Decision Trees

One disadvantage of many classification techniques is that the classification process is difficult to understand. For a nearest neighbor or bayesian classifier, comparing dozens or hundreds of features determines the final class. A user that wants to know why sometime was classified the way it was is forced to examine these same dozens or hundreds of features. Similarly, mathematical classification techniques are often difficult for humans to understand.

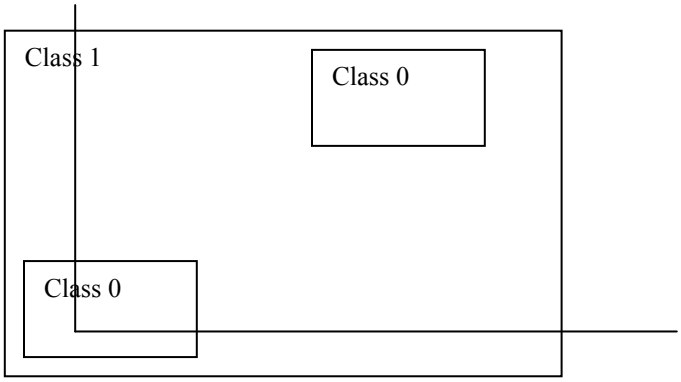
However, humans do easily understand and accept decision rules. These are rules in the format of "*If X is true and Y is false, then conclude class 1.*" Typically, continuous variables are split up into a range, so that something like age could be checked through "*If age>60 and heart attack=true then…*".

For decision trees we'll try to build up a set of conjunctive decision rules. In this format, we only have AND's within each rule, but each rule exists within an IF-THEN-ELSE structure. For example, the following set of rules solves the XOR problem:

```
If x=0 then
        If y=0 then class=0
        Else class = 1
Else if x=1 then
        If y=0 then class=1
        Else class = 0
```

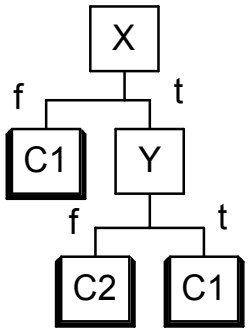
Note that the order that the rules are executed is important. Also, the rules cover all of the classes; this can make a large decision tree difficult to understand.

Graphically, decision trees can be interpreted as drawing rectangles in the decision space. The degree to which we can effectively draw the rectangles determines the classification accuracy.



Here is an example of a binary decision tree. A decision tree consists of nodes for the current feature to examine, and branches for the true/false result of comparing that feature

to a test case. A terminal node denotes the class. It is convention to have the false branch on the left and the true branch right.



When a new case is presented, we check to see if X is true or false. If X is false, we go to the left and conclude that the case is C1. If X is true, we move to the right and now test Y. If Y is false, we conclude C2. If Y is true, we conclude C1. In this example, two rules cover C1 while one rule covers C2.

Decision trees are simple to apply; however they are more complicated to build. We would like to find the smallest tree that perfectly classifies our training data. However, we often have to settle for less. It is easy to find a tree that will cover our sample without any errors, but more difficult to find a tree that performs well on test data. This means that you shouldn't be too impressed with yourself if the error on the training data is zero; the error on the test set may be much worse (nearest neighbor after all gets 0 training error).

Building The Tree

To build our tree, the idea is very simple. Start with some feature to test, say X. For now let's put off until later how we decided to pick X. Now, split up the set of cases into two sets, one where X is true and the other where X is false. If any of these sets contains cases entirely from one class, then make that branch a terminal node labeled with that class. Otherwise, repeat the process with the newly formed set(s). As an alternate method, nodes may become terminal when the size is below some threshold (say, 5), in which case we just assign that node to be the most prevalent class. This technique is necessary if we reach a point where there is no discriminating feature to separate the classes.

Ideally we would like to find a small tree; this will give us rules that are easy to understand and the performance will actually be better (more on this later). To do this, we want to split based upon the most *predictive* features first.

To determine the most predictive feature, a heuristic search is employed. This heuristic attempts to reduce the degree of randomness, or "impurity" of the current feature. We'll

apply the heuristic for all our feature tests, looking for the best one. For example, a feature that splits the data into two sets, where both sets have a 50% mixture of cases in C1 and C2, then the impurity or randomness is high. However, if we find a feature that splits the data into two sets where one set is 100% C1 and the other set is 100% C2 the the impurity or randomness is 0. We want to minimize the impurity to find the right feature to select.

