CS 170 Algorithms
Fall 2014 David Wagner

11/26 Notes

Random forests

Next, we'll look at random forests, another machine learning algorithm for the classification problem. The random forest classifier is a collection of decision trees—we'll see in a moment what that means. Its primary benefits are (a) empirically, this approach often seems to work reasonably well on a broad variety of applications, (b) it is well-suited to efficient implementation.

We'll focus on classification where all our features are continuous variables (real numbers). In other words, each observation is a point in $\mathbb{R}^d$, where d is the number of features. Let $\mathcal{Y}$ denote the set of classes (e.g., $\mathcal{Y} = \{\text{spam}, \text{not-spam}\}$, for spam classification).

Decision trees

A decision tree is a binary tree that represents a function $\mathbb{R}^d \to \mathcal{Y}$. In particular:

- Each leaf is labeled with a decision: a class $y \in \mathcal{Y}$.
- Each internal node is labeled with a binary condition: a coordinate i and a threshold t. If we have an observation $x \in \mathbb{R}^d$, and if we reach this node, then if $x_i \leq t$, we'll go left (to this node's left child), otherwise we'll go right (to this node's right child). Thus, the binary condition splits the set of observations into two sets.

We think of the decision tree as representing an iterative decision process. Given a point $x \in \mathbb{R}^d$, we start at the root of the tree. We check whether the binary condition at the root is true or false, and use that to go left or right to its left/right child. We repeat this until we reach a leaf. The value y at the leaf reached in this way is the value of the function on input x.

Decision tree inference

Suppose we have a training set S of observations labeled with their classes. We'd like an algorithm to build a decision tree that agrees with most or all of the observations in the training set. How do we do it?

In practice, we typically want to put a limit on the size of the decision tree, to avoid *overfitting* (discussed later). Therefore, the real goal is to find a decision tree that is "not too large" and that agrees with as much of the training set as possible. Here "not too large" might be defined in terms of the depth of the decision tree.

Unfortunately, finding an optimal decision tree is NP-hard, and there are exponentially many possible decision trees, so we can't try them all. Therefore, we will use a heuristic: an algorithm that often works well in practice, even though we can't prove anything about it.

In particular, we will use a greedy algorithm. Our greedy algorithm tries all possible binary conditions that could appear at the root and selects the one that seems best. Then, it uses this condition to split the training set S into S_L and S_R, where S_L are the training samples that are sent to the left of the root and S_R are the training samples that are sent to the right, and it recursively applies the same greedy strategy to both of those sets. Thus, we build a decision tree in a top-down fashion, selecting at each node the binary condition that maximizes some measure of how good it is.

What goodness measure shall we use? Intuitively, we are hoping to find a binary condition that splits S into S_L and S_R, where the class of most observations in S_L is the same, and the class of most observations in S_R is the same. Indeed, the best possible case would be where everything in S_L has the same class, and everything in S_R has the same class—then we could make both children of the root be a leaf, and our tree-building process would terminate. In practice, usually we can't achieve a perfect split like that, but our goodness measure should try to measure how close to that we are. So, we'll use

$$\text{goodness}(\text{condition}) = \text{impurity}(S) - \left[\frac{|S_L|}{|S|} \times \text{impurity}(S_L) + \frac{|S_R|}{|S|} \times \text{impurity}(S_R)\right],$$

where $\text{impurity}(S)$ is some measure of how much diversity there is in the classes of the observations in S. The term in the square brackets is a weighted average, weighted by the number of observations that are classed in S_L or S_R; it is the expected impurity, for a randomly chosen observation from S. Thus, our goodness measure quantifies how much improvement we've made: how much closer we've got to a perfect split.

All that remains is to select an impurity metric. What shall we use? There are several standard impurity metrics that are commonly used in practice, but fortunately, they all seem to perform about equally well, so you can use any one of them. We'll use the *entropy* of the classes as our impurity metric. The corresponding goodness metric is called the *information gain* metric. Thus, for each possible binary condition, we'll calculate its information gain, and choose the condition whose information gain is highest.

