Lecture 10: Regression Trees

36-350: Data Mining

October 11, 2006

READING: Textbook, sections 5.2 and 10.5.

The next three lectures are going to be about a particular kind of nonlinear predictive model, namely **prediction trees**. These have two varieties, **regression trees**, which we'll start with today, and **classification trees**, the subject of the next lecture. The third lecture of this sequence will introduce ways of combining multiple trees.

You're all familiar with the idea of linear regression as a way of making quantitative predictions. In simple linear regression, a real-valued dependent variable Y is modeled as a linear function of a real-valued independent variable X plus noise:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

In multiple regression, we let there be multiple independent variables $X_1, X_2, \ldots X_p \equiv \mathbf{X}$,

$$Y = \beta_0 + \beta^T \mathbf{X} + \epsilon$$

This is all very well so long as the independent variables each has a separate, strictly additive effect on Y, regardless of what the other variables are doing. It's possible to incorporate some kinds of interaction,

$$Y = \beta_0 + \beta^T \mathbf{X} + \gamma \mathbf{X}\mathbf{X}^T + \epsilon$$

but the number of parameters is clearly getting very large very fast with even two-way interactions among the independent variables, and stronger nonlinearities are going to be trouble...

Linear regression is a **global model**, where there is a single predictive formula holding over the entire data-space. When the data has lots of features which interact in complicated, nonlinear ways, assembling a single global model can be very difficult, and hopelessly confusing when you do succeed. An alternative approach to nonlinear regression is to sub-divide, or **partition**, the space into smaller regions, where the interactions are more manageable. We then partition the sub-divisions again — this is called **recursive partitioning** — until finally we get to chunks of the space which are so tame that we can fit simple models to them. The global model thus has two parts: one is just the recursive partition, the other is a simple model for each cell of the partition.

Prediction trees use the tree to represent the recursive partition. Each of the **terminal nodes**, or **leaves**, of the tree represents a cell of the partition, and has attached to it a simple model which applies in that cell only. A point x **belongs** to a leaf if x falls in the corresponding cell of the partition. To figure out which cell we are in, we start at the **root node** of the tree, and ask a sequence of questions about the features. The interior nodes are labeled with questions, and the edges or branches between them labeled by the answers. Which question we ask next depends on the answers to previous questions. In the classic version, each question refers to only a single attribute, and has a yes or no answer, e.g., "Is `Horsepower` > 50?" or "Is `GraduateStudent` == FALSE?" Notice that the variables do not all have to be of the same type; some can be continuous, some can be discrete but ordered, some can be categorical, etc. You could do more-than-binary questions, but that can always be accommodated as a larger binary tree. Somewhat more useful would be questions which involve two or more variables, but we'll see a way to fake that in the lecture on multiple trees.

That's the recursive partition part; what about the simple local models? For classic regression trees, the model in each cell is just a *constant* estimate of Y. That is, suppose the points $(x_i, y_i), (x_2, y_2), \ldots (x_c, y_c)$ are all the samples belonging to the leaf-node l. Then our model for l is just $\hat{y} = \frac{1}{c}\sum_{i=1}^{c} y_i$, the sample mean of the dependent variable in that cell. This is a piecewise-constant model.[1] There are several advantages to this:

- Making predictions is fast (no complicated calculations, just looking up constants in the tree)
- It's easy to understand what variables are important in making the prediction (look at the tree)
- If some data is missing, we might not be able to go all the way down the tree to a leaf, but we can still make a prediction by averaging all the leaves in the sub-tree we do reach
- The model gives a jagged response, so it can work when the true regression surface is not smooth. If it is smooth, though, the piecewise-constant surface can approximate it arbitrarily closely (with enough leaves)
- There are fast, reliable algorithms to learn these trees

Figure 1 shows an example of a regression tree, which predicts the price of cars. (All the variables have been standardized to have mean 0 and standard deviation 1.) The R^2 of the tree is 0.85, which is significantly higher than that of a multiple linear regression fit to the same data ($R^2 = 0.8$, including an interaction between `Wheelbase` and `Horsepower` < 0.)

[1] If all the variables are quantitative, then instead of taking the average of the dependent variable, we could, say, fit a linear regression to all the points in the leaf. This would give a piecewise-linear model, rather than a piecewise-constant one. If we've built the tree well, however, there are only a few, closely-spaced points in each leaf, so the regression surface would be nearly constant anyway.

