CS 170 Algorithms
Fall 2014 David Wagner

11/24 Notes

Machine learning: the classification problem

Next, we will study machine learning, and specifically, classification using supervised learning. We are given n observations $x_1, \ldots, x_n$; each observation x_i is labeled with a class y_i. Now we are given a new observation x, and the challenge is to predict its class.

Let's take an example: spam filtering. Each observation x_i might be a single email message, and the classification y_i might indicate whether this email is spam or not. We might have a large training set $(x_1, y_1), \ldots, (x_n, y_n)$ of messages that are already labeled as spam or not-spam. (For instance, these might be obtained from a user's saved email messages and their spam folder.) The spam filter can process the training set to look for patterns that might tend to be associated with spam. Now when a new email x arrives, the spam filter's job is to predict whether x is spam or not. The goal of a classification algorithm is to automate this entire process, including looking for patterns in the training set and using this to classify new emails x as they arrive.

Spam filtering is an example of a boolean classification problem. Boolean classification is the special case of the classification problem, where we have only two possible classes (e.g., spam and not-spam).

The classification problem can be modeled as follows. Let $\mathcal{X}$ be the space that observations are drawn from, and $\mathcal{Y}$ the set of classes. We are given n observations labeled with their class, namely, $(x_1, y_1), \ldots, (x_n, y_n)$, where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. Now given $x \in \mathcal{X}$, we want to predict the class $y \in \mathcal{Y}$ of x.

Conceptually, we imagine that there is some function $f : \mathcal{X} \to \mathcal{Y}$ that determines the class of an observation, as a function of the observation itself. In particular, we might assume that every observation in the training set is correctly labeled: in other words, that $f(x_i) = y_i$ for all i. Our goal is to try to learn the function f, or some approximation to it.

Algorithms for the classification problem are extremely useful in practice. They're used widely in industry, e.g., to predict which machines might fail, or which transactions might be fraud, or which customers might be worth extra effort to please, or which patients might be at greatest risk of illness, to name just a few examples. New algorithms—and the availability of richer data sets—have spurred tremendous advances and broad use of these methods in industry.

There are many widely used algorithms for classification: some common ones include naive Bayes classifiers, neural networks (perceptrons), k-nearest neighbor classifiers, support vector machines (SVMs), decision trees, random forests, linear regression, hidden Markov models, and many more. In this class we'll look at just two of these: k-nearest neighbor classifiers and random forests. This only scratches the surface of this exciting area. You can learn more in CS 188 and CS 189.

Features

Typically, $\mathcal{Y}$ is a small set of a few classes. For instance, in boolean classification problems, the set $\mathcal{Y}$ has just two elements: say, $\mathcal{Y} = \{0, 1\}$.

What might the set $\mathcal{X}$ look like? We will explore algorithms that work when $\mathcal{X} = \mathbb{R}^d$, i.e., $\mathcal{X}$ is the set of length-d vectors of real numbers.

Of course, in practical applications, our observations might be anything: emails, people, or almost anything. How do we shoe-horn them into the framework that our algorithms expect? Basically, we need to a way to map each email/person/etc. to a d-dimensional vector. We do this by identifying features. Typically, we select d attributes of each observation, or d things that we can measure about them. Each one is called a feature. Then we map each email/person/etc. to a vector $x \in \mathbb{R}^d$, whose ith coefficient x_i denotes the value of the ith feature. For instance, in spam filtering, the first feature might be the length of the email, the second feature might be the fraction of words that are mis-spelled, the third feature might be the number of times that the word "Cialis" appears in the email, and so on.

This allows us to treat each observation as an element of the d-dimensional space $\mathbb{R}^n$, or in other words, as a vector of d real numbers. This transforms our data into a form that can be effectively exploited by the classification algorithms we'll see next.