The information gain is defined in terms of entropy, so we first need to define the entropy of a probability distribution. Suppose we have a random variable Y that takes on values in some set $\mathcal{Y}$. Then its entropy is defined to be

$$H(Y) = -\sum_{y \in \mathcal{Y}} \Pr[Y = y] \lg \Pr[Y = y].$$

(To make sure the expression above is well-defined, the sum is taken over all $y \in \mathcal{Y}$ such that $\Pr[Y = y] \neq 0$. Or, another way to put it is that we define $0 \lg 0$ to be 0.) We define $\text{impurity}(S)$ as $H(Y)$, where Y is a random variable that indicates the class of a randomly chosen observation (choose an observation uniformly at random from S, then let Y be its class). Note that $\Pr[Y = y]$ is the fraction of observations in S that take on the class y.

The information gain metric is then

$$\text{goodness}(\text{condition}) = H(Y) - \left[\frac{|S_L|}{|S|} \times H(Y_L) + \frac{|S_R|}{|S|} \times H(Y_R)\right],$$

where S_L, S_R are the results of partitioning S using the binary condition mentioned above, Y is the class of a random observation in S, Y_L is the class of a random observation in S_L, and Y_R is the class of a random observation in S_R. At each node in the tree, the greedy algorithm selects the binary condition that maximizes this metric (that maximizes the information gain).

This describes how to choose the binary condition at each node of the decision tree: choose a condition that maximizes the information gain, and then recurse. When do we stop recursing? There are many possibilities. To list a few:

1. Stop when the set of observations all has the same class. Obviously, at this point we don't need to split any more: we can create a leaf labeled with the class of those observations.

2. Stop when the depth of the node reaches some predetermined level (e.g., depth 10). At this point we can look at the class of all the observations that reach this node, find the class that occurs most often, and make it the label for this node.

3. Stop when the entropy of the class of the observations that reach this node falls below some threshold. Again, we can label the new leaf with the most-common class of the observations that reach this point.

We might combine several of these rules, e.g., 1+2 or 1+3.

Another alternative is to build the entire decision tree, without stopping (except for rule 1 above), and then after it is built, prune the tree to make it smaller. We won't examine this further in this course, but in some settings this can yield even better results.

In pseudocode, we get the following algorithm:

BuildDecTree(S):
1. If all of the entries of S have the same class (or some other stopping condition is met):
2. Build a tree with a single node (a leaf). Label the leaf with the class that appears most frequently in S. Return this tree.
3. For each feature i (i.e., for $i := 1, 2, .., d$):
4. Sort all the values of feature i that appear in S. Let $v_1, \ldots, v_k$ denote those values in increasing order, with duplicates removed, so that $v_1 < v_2 < \cdots < v_k$.
5. For each $t \in \{(v_1+v_2)/2, (v_2+v_3)/2, \ldots, (v_{k-1}+v_k)/2\}$:
6. Compute the information gain metric for the binary condition $x_i \leq t$.
7. Let i, t be the values of i, t that maximize the information gain metric.
8. Split S into S_L and S_R based on the binary condition $x_i \leq t$.
9. Let $T_L :=$ BuildDecTree(S_L) and $T_R :=$ BuildDecTree(S_R).
10. Build a decision tree whose root is labeled $x_i \leq t$, with T_L as its left subtree and T_R as its right subtree. Return this tree.

This greedy algorithm for building a decision tree is often known under the name ID3. We could use this algorithm for classification, by learning a decision tree from the training data and then using it to classify future observations. It turns out this is not a terrible classifier, but we can do even better using the ideas presented next.

Random forests

A *random forest* is a collection of a decision trees. To classify an observation $x \in \mathbb{R}^d$, we classify x with each of the decision trees, treat each tree's output as a vote for x's class, and output whichever class received the most votes. The idea is that any one decision tree might be wrong, but taking a majority vote of multiple different trees will (hopefully) reduce our error rate.

This idea only works if each tree is different. The ID3 algorithm above is deterministic, so if we run it multiple times on the same training set S, we'll get the same decision tree each time, which is not of any help. We want each decision tree in the forest to be different, and ideally as different from the others as possible. We'll do this by injecting some randomness into the tree-building process—this is the "random" part of random forests.

The random forests classifier introduces randomness in two different ways: (a) bagging, and (b) feature sampling. We'll describe each of them next.

