Classification and Regression Trees

Ryan P. Adams

So far, we have primarily examined linear classifiers and regressors, and considered several different ways to train them. When we've found the linearity assumption to be lacking, we've introduced nonlinear basis functions to transform the input space. In the case of neural networks, we've even looked at how to adjust those basis functions to explain the data better. There are some things that you might not like about linear methods, however. First, they are *global* methods and apply the same types of rules to data everywhere. That is, there's no easy way to learn a rule like "be a linear function of x_1 over here, but be a function of x_2^3 over in this other region of the space." Second, linear methods are always additive in their features. Our regressors (and the activations for classifiers) have had the form:

$$y(\boldsymbol{x}, \boldsymbol{w}) = w_0 + \sum_{j=1}^{J} \phi_j(\boldsymbol{x}) w_j. \tag{1}$$

This means that the effects of any two $\phi_j(\cdot)$ functions always have to be weighted sums. They can't have a more complicated relationship. Finally, these methods can be frustrating, because they're not easily *interpretable*, i.e., we can't look inside something like a neural network and readily understand how it produces the decisions it does.

One alternative approach to linearity, that resolves some of these issues, is to use a *decision tree*. A decision tree is a very simple way to make a prediction from a feature vector, by applying a sequence of rules recursively. Following the setup that we've been using, let $\boldsymbol{x}$ be a D-dimensional feature vector, and let t be a scalar target. Decision trees can be classifiers or regressors, depending on the type of the target. For example, Table 1 shows a simple toy classification problem in which the objective is to decide whether to play golf, and the inputs are five attributes about the weather: "Outlook" can be *sunny*, *overcast*, or *rain*; "Temperature" is a positive number; "Humidity" is a positive number; and "Windy" is a binary value. Figure 1 shows one possible decision tree for these data. The basic idea is straightforward. Each level of the tree examines one of the attributes and determines which branch to take. If you encounter a terminal node, you return the value associated with that node. So, in Figure 1, with a sunny outlook with low humidity we play golf, while we don't play if it is both windy and rainy. If it is overcast, we always play.

The constraints usually placed on a decision tree are that each node branches on exactly one attribute (dimension of the input), and that each attribute appears at most once in a given path. Note that different paths might use different sets of attributes or use them in a different order. For example, in Figure 1, Humidity is not accessed on the right branch, and Windy is not accessed on the left branch. A *decision stump* is a special case of a decision tree, which has only a branching node, i.e., it makes a decision based on a single attribute. When used for regression, decision trees return real-valued numbers at the terminal nodes.

Outlook	Temperature	Humidity	Windy	Play?
sunny	85	85	false	no
sunny	80	90	true	no
overcast	83	78	false	yes
rain	70	96	false	yes
rain	68	80	false	yes
rain	65	70	true	no
overcast	64	65	true	yes
sunny	72	95	false	no
sunny	69	70	false	yes
rain	75	80	false	yes
sunny	75	70	true	yes
overcast	72	90	true	yes
overcast	81	75	false	yes
rain	71	80	true	no

Table 1: Table of attributes and targets for when to play golf.

Although very simple, decision trees have several desirable properties. First they are very simple and fast at test time – they may not even need to examine all of the dimensions of the input to make a prediction. Second, they are highly interpretable – they are equivalent to a very straightforward nested sequence of if/else statements. Third, they can represent complicated nonlinear effects – as lower branches can be different across the tree, they can use different kinds of criteria in different regions of the input space. The downsides of decision trees are that they cannot easily represent functions that are smoothly varying with inputs, or those that depend on many input variables. For example, it is very challenging for a decision tree to implement the simple majority function over a set of features, whereas a linear classifier can easily capture this.

