Decision Trees

Alex Sverdlov
alex@theparticle.com

1 Decision Trees

We begin the pattern recognition review with decision trees. They are a graphical representation of decision making that is easily interpretable by humans. Despite their simplicity, they are quite robust. Their step-by-step explanatory capability is very valuable in getting people to trust the output. Often, another mechanism is trained for the sole purpose of labeling enough examples for a 'final model' decision tree to be constructed—that way we get the flexibility of another model, and the readability of a decision tree.

To label an unknown example $\boldsymbol{x}$, we start from the root of the tree, and at each node apply a test that tests a particular attribute x_i. The result of this test tells us which child branch to explore for subsequent tests. This continues recursively until a leaf node is reached—the label on the leaf node is the output. When the target is numeric, we may call this setup a Regression Tree.

In this scenario, the training data are often represented as a set of tuples $\boldsymbol{X}$ with each tuple being $(x_1, \ldots, x_n, x_t)$ where each x_i is either a string or a number. The x_t attribute is the target label. The classification task is: Given a new $(x_1, \ldots, x_n)$, predict the target label x_t.

Learning decision trees is done by accepting a set of labeled tuples (often very expensive to label), and recursively deciding what test (on which attribute) to apply at each node. The goal is to induce a decision tree that has good accuracy on the training set, and generalizes well on verification set.

A related view of decision trees is to consider the subspaces decision rules create—the tree recursively partitions the concept space into disjoint subspaces—with each subspace representing a particular label.

We can also view a decision tree as a collection of tests. Every path from the root node to the leaf represents a list of tests that need to be applied to classify an example with the leaf label. With this view (treating the tree as a bag of tests as opposed to a hierarchical tree), it is easier to spot undesirable tests and remove them.

Most decision tree learning algorithms are variations on the top-down greedy search algorithm, with the most notable example being ID3 (Interactive Dichotomizer 3) by Quinlan. [**?**]. Before Quinlan, Breiman et al. presented the concept in an 1984 book *Classification*

and Regression Trees [?], and before that, Hunt did experiments with a Concept Learning System (CLS) [?], which Quinlan references as inspiration and a precursor to ID3.

Hunt's Concept Learning System was a divide and conquer scheme that could handle binary (positive and negative) target values, with the decision attribute being decided by a heuristic based on the largest number of positive cases. Hunt speculated about using information theoretic measure—noting that humans in the game of 20 questions will ask questions that maximize information gain.

ID3 improves on Hunt's approach by using an information theoretic measure of *information gain* to decide on the test attribute, and allowing for multi-valued target labels.

C4.5, also proposed by Quinlan [?], improves on ID3 with ability to handle numeric attributes and dealing with missing values. C4.5 also introduced a rule pruning mechanism—to avoid over-fitting the tree to the training data.

1.1 ID3

Starting with a set of tuples $\boldsymbol{X}$, where each tuple is $(x_1, \ldots, x_n, x_t)$, ID3 algorithm calculates *information gain* on every attribute, and uses the highest information gain attribute to split the dataset. The algorithm then proceeds recursively on each resulting tuple set.

Information gain measures the reduction in entropy of the target attribute minus the entropy of the target attribute after the split on attribute i:

$$Gain(\boldsymbol{X}, i) = Entropy(\boldsymbol{X}) - \sum_{v \in Values(i)} \frac{|\boldsymbol{X}_v|}{|\boldsymbol{X}|} Entropy(\boldsymbol{X}_v)$$

where *Entropy* is a measure of bits it takes to represent the *target attribute* x_t and $\boldsymbol{X}_v$ is the set of tuples resulting from the split on value v of attribute i. The $|\boldsymbol{X}_v|/|\boldsymbol{X}|$ scales the split entropy by the number of elements in the set after the split on v.

$$Entropy(X) = -\sum_{v \in Values(\boldsymbol{X}_t)} P(v) \log_2 P(v)$$

where $\boldsymbol{X}_t$ are all the values of the target attribute x_t.

Consider the dataset in Table 1. We would like to use this dataset to decide on the form of transportation, given whether we are in a hurry, how much money we have, and whether the train is late. For example, if we are in a hurry, and have $50, then we should take the taxi. Similarly, even if we are in a hurry, but only have $10, then we are stuck taking the train.

To build a decision tree using this dataset, we need to compute information gain of splitting on every column.

The entropy of the $Method$ column is:

$$-\left(\frac{2}{7}\log\left(\frac{2}{7}\right) + \frac{5}{7}\log\left(\frac{5}{7}\right)\right) = 0.59827$$

Hurry	Money	TrainLate	Method
N	50	N	Train
N	50	Y	Taxi
Y	50	N	Taxi
Y	10	N	Train
Y	10	Y	Train
N	10	N	Train
N	10	Y	Train

Table 1: Decision Tree training tuple.

The entropy of $Method$ after splitting on each other column is:

$$\begin{aligned}
\text{Hurry:} \quad & -((\tfrac{1}{4})\log(\tfrac{1}{4}) + \tfrac{3}{4}\log(\tfrac{3}{4}))\tfrac{4}{7} + -(\tfrac{1}{3}\log(\tfrac{1}{3}) + \tfrac{2}{3}\log(\tfrac{2}{3}))\tfrac{3}{7} = 0.59413 \\
\text{Money:} \quad & -(\tfrac{1}{3}\log(\tfrac{1}{3}) + \tfrac{2}{3}\log(\tfrac{2}{3})) * \tfrac{3}{7} + 0 = 0.27279 \\
\text{TrainLate:} \quad & -((\tfrac{1}{4})\log(\tfrac{1}{4}) + \tfrac{3}{4}\log(\tfrac{3}{4}))\tfrac{4}{7} + -(\tfrac{1}{3}\log(\tfrac{1}{3}) + \tfrac{2}{3}\log(\tfrac{2}{3})) * \tfrac{3}{7} = 0.59413
\end{aligned}$$

So we decide to split on the $Money$ column, this leaves us with two datasets, Table 2. The dataset where the only $Method$ is $Train$ cannot be split—in fact, the entropy for that one is 0. We only have to worry about the smaller dataset.