A well-chosen set of features can make the difference between success and failure in practice: it can often make more difference than the choice of learning algorithm. Unfortunately, selecting a suitable set of features is an art. It often involves guessing what characteristics of each observation might be helpful at predicting its class, based on domain expertise and knowledge of the application setting. In particular, there is no uniform formula or method for selecting a good set of features.

For this class, we'll assume that you have selected a set of d features and transformed each observation to a d-vector in $\mathbb{R}^d$, so we can take $\mathcal{X} = \mathbb{R}^d$.

The nearest-neighbor classifier

The first classification algorithm we'll look at is the nearest-neighbor classifier. Intuitively, it is based on the idea that if x, x' are close to each other, then probably they'll have the same class.

If $x, x' \in \mathbb{R}^d$ are two length-d vectors, define $||x - x'||$ to be the distance between x and x'. For instance, we might use the Euclidean distance:

$$||x - x'|| = \left(\sum_i (x_i - x'_i)^2 \right)^{1/2}.$$

We can try other distance measure as well, but the Euclidean distance is a reasonable starting point.

Now the nearest-neighbor classifier classifies the observation x using the following rule:

> Let x_i be the nearest observation to x. Classify x with the class y_i.

In other words, out of all the observations in the training set, we find the one that is closest to x;

then, we classify x with the same class as its nearest neighbor in the training set. This rule is based on the intuition that two similar observations will probably belong to the same class.

Formally, given x, we define

$$i^* = \arg\min_i ||x - x_i||,$$

and assign x the class y_{i^*}.

Here is one simple algorithm for the nearest-neighbor classifier:

Classify(x):
1. Set $i^* := 1$.
2. For $i := 2, 3, \ldots, n$:
3. If $||x - x_i|| < ||x - x_{i^*}||$, set $i^* := i$.
4. Return y_{i^*}.

The running time of this algorithm is $\Theta(nd)$, since we do n iterations and each iteration computes the distance between two length-d vectors, which takes $\Theta(d)$ time. Note that we need to spend $\Theta(nd)$ time per new observation that we want to classify. If the training set is large (so n is large), and if we want to classify many observations, this might be a bit slow. We'll see later an alternative algorithm that is more efficient in some cases.

The k-nearest-neighbors classifier

The nearest neighbor classifier can be improved by looking at the k closest observations in the training set.

Given x, we compute the distance from x to each observation in the training set, and then keep the k closest observations. Intuitively, the class of each of them gives a plausible suggestion for the class of x. Therefore, we can treat each as a vote for the class of x. We take these k votes, find which class received the most votes, and classify x with this class. In many applications, this improves the accuracy of the nearest neighbor classifier by "smoothing" out the classifier's predictions.

This is called the k-nearest neighbors classifier, or k-NN for short. You can think of k as a small constant, chosen in advance, e.g., $k = 5$. For boolean classification, it is often convenient to choose k to be odd, to avoid ties.

One simple way to implement the classifier is to compute the distance from x to each observation in the training set, and keep the k closest (e.g., using a binary heap of size k). The running time will be $\Theta(n(d + \lg k))$. For small values of k, this is essentially as fast as the corresponding algorithm for $k = 1$.

Efficient classification and k-d trees

It is possible to use a divide-and-conquer algorithm to speed up classification, if the dimension d is not too large. In particular, we store the observations in a data structure called a k-d tree. (In our case, where we have d dimensions, it might make more sense to call it a d-d tree, but k-d tree is the traditional name.)

A k-d tree is a way to store a set S of points in $\mathbb{R}^d$. It is a binary tree, and, the points in S are stored in the leaves of the tree. Each internal node in the tree splits the space $\mathbb{R}^d$ into two parts; its left child contains all the points that fall in the first part, and the right child contains the points in the second part. To tell whether x is in the tree, we traverse the tree. Each node indicates a particular dimension (say j, where $1 \leq j \leq d$) and a threshold (say t, where $t \in \mathbb{R}$). When we reach that node, we test whether $x_j \leq t$ (whether the jth coefficient of x is at most t), and if yes, we move to the node's left child, otherwise we move to its right child.