1 Learning a Classification Tree

We've so far talked about *using* a decision tree, but how do we learn one from data? You might at first think about trying to enumerate all possible trees and then pick the one that is best. However, this isn't feasible in general. Imagine that the input was a simple D-dimensional binary vector and that our goal was to produce a binary classification. Given a truth table for such a binary-to-binary classifier, there are clearly many possible trees that can implement it — at least $D!$ of them because we can look at the attributes in any order. And there are 2^{2^D} possible truth tables! (There are 2^D rows in a truth table and each row can have two possible output values.) Needless, to say, we can't expect to enumerate this. Instead, we use a "greedy" algorithm that tries to build a good tree from top to bottom, recursively adding branches that look like they improve prediction. The idea is to start at the root of the tree and choose the most informative attribute you can, and then descend into the branches and repeat. At each stage, you're greedily adding the best decision that you can into that portion of the tree.

But what makes one attribute better than another? How do we pick which decision to add next? We need a way to measure the quality of a new node in the decision tree. The most common

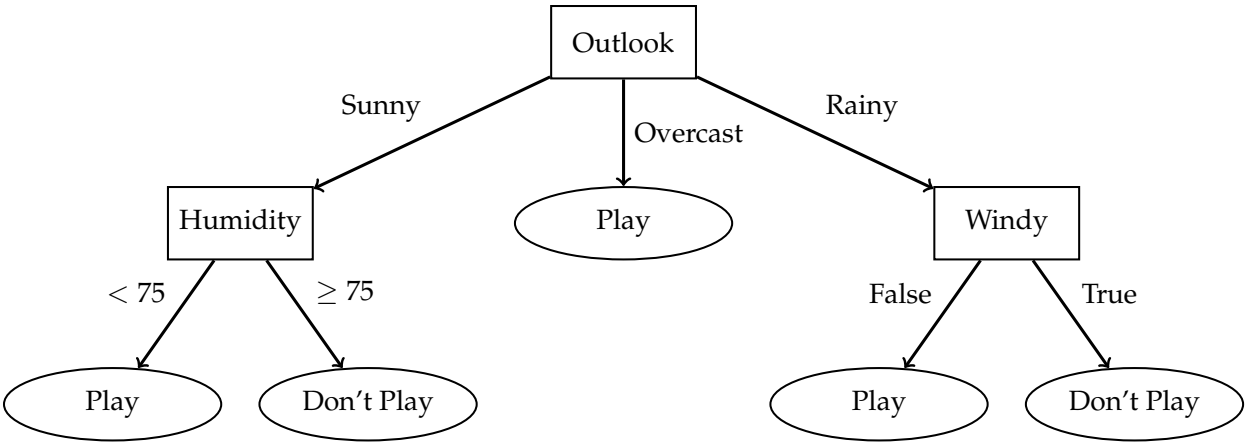

Figure 1: Example of a decision tree for the golf data in Table 1.

way to do this is to use the concept of *information gain*, which is the basis of the popular greedy decision tree learning algorithm called ID3. Information gain attempts to measure exactly what it sounds like: if we knew the answer to this node's question (e.g., if we knew whether it was windy or not) how much information would that give us about the eventual label that we want to produce? Information is conventionally measured in *bits*, just like on your computer. If there is a binary variable that we'd like to know, and we have 0 bits about it, then we know nothing. If we have 1 bit about it, then we know it perfectly. This measure of information, however, allows us to talk about fractional bits in which we know something but not everything about some unknown quantity.

1.1 Shannon Entropy

This measure of information arises from a concept called *Shannon entropy*, which is a way to measure the uniformity of a distribution. If Z is a random variable that can take K possible values, with probabilities $p_1, p_2, \cdots, p_K$, where $p_k > 0$ and $\sum_k p_k = 1$, then the entropy of Z is

$$H(Z) = -\sum_{k=1}^{K} p_k \log_2 p_k, \tag{2}$$

where we conventionally take $0 \log_2 0 = 0$. This quantity has the units of bits and can be tied back to our previous notions of information when $K = 2$. If we are perfectly uncertain about Z, then $p_1 = p_2 = 1/2$ and $H(A) = 1$. When we are perfectly certain about Z, then either $p_1 = 1$ and $p_2 = 0$ or vice versa, and $H(A) = 0$. When $K > 2$ then the maximum entropy is when $p_k = 1/K$ and $H(A) = \log_2 K$.