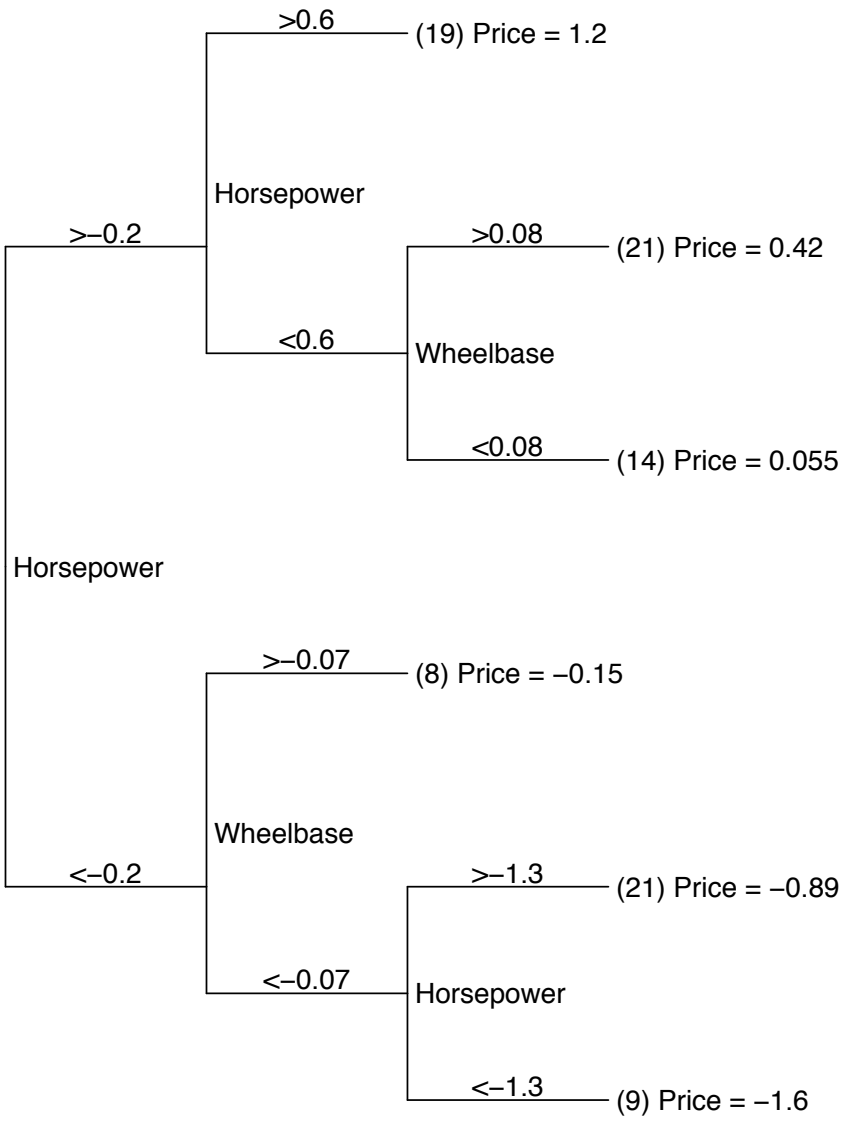

Figure 1: Regression tree for predicting price of 1993-model cars. All features have been standardized to have zero mean and unit variance. Note that the order in which variables are examined depends on the answers to previous questions. The numbers in parentheses at the leaves indicate how many cases (data points) belong to each leaf.

