot \phi < c \tag{4}$$

The separating hyperplane H is thus the set of points  $h$  such that  $h \cdot \phi = c$ .

In general, there are an infinite number of such separating hyperplanes. Ideally you would want the one that is most generalizable to unseen data, so in the absence of any other prior knowledge, you would want the hyperplane equidistant from the “boundary” of each of the two classes of points. This is the crux of an SVM: the support vectors are defined to be the two points nearest the separating hyperplane, such that one point belongs to one class and the other point belongs to the other class. The maximum-margin hyperplane is halfway between these two points. So really, you only care about points that are very close to the hyperplane – the ones that are presumably more difficult to classify – and can safely ignore the ones far away that are clearly in one class or the other. Determining the maximum-margin hyperplane is a quadratic programming optimization problem

In real life, data are not perfectly separable, hence the notion of a soft margin hyperplane, which involves assessing a penalty for misclassified points while still maximizing the distance of the correctly classified points to the hyperplane.

### 3.3.1 The kernel function

In the simplest formulation of an SVM, computation occurs in Euclidean space. However, the strength of SVMs lies in their ability to define points in transformed spaces – colloquially, this means we can bend space if it helps separate the points better. The computation turns out to be exactly the same as the vanilla SVM, except that instead of a dot product we use a non-linear kernel function appropriate to the problem. This obviates the need to do an explicit coordinate transform on the input data. A kernel function has the form  $K(x, y)$ , where  $x$  and  $y$  are elements in your space; for the simple non-transformed Euclidean space,  $K(x, y) = x \cdot y$ . The output is a real (or complex) number.

The justification for using a kernel function is very interesting if you're into linear algebra. I refer you to J. Kim's notes. Suffice it to say, as long as your function satisfies the following properties, it is a valid kernel function:

1. Non-degeneracy: if  $K(x, y) = 0$  for all  $y$ , then  $x$  must be the zero vector
2. Symmetry:  $K(x, y) = K(y, x)$
3. Bilinearity:  $K(x + y, z) = K(x, z) + K(y, z)$  and  $K(\alpha x, y) = \alpha K(x, y)$  for any scalar  $\alpha$
4. Positive-definiteness:  $K(x, y) > 0$  if  $x \neq 0$

It turns out that for any symmetric, positive definite matrix ( $x^T A x > 0$ )  $G$ ,  $K(x, y) = x' G y$  is a valid kernel function for a finite dimensional space. For infinite dimensions, Mercer's theorem applies.

Common kernels used besides the Euclidean dot product are the  $n$ th-order polynomial kernel

$$K(x, y) = (x \cdot y + c)^n \tag{5}$$

where  $n$  is empirically chosen according to the properties of the problem, and the radial kernel

$$K(x, y) = \exp \left\{ -\frac{\|x - y\|^2}{2\sigma^2} \right\} \tag{6}$$

which essentially defines a Gaussian in feature space. The negative exponential dampens the difference between points very far apart.

## 3.4 Decision trees

A decision tree can be thought of as a series of cascading questions that you ask of a data point, and based on the answers to these questions, a category is assigned. It's particularly well suited for discrete-valued features, such as presence/absence of a particular feature, or fuzzy features such as low/medium/high expression levels. In terms of the architecture of a tree, you start at the root node, which asks a question that can have any finite number of responses. The number of possible responses equals the number of descendent nodes the root has. Depending on the answer to that question based on the input, you move on to the appropriate descendent node and answer that question, continuing until you reach a leaf node, which corresponds to a category assignment. At any given tree height, the nodes will not necessarily be asking the same questions.

The challenge in training a decision tree is determining the order in which to ask the questions, including whether some questions are irrelevant to class assignment. One straightforward way to determine an ordering is to always ask the question that partitions the training data into near equal-sized parts. The intuition is that a question is not “informative” if it only distinguishes a couple of data points from the rest – yes, you are closer to assigning a category to those couple, but you have gotten nowhere with the majority of the points. In information theoretic terms, you want to choose the question that has highest entropy. This is the basis for the ID3 algorithm.

### 3.5 k-Nearest neighbors

The kNN algorithm is straightforward. Given a set of labeled points in your feature space, a new point is classified based on the categories of the k nearest points according to some distance measure. Usually k is odd, so the majority of the points nearest to the new point will define the category for that new point. kNN is particularly well suited for feature spaces with ill-defined global structure – e.g., clumpy data – since class assignment is a very local procedure. However, computation is inefficient as compared to other algorithms, since a new point must potentially be compared to every reference point to determine the k nearest neighbors.

## 4 Unsupervised clustering algorithms

### 4.1 k-Means clustering

The goal of k-means is to determine k centroid points in feature space (not necessarily points in your input set) that define k non-overlapping clusters. An input point is assigned to the closest cluster, according to shortest distance to the centroid. Finding near-optimal centroids amounts to randomly picking k points, determining the clusters that result, then recomputing the centroids to be the mean point of each of the clusters; repeat until centroids don’t change. k-means does not usually produce unique centroids – running the algorithm several times with different starting centroids will potentially yield different final clusters. Statistics other than the mean can be used, such as the median.

k is a user-defined parameter and is not necessarily known in advance. k-means will only work if there are meaningful convex clusters in feature space. Imagine a T-shaped set of points, where the horizontal part of the T constitutes one class, while the vertical part a different class.

### 4.2 Hierarchical clustering

Basically, a binary tree of points is created, such that any subtree can be considered a cluster of points that are close together in space. This tree is created by pairwise grouping of points or clusters of points: start by grouping together the two closest points into one point cluster and treat that cluster as a new point; continue until there is only one “point” remaining. Distance between two points is well-defined according to Euclidean distance, for example. The distance between two clusters or between a point and a cluster depends on the type of hierarchical clustering being performed:

- Single linkage: the distance between two clusters is the shortest distance between a point in each cluster
- Complete linkage: the distance between two clusters is the longest distance between a point in each cluster
- Average linkage: the distance is the average of all pairwise distances between points in the clusters
- Centroid: a cluster is defined by a centroid point, and distances are computed with respect to that point