When we're learning a classifier, we are concerned about the entropy of the label; we want information about how to predict it. This means we'll need to compute an empirical estimate of the entropy from data. To do that, we first fit a distribution to the data, then we compute entropy of that distribution. For example, in the golf data of Table 1, there are 5 *no* labels and 9 *yes* labels. If we assume these are from an unknown Bernoulli distribution with probability ρ (where ρ is the probability of a *yes*), we can compute ρ from the data using maximum likelihood, ignoring the

input features for now:

$$\rho_{\mathsf{MLE}} = \arg\max_{\rho \in (0,1)} p(9 \text{ yes and } 5 \text{ no} \mid \rho) \tag{3}$$

$$= \arg\max_{\rho \in (0,1)} \rho^9 (1-\rho)^5 \tag{4}$$

$$= \arg\max_{\rho \in (0,1)} \left(9 \log \rho + 5 \log(1-\rho)\right). \tag{5}$$

As usual, we differentiate in terms of ρ, set to zero and solve:

$$\frac{d}{d\rho}\left[9 \log \rho + 5 \log(1-\rho)\right] = \frac{9}{\rho} - \frac{5}{1-\rho} = 0 \tag{6}$$

$$\rho = 9/14. \tag{7}$$

That is, the maximum likelihood estimate of ρ is just the empirical frequency of the *yes* labels. Now we can go back and plug this into our entropy computation and get

$$H(\text{label}) = -9/14 \log_2(9/14) - 5/14 \log_2(5/14) \approx 0.94\text{bits}. \tag{8}$$

1.2 Specific Conditional Entropy

Knowing the entropy of the label isn't all that useful by itself. What we really want to know is how much information we get from various attributes. One step in that direction is to compute the *specific conditional entropy*. This is the entropy of a random variable, conditioning on a particular value of another random variable. If we have two random variable A and B, then we can compute the specific conditional entropy of A given that $B = b$:

$$H(A \mid B = b) = -\sum_{a \in \mathcal{A}} p(a \mid b) \log_2 p(a \mid b), \tag{9}$$

where $\mathcal{A}$ is the set of outcomes that A can take. For example, we might talk about the entropy of the label, conditioned on Outlook=rain. We can go back into Table 1 and look at all the rows in which the outlook is rainy and then compute the empirical entropy just like we did before: there are now 3 *yes* and 2 *no*, so the entropy is approximately 0.97 bits.

1.3 Conditional Entropy

The specific conditional entropy isn't necessarily all that useful. What we really want is the *conditional entropy*, because it averages over all possible values of the thing we're conditioning on. It answers the question "on average, what will the entropy of A be after seeing B?" You can probably see how this is starting to look useful because we can now pose questions about how much knowing one thing changes our uncertainty about another thing. If our two random variables are A and B as before, the conditional entropy is

$$H(A \mid B) = \sum_{b \in \mathcal{B}} p(b) H(A \mid B = b) \tag{10}$$

$$= -\sum_{a \in \mathcal{A}} \sum_{b \in \mathcal{B}} p(a, b) \log_2 p(a \mid b). \tag{11}$$

Here $\mathcal{B}$ is the set of outcomes that B can take. Let's see how this would work for the Windy attribute from the golfing example. We can make a table with the different counts:

	Play	Don't Play		
Windy	3/14	3/14	$H(\text{Play} \mid \text{Windy}) = 1$	$p(\text{Windy}) = 6/14$
Not Windy	6/14	2/14	$H(\text{Play} \mid \text{Not Windy}) \approx 0.81$	$p(\text{Not Windy}) = 8/14$

So the conditional entropy of the label given knowledge of the wind is about 0.89 bits.