Hurry	Money	TrainLate	Method
Y	10	N	Train
Y	10	Y	Train
N	10	N	Train
N	10	Y	Train

Hurry	Money	TrainLate	Method
N	50	N	Train
N	50	Y	Taxi
Y	50	N	Taxi

Table 2: Dataset after splitting on the $Money$ column.

Now for the smaller of the two tables, the entropy of $Method$ after splitting on each other column is:

$$\begin{aligned}
\text{Hurry:} \quad & -(\tfrac{1}{2}\log(\tfrac{1}{2}) + \tfrac{1}{2}\log(\tfrac{1}{2}))\tfrac{1}{2} + 0 = 0.34657 \\
\text{TrainLate:} \quad & -(\tfrac{1}{2}\log(\tfrac{1}{2}) + \tfrac{1}{2}\log(\tfrac{1}{2}))\tfrac{1}{2} + 0 = 0.34657
\end{aligned}$$

Since we get the same information gain, we can just pick to split on $Hurry$ column, Table 3.

Hurry	Money	TrainLate	Method
N	50	N	Train
N	50	Y	Taxi

Hurry	Money	TrainLate	Method
Y	50	N	Taxi

Table 3: Splitting the smaller dataset on $Hurry$ column.

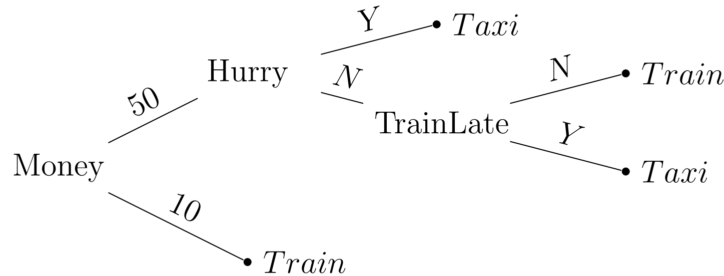

Figure 1: Decision tree built from the sample data.

We do the same logic for the *TrainLate* column, and end up with a tree illustrated in Figure 1.

Using such a decision tree is very intuitive and it is easy to understand what the tree is doing. We start at the root, and determine if our *Money* amount allows for any options—if not, we take the *Train*. If we have enough money, we need to determine if we are in a *Hurry*, if yes, we take *Taxi*. If we are not in a hurry, we check to see if the train is late, if yes, we take *Taxi*, otherwise we take *Train*. Easy to understand and apply.

The major limitation of ID3 is the inability to calculate information gain for a numeric attribute. It is lucky that our *Money* variable only had two values. In general, numerical attributes, especially those that function as keys, will have very high information gain, without any predictive power.

Running ID3 to conclusion (recursively until the entire dataset is exhausted), results in over fitting the training data. This can be avoided by stopping the algorithm early or later pruning branches.

1.2 C4.5

The ID3 method is not without problems, and C4.5 is essentially a set of adjustments to the basic ID3 algorithm to make it work better. For one, the *Gain* has a tendency of favoring unique identifiers. If we apply *Gain* on a database table, it will pick out all the keys, dates, ids, etc—none of which generalize.

When calculating which value to split on, C4.5 takes the number of distinct values into consideration. If a node will branch out a million different children, then we generally do not want to use that attribute.

$$\mathit{Split}(\boldsymbol{X}, i) = -\sum_{j=1}^{n} \frac{|\boldsymbol{X}_j|}{|\boldsymbol{X}|} \log_2 \frac{|\boldsymbol{X}_j|}{|\boldsymbol{X}|} \qquad \mathit{GainRatio}(\boldsymbol{X}, i) = \frac{\mathit{Gain}(\boldsymbol{X}, i)}{\mathit{Split}(\boldsymbol{X}, i)}$$

where $\boldsymbol{X}_j$s are n subsets resulting from partitioning $\boldsymbol{X}$ by n values of attribute i.

C4.5 introduces a way of dealing with numerical values. If an attribute i is numeric, it has N distinct numeric values (the datasets are finite). Such an attribute presents $N-1$ potential splits (we can split that attribute at any of the $N-1$ values.

To efficiently calculate the information gain for each of the $N-1$ split points of a numerical attribute we need to sort the dataset on values of that attribute. Once the numeric attribute is sorted, it is feasible to calculate information gain for each of the $N-1$ split points using a single iteration over the data.

Missing values are addressed by calculating the ratio of non-missing values within the split, and weighing the tuple with the missing values according to the ratio of the non-missing values. For example, if a node is to be split in two, and has 2 negative values, and 3 positive values, and two missing values, then the two missing values will be weighted as 0.4 negative, and 0.6 positive.

Another major improvement C4.5 brings is pruning. We start out over-fitting the tree—apply the tree building algorithm to completion. Then we convert the resulting tree into a bag of rules (each path from root to leaf becomes a conjunction rule). For each such conjunction rule, remove individual attribute tests if such a removal does not hurt the rule's classification performance (using validation tests). Sort the conjunction rules by their *estimated accuracy* (estimated by applying rule to either training samples or a different verification set), and apply them in order until a rule succeeds—once a rule succeeds, we have our classification. This scheme avoids the dangers of over-fitting and under fitting.