A common heuristic to use is the entropy function. The entropy is defined as:

$$entropy(n) = -\sum_{c} p_c \log_2 p_c$$

The entropy of a particular state is the negative sum over all the classes of the probability of each class multiplied by the log of the probability. For example, let's say that we have:

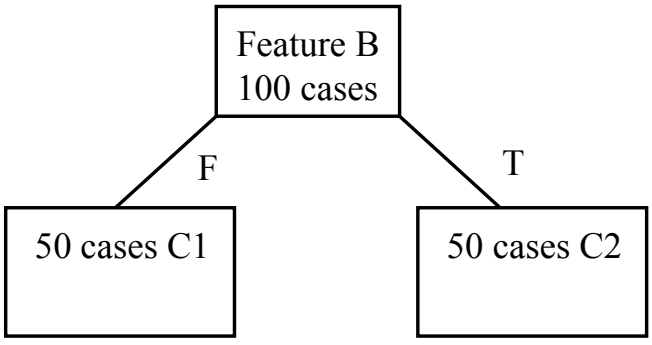
2 classes, C1 and C2
100 cases
50 cases are in each class
Thus the probability of each class, P1 and P2 are 0.5.
The entropy of this node = -[(0.5)(lg 0.5) + (0.5)(lg 0.5)] = 1

Our algorithm will pick the feature or test that reduces the entropy the greatest. This can be achieved by maximizing the following equation:

$$\Delta entropy(n) = entropy(n) - p_{left} entropy(n_{left}) - p_{right} entropy(n_{right})$$

The probabilities of branching left or right are simply the percentage of cases in node N that branch left or right. For all of our feature tests, we would calculate Delta-Entropy and pick the feature that has the greatest change in entropy.

For example, let's say that we are at a node with an entropy of 1 as calculated above. One feature test may result in 50% of the cases going left and 50% going right. Of the 50% going left, all are in C1. This means the entropy of Node(Left) = 0. Similarly, of the 50% going right, all are in C2. This means the entropy of Node(Right)=0. The change in entropy is then 1 if we selected this feature:

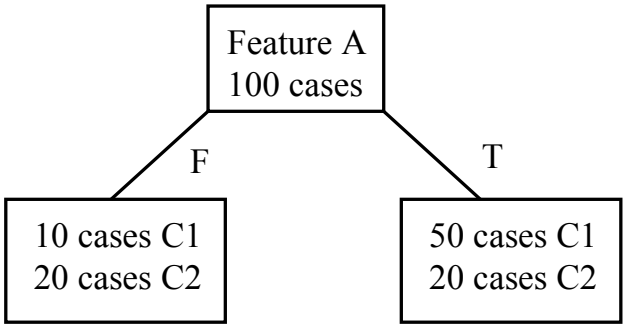


E=-[(1)*lg(1) + (0)*lg(0)
= 0 (actually undefined at lg(0))

E=-[(0)*lg(0) + (1)*lg(1)
= 0

$$\Delta entropy = 1 - (0.5)0 - (0.5)0 = 1$$

If we tried another feature it might result in a less clear separation:



E=-[(1/3)*lg(1/3) + (2/3)*lg(2/3) = 0.92

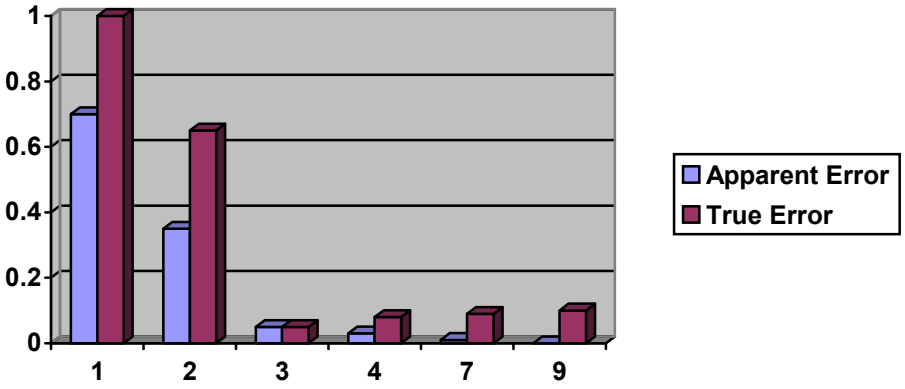
E=-[(5/7)*lg(5/7) + (2/7)*lg(2/7) = 0.86

$$\Delta entropy = 1 - (0.3)0.92 - (0.7)0.86 = 0.122$$

Note that there are many other heuristic functions in use; the gini function, LaPlace heuristic, and statistical measures have also been proposed.