1.4 Mutual Information

Now we can really answer the question we care about: how many bits about A to we get from knowing B? The *mutual information* gives us a direct way to quantify the knowledge we gain about the label by knowing one of the attributes. Mutual information is a lot like correlation, but it is actually more general. Whereas correlation tells us about *linear* dependence, mutual information tells us about any kind of dependence. In general, mutual information is very challenging to estimate, but when things are finite and discrete we can compute it using counts like we've been doing here. What mutual information computes is the difference in entropy before we know a related quantity, versus the expected entropy after knowing the related quantity. If the entropy drops a lot, that means we gained a lot of information!

$$
\begin{aligned}
I(A\,;\,B) &= H(A) - H(A \mid B) && \text{(Entropy minus conditional entropy.)} && (12)\\
&= H(B) - H(B \mid A) && \text{(MI is symmetric.)} && (13)\\
&= H(A) + H(B) - H(A, B) && \text{(Individual entropies, minus joint entropy.)} && (14)\\
&= \sum_{a \in \mathcal{A}} \sum_{b \in \mathcal{B}} p(a, b) \log_2 \frac{p(a, b)}{p(a)\, p(b)} && && (15)
\end{aligned}
$$

Above are several ways to think about mutual information. In particular, notice that if A and B are independent, i.e., $p(a, b) = p(a)p(b)$, then the mutual information is zero. Now, let's return to the running example and compute the mutual information between windiness and the label of whether to play or not. We computed the entropy of the label by itself to be $H(\text{Play}) = 0.94$ and in the previous section we computed the conditional entropy of play given windiness to be $H(\text{Play} \mid \text{Windy}) = 0.89$. This means that the mutual information between these two quantities can be estimated as

$$
I(\text{Play}\,;\,\text{Wind}) = H(\text{Play}) - H(\text{Play} \mid \text{Windy}) = 0.94 - 0.89 = 0.05 \text{ bits.} \tag{16}
$$

1.5 Selecting Attributes for Branching

Mutual information gives us a great way to assess how useful an attribute will be for our decision tree. If the mutual information is large, then we get lots of information about the label when we know the attribute. When it is very small, we get little information. We've done the computation now for the Windy attribute. Let's do it for the Outlook attribute :

	Play	Don't Play		
sunny	2/14	3/14	$H(\text{Play} \mid \text{sunny}) \approx 0.97$	$p(\text{sunny}) = 5/14$
overcast	4/14	0/14	$H(\text{Play} \mid \text{overcast}) = 0$	$p(\text{overcast}) = 4/14$
rain	3/14	2/14	$H(\text{Play} \mid \text{rain}) \approx 0.97$	$p(\text{rain}) = 5/14$

The overall entropy of the label is 0.94 as before, but now if we know the Outlook attribute, we reduce it to about 0.69 bits. This means that the mutual information between Outlook and the label is about 0.25 bits. As a result, we'd choose to use the Outlook attribute at the root, rather than the Windy attribute – we get more information from it.

What about the Temperature and Humidity attributes, which are numbers? Here the natural thing to do is to create a threshold, and set that threshold using the mutual information. If there is a threshold that makes it highly informative, then choose that attribute and threshold for the node. Let's look at a few thresholds for Temperature:

Threshold	Above Threshold	Above and Play	$I(\text{Play}\,;\,\text{Temperature})$
60	14/14	9/14	0
65	12/14	8/12	0.01
70	9/14	5/9	0.05
75	4/14	2/4	0.03
80	3/14	2/3	0.00
85	0/14	0/0	0

We can do the same for Humidity:

Threshold	Above Threshold	Above and Play	$I(\text{Play}\,;\,\text{Humidity})$
60	14/14	9/14	0
65	13/14	8/13	0.05
70	10/14	6/10	0.01
75	9/14	5/9	0.05
80	5/14	2/5	0.10
85	4/14	2/4	0.03
90	2/14	1/2	0.01
95	1/14	1/1	0.05
100	0/14	0/0	0

From this we can see that none of these thresholds beat Outlook for mutual information with the label. That's exactly what we see in Figure 1: we greedily make the first node of the tree split on the Outlook attribute.