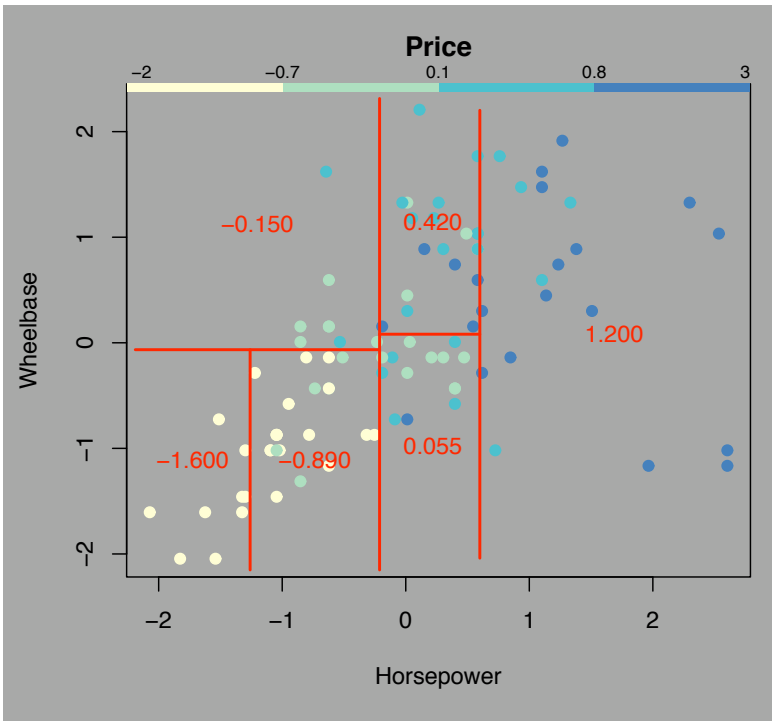

Figure 2: The partition of the data implied by the regression tree from Figure 1. Notice that all the dividing lines are parallel to the axes, because each internal node checks whether a single variable is above or below a given value.

The tree correctly represents the interaction between `Horsepower` and `Wheelbase`. When `Horsepower` > 0.6, `Wheelbase` no longer matters. When both are equally important, the tree switches between them. (See Figure 2.)

Once we fix the tree, the local models are completely determined, and easy to find (we just average), so all the effort should go into finding a good tree, which is to say into finding a good partitioning of the data. We've already seen, in clustering, some ways of doing this, and we're going to apply the same ideas here.

In clustering, remember, what we would ideally do was maximizing $I[C; X]$, the information the cluster gave us about the features X. With regression trees, what we want to do is maximize $I[C; Y]$, where Y is now the dependent variable, and C are now is the variable saying which leaf of the tree we end up at. Once again, we can't do a direct maximization, so we again do a greedy search. We start by finding the one binary question which maximizes the information we get about Y; this gives us our root node and two daughter nodes. At each daughter node, we repeat our initial procedure, asking which question would give us the maximum information about Y, given where we already are in the

tree. We repeat this recursively.

One of the problems with clustering was that we needed to balance the informativeness of the clusters with parsimony, so as to not just put every point in its own cluster. Similarly, we could just end up putting every point in its own leaf-node, which would not be very useful. A typical **stopping criterion** is to stop growing the tree when further splits gives less than some minimal amount of extra information, or when they would result in nodes containing less than, say, five percent of the total data. (We will come back to this in a little bit.)

We have only seen entropy and information defined for discrete variables. You *can* define them for continuous variables, and sometimes the continuous information is used for building regression trees, but it's more common to do the same thing that we did with clustering, and look not at the mutual information but at the sum of squares. The sum of squared errors for a tree T is

$$S = \sum_{c \in \text{leaves}(T)} \sum_{i \in C} (y_i - m_c)^2$$

where $m_c = \frac{1}{n_c} \sum_{i \in C} y_i$, the prediction for leaf c. Just as with clustering, we can re-write this as

$$S = \sum_{c \in \text{leaves}(T)} n_c V_c$$

where V_c is the within-leave variance of leaf c. So we will make our splits so as to minimize S.

The basic regression-tree-growing algorithm then is as follows:

1. Start with a single node containing all points. Calculate m_c and S.
2. If all the points in the node have the same value for all the independent variables, stop. Otherwise, search over all binary splits of all variables for the one which will reduce S as much as possible. If the largest decrease in S would be less than some threshold δ, or one of the resulting nodes would contain less than q points, stop. Otherwise, take that split, creating two new nodes.
3. In each new node, go back to step 1.

Trees use only one predictor (independent variable) at each step. If multiple predictors are equally good, which one is chosen is basically a matter of chance. (In the example, it turns out that `Weight` is just as good as `Wheelbase`: Figure 3.) When we come to multiple trees, two lectures from now, we'll see a way of actually exploiting this.