For instance, suppose we are in $d = 2$ dimensions, so each x value is a pair of real numbers. The root node might test whether $x_1 \leq 5$. Its left child might test whether $x_2 \leq 7$ and its right child might test whether $x_2 \leq 6$. This divides the plane into four spaces: $\{(x_1,x_2) : x_1 \leq 5, x_2 \leq 7\}$, $\{(x_1,x_2) : x_1 \leq 5, x_2 > 7\}$, $\{(x_1,x_2) : x_1 > 5, x_2 \leq 6\}$, and $\{(x_1,x_2) : x_1 > 5, x_2 > 6\}$.

How do we build a k-d tree? We might cycle through the dimensions as we go down the tree. In other words, each node at depth j might examine the j mod dth coefficient of x. The threshold might be chosen by looking at the median value of this coefficient, for all the points that are stored under this node (in one of the leaves that is a descendant of this node). This builds a k-d tree that is well-balanced: its depth is $O(d \lg n)$. Searching in such a tree can thus be done in $O(d \lg n)$ time.

k-d trees support efficient nearest neighbor queries. Suppose we are given x and want to find its nearest neighbor in S, given a k-d tree for S. How do we do that? The trick is to use divide-and-conquer. Suppose the root node of the k-d tree tests whether $x_1 \leq t$, so it splits S into two sets: $S_l = \{s \in S : s_1 \leq t\}$ and $S_g = \{s \in S : s_1 > t\}$. Suppose $x_1 \leq t$. We start by recursively finding x's nearest neighbor in S_l, say $y \in S_l$. Now depending on how close x lies to the separating line $x_1 \leq t$, we might or might not need to look at anything in S_g. We use the fact that the distance from x to anything in S_g must be at least $|t - x_1|$. There are two cases.

- If $||x - y|| < |t - x_1|$, there is no need to look at anything in S_g (because then the distance from x to anything in S_g will be larger than the distance to y), so we can stop searching immediately.
- If $||x - y|| \geq |t - x_1|$, we recursively find x's nearest neighbor in S_g, say z, and keep whichever is closer to x, y or z.

This idea allows us to prune the set of points in S: rather than computing the distance from x to everything in S, it is often possible to compute the distance from x to only a small fraction of the points in S.

Therefore, k-d trees allow us to speed up classification with nearest-neighbors classifiers. We store the observations in the training set into a k-d tree (using $S = \{x_1, \ldots, x_n\}$, where $x_1, \ldots, x_n$ are the observations) and use it to efficiently find the nearest neighbor to any new observation x. The divide-and-conquer algorithm sketched above can be generalized to allow finding the k nearest neighbors.

In practice, k-d trees are often significantly faster than computing the distance to everything in the training set if the number of dimensions is small enough ($d \leq 20$ or so), but when d is too large, they might not offer any benefit.

Practical considerations with k-NN classifiers

Suppose we want to build a model to predict a man's shoe size (size 7, 8, 9, 10, 11, or 12) from their height and weight. This sounds reasonable—but one challenge is that, because the weight has a much larger range of values than height, the distance between two observations will be dominated numerically by the difference between their weights. In other words, the narrow range of heights will force the height to become essentially irrelevant, due to how the Euclidean distance works. This is undesirable.

A simple technique to avoid this problem is to normalize all features to have the same range of values (e.g., $[0,1]$) before applying k-nearest neighbors. A slightly more sophisticated approach that is commonly used is to standardize each of the features: we replace the feature value v_i (for the ith feature) with $(v_i - \mu)/\sigma$, where μ is the mean value of the ith feature (among the training set) and σ is the standard deviation of the ith feature (among the training set). This also helps normalize the features, in a way that is a bit less sensitive to outliers.

In general, k-nearest neighbors will tend to perform worse if the dimension d is too large. Therefore, it is often to your advantage to avoid adding too many features.