What should we add next to the tree? We can determine this by recursing into each of the three branches and applying the mutual information criterion again. This time, however, we'll only use a subset of the data – the data which matched the Outlook attribute. In the case where the Outlook is overcast, we're already done! We don't need to split any further because *all* of the data in this case can be labeled with Play=yes.

There are only five rows with Outlook=sunny. We can make our second level split of these data using Windy, Temperature, or Humidity. For Windy, we get

	Play	Don't Play
Windy	1/5	1/5
Not Windy	1/5	2/5

and we compute a mutual information of 0.02 bits. For Temperature:

Threshold	Above Threshold	Above and Play	MI
65	5/5	2/5	0
70	4/5	1/4	0.32
75	2/5	0/2	0.42
80	1/5	0/1	0.17
85	0/5	0/0	0

The best we can do with Temperature if 0.42 bits, but with Humidity we get:

Threshold	Above Threshold	Above and Play	MI
65	5/5	2/5	0
70	3/5	0/3	0.97
75	3/5	0/3	0.97
80	3/5	0/3	0.97
85	2/5	0/2	0.42
90	1/5	0/1	0.17
95	0/5	0/0	0

We can set a threshold on Humidity to get 0.97 bits. In fact, this separates the remaining five data perfectly. This gives us the left branch of the tree in Figure 1. If we do the same thing again with the right branch of the tree, which is when Outlook is rain, then we see that even though Windy was not useful for sunny Outlooks, it perfectly predicts under rainy outlooks. This highlights one of the reasons decision trees are interesting: you can do test different conditions based on the outcome of earlier conditions.

Thus the overall algorithm starts with all of the data. It then computes the mutual information between each attribute (perhaps via thresholding) and the label. A node is then introduced that uses that attribute to split the data on the selected attribute criterion. You then recurse into each split. If the remaining data are *pure* – all of one class – then you add a terminal node that returns that label. If there are no further attributes to examine, or if you reach the maximum depth you wish to consider, you return the majority label among the data in that partition.

2 Learning a Regression Tree

In principle, it is possible to learn a regression tree using the notions of entropy and mutual information. However, in continuous spaces these quantities can be much harder to compute because it is necessary to perform density estimation. We have the additional complication that each of our data may have a unique label, as they are real numbers. As a result, we use a measure of squared error rather than mutual information to decide when to split.

In classification, at a terminal node, we said the predicted label was the majority vote among the data assigned to that node's partition. In regression, the predicted label is the *mean* of the data within that partition. Now, extending the intuitions we've built from K-Means clustering, PCA, and regression, we can consider the squared loss associated with a particular partitioning. This squared loss is the sum of squared differences between a given datum's label and the mean for the partition it belongs to. We'd like this to be small, because that means all of the labels in the partition are predicted well. When we consider adding a new node to the tree, we're now asking: "is there an attribute on which a split would result in a reduced overall squared error?" That is,

can we use an attribute to put the data into buckets such that the means with the buckets are better fits to each labels? This is almost exactly what we did with K-Means clustering, except now we're driving the decisions about partitioning with the attributes. This could be extended further by having each of the partitions have a complete local regression model, using the squared residuals to decide where and when to split.

3 Regularization

If you have many input attributes, the algorithm described above will continue recursing until it has perfectly explained all of the data. As we've seen in lecture and in the practicals, this kind of "perfect fitting" can really be overfitting, because the model is explaining the noise along with whatever signal is present. The natural inductive bias for a decision tree is to limit the depth. An obvious first pass at this is to simply put a limit on how many levels of recursion are allowed. Another idea is to put a minimum on the mutual information that is required for a node to be added; this prevents "weakly informative" nodes from making the tree too large. Such thresholds could also be determined via statistical tests: only split the data set if the two partitions are significantly different according to a chi-squared or binomial test. These are all ideas for *pre-pruning*, but one could also do *post-pruning* and remove parts of the tree based on validation performance.