One problem with the straight-forward algorithm I've just given is that it can stop too early, in the following sense. There can be variables which are not very informative themselves, but which lead to very informative subsequent splits. (When we were looking at interactions in the Usenet data, we saw that the word "to" was not very informative on its own, but was highly informative in combination with the word "cars".) This suggests a problem with stopping

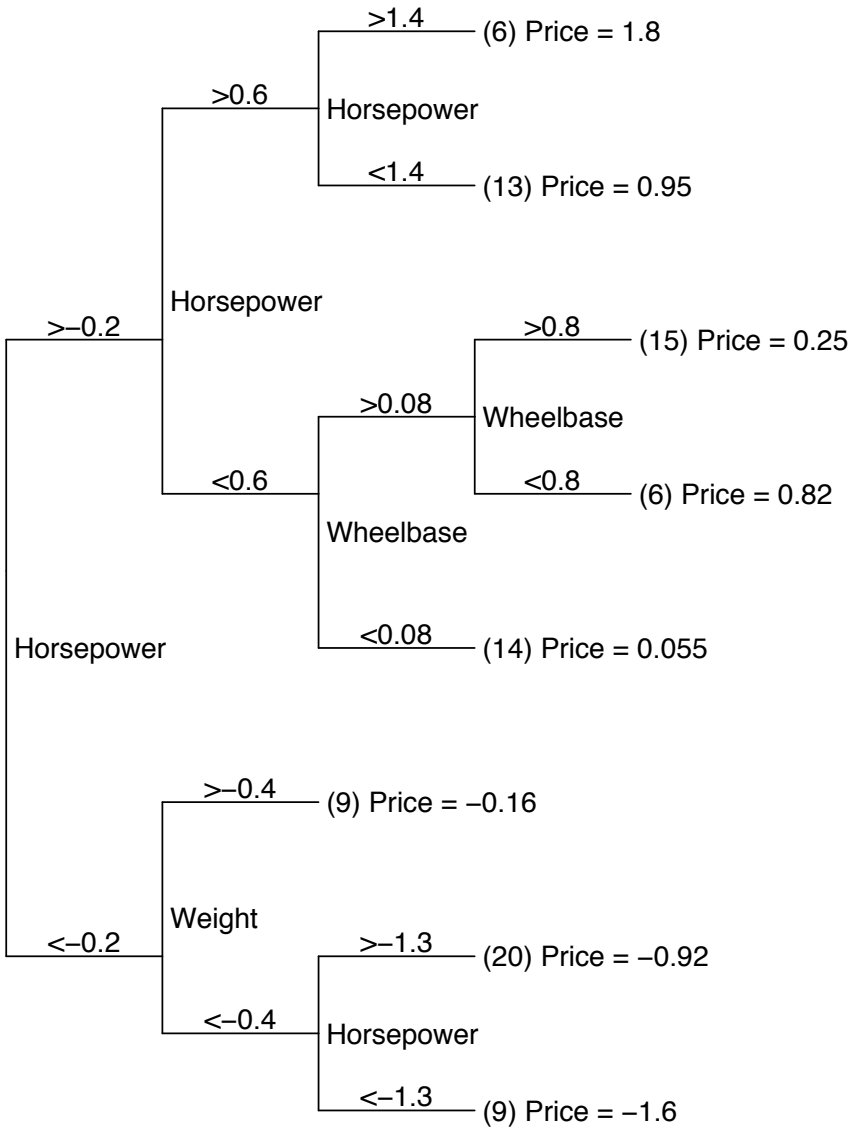

Figure 3: Another regression tree for the price of cars, where `Weight` was used in place of `Wheelbase` at the second level. The two perform equally well.

when the decrease in S becomes less than some δ. Similar problems can arise from arbitrarily setting a minimum number of points q per node.

A more successful approach to finding regression trees uses the idea of cross-validation from last time. We randomly divide our data into a training set and a testing set, as in the last lecture (say, 50% training and 50% testing). We then apply the basic tree-growing algorithm to the training data *only*, with $q = 1$ and $\delta = 0$ — that is, we grow the largest tree we can. This is generally going to be too large and will over-fit the data. We then use cross-validation to **prune** the tree. At each pair of leaf nodes with a common parent, we evaluate the error on the *testing* data, and see whether the sum of squares would be smaller by remove those two nodes and making their parent a leaf. This is repeated until pruning no longer improves the error on the testing data.

There are lots of other cross-validation tricks for trees. One cute one is to alternate growing and pruning. We divide the data into two parts, as before, and first grow and then prune the tree. We then exchange the role of the training and testing sets, and try to grow our pruned tree to fit the second half. We then prune again, on the first half. We keep alternating in this manner until the size of the tree doesn't change.