Bagging. Rather than training each tree on the same training set S, we sample the training set to generate a random subset of S and train on that subset. In particular, suppose S contains n observations. The simplest approach is to sample, with replacement, n times. In other words, we repeat the following n times: pick a random element of S (uniformly at random, without regard to what prior choices we've made) and add it to our subset. Note that we sample *with replacement*, which means that the same element of S might be selected multiple times. This is OK; we don't try to avoid selecting the same element multiple times.

Let S^* denote the subset selected in this way. We then build a decision tree using the greedy algorithm outlined above (ID3), using S^* as the training set for this tree. We choose a different subset S^* for each decision tree in the forest.

This procedure is called bagging. It helps ensure that each decision tree is different: because each tree is built from a different subset of the training data, the trees will likely be slightly different. Ideally, what we hope will happen is that each tree is better at classifying a certain type of data.

How large will the subset S^* be, with this procedure? We can calculate the answer using linearity of expectation. Let X_i be an indicator random variable, which is 1 if the ith element of S is selected (included in S^*) or 0 if not. Then $|S^*| = X_1 + X_2 + \cdots + X_n$, so the expected size of S^* is $\mathbb{E}[X_1 + \cdots + X_n]$, which is the same as $\mathbb{E}[X_1] + \cdots + \mathbb{E}[X_n]$ (by linearity of expectation). Now

$$\Pr[X_1 = 0] = (1 - 1/n)^n \approx 1/e \approx 0.368,$$

so

$$\mathbb{E}[X_1] = \Pr[X_1 = 1] = 1 - (1 - 1/n)^n \approx 1 - 1/e \approx 0.632.$$

The same is true for each X_i, by symmetry. Therefore, the expected size of S^* is about $0.632n$, or about 63% of the size of the full training set S.

As a generalization, instead of sampling n times from S, you can form S^* by sampling cn times from S (with replacement), where $0 < c \leq 1$ is some small constant, say $c = 1$ or $c = 0.7$.

Feature sampling. The second way we inject randomness is by artificially restricting the set of features that can be used at each node of each decision tree. In particular, when we use ID3 (the greedy algorithm) to build a decision tree, at each node, instead of considering all d possible features, we consider only a subset of the features. In particular, we choose a random subset of $\sqrt{d}$ of the features. Then, we try each of those $\sqrt{d}$ features, and each possible threshold, to find the one that maximizes the information gain; that's what we use at this node of the tree. We choose a different, independent subset of features at each node and for each tree.

The purpose of this is to try to force each decision tree to be as different as possible from the others. Suppose there is a single feature that provides the largest information gain. Without the feature sampling tweak, every decision tree would use that feature at its root. With feature sampling, some trees will use that feature at their root and some won't. (Probably many or most trees will end up using that feature somewhere, but not necessarily at the root.)

The size of the subset is a parameter that can be varied. There is nothing magic about using $\sqrt{d}$ (the square root of the number of features); it's just a value that seems to work fairly well in a number of settings. Others have suggested $2\sqrt{d}$, or trying several different values to see which works best.

Putting it all together. So, here is how we build a random forests classifier. We pick a number of trees—say, 50. We generate 50 random decision trees. For each tree, we use bagging to select a random subset of the training data (a different subset for each tree). We apply the greedy algorithm to this subset of training data to build a decision tree, but at each node considering only a random subset of the features (a different subset for each node and each tree). This gives us a collection of 50 decision trees; that's our forest. Now, to classify a new observation x, each tree votes on the class of x. We output the class that appears most commonly among these 50 votes.

Empirically, the random forests classifier seems to work well in a broad variety of settings. It is an excellent choice for a first machine learning algorithm to try, if you have a new data set.

One of the handy features about random forests is that they can be efficient, and they are very well suited to parallel implementation: you can parallelize both the training process (you can build each decision tree independently, on a separate machine in a big cluster) and the classification process (you can apply each decision tree to your observation in parallel, on multiple cores). This makes random forests a good candidate to consider when very fast classification is needed.

Other sources. If you'd like more information on decision trees and random forests, one nice source is `https://alliance.seas.upenn.edu/~cis520/wiki/index.php?n=Lectures.DecisionTrees`.