Shrinking The Tree

Decision trees to seem to operate according to Ockham's Razor, which says that the most likely hypothesis tends to be the simplest one that is consistent with all observations. We see this show up with large trees vs. small trees. Smaller trees that are consistent with the data tend to perform better than the large trees.



# of Terminals vs. Error Rates (for Iris Data problem)

In the graph above, the apparent error is the error rate on the training examples. Initially, as the number of nodes in the tree is small, it is considerably lower than the true error rate. As we increase the tree size to 3, the apparent error and the true error are smallest. But then as we increase past three, the true error actually increases, while our apparent error decreases.

We would actually do better with a tree of size 3 than a tree of size 9 (the tree that gives us 0 error on the training examples). How do we find the subtree that yields the best performance?

The most straightforward method is to *prune* branches off the decision tree. Working backward from the bottom of an induced tree, the subtree starting at each nonterminal node is examined. If the error rate on the test cases improves by pruning it, then the subtree is removed. Although this technique has the subtle flaw of "indirectly training on the test cases" it performs well on large samples (say >1000 test cases).

There are several other methods to pruning trees; one technique involves lookahead during tree generation. Others involve heuristics to determine which branch to prune or using cross-validation to get a better estimate on good prunes.

The decision tree algorithm presented here is the basis for an algorithm named ID3. Ross Quinlan designed the use of the entropy function in 1979. More recent decision tree algorithms include C4.5 and C5.0, a commercial product.

Web demo:

http://www.cs.ualberta.ca/~aixplore/learning/DecisionTrees/

Rule Induction – Decision Lists

Decision trees have become one of the "standard" algorithms for machine learning in the AI community. However, the rules produced by decision trees are not only ordered (we can't jump in the middle of the tree to start the evaluation) but cover all classes. For large trees, these rules can still be extremely difficult for people to understand.

An easier form to understand is a direct set of rules, in disjunctive normal form:

1. if X then C1
2. if X and Y then C2
3. if NOT X and Z and Y then C3
4. if B then C2

In DNF, we have only AND's within each rule, but an OR of all rules.

Although relaxing the mutual exclusivity requirement of decision trees appears minor, developing a learning system for this representation is actually much more difficult. One immediate problem that arises is what to do if two rules fire for the same input, but predict different classes. The common procedure is to predict the most common class in this case. In general, we have problems if we just OR together all of our rules.

The problem with OR'ing together rules is that, surprisingly, we might end up lowering our performance! Consider if you have two rules and 1000 cases. Rule 1 is activated (covers) 100 cases, and is correct on 90 of them. Rule 2 also covers 100 cases, and it is correct on 90 of them. What happens when we OR these rules together? In the best case, the 90 correct cases are different, but the incorrect cases are identical. The accuracy is now (90+90) / (90 + 90 + 10) = 0.95. However, in the worst case the two rules are correct on the SAME cases but wrong on different cases. The combined classifier is now 90 / (90+10+10) = 0.82.

The end result is that we need to be careful inducing the rules, and that in some cases decision trees may be a better way to go.

CN2 Induction Algorithm

CN2 is one unordered rule induction algorithm designed by Peter Clark. There are many other algorithms; all operate under the same general principles. A somewhat simplified version is presented here.

The idea of CN2 and other rule induction algorithms is to search the space of decision rules from the general to the specific. We'll employ a beam search to determine what rules to generate; this search space is huge. If we only have 3 features, X, Y, and Z, then we could generate the following possible rules:

If X then…
If X and Y then…
If X and Y and Z then…
If X and Z then …
If Y then …
If Y and Z then …
If Z then…

The number of rules grows exponentially, 2^n-1. There are even more rules if we consider NOT X, NOT Y, etc. With features that are typically in the tens or hundreds, search through this space can be extremely difficult. Essentially we will be searching through this space of possible rules for the best rule on the training data.

There are three procedures in the algorithm:

CN2Unordered(allexamples, allclasses)

```
    Ruleset ← {}
    For each class in allclasses
        Generate rules by CN2ForOneClass(allexamples, class)
        Add rules to ruleset
    Return ruleset

CN2ForOneClass(examples, class)
    Rules ← {}
    Repeat
        Bestcond ← FindBestCondition(examples, class)
        If bestcond <> null then
            Add the rule "IF bestcond THEN PREDICT class"
            Remove from examples all cases in class covered by bestcond
    Until bestcond = null
    Return rules
```

In the decision tree algorithm, we essentially removed all examples covered by bestcond whether or not they were in a particular class. In the rule induction algorithm, it is important that we keep negative examples of the rule around so that future rules stand out from the negatives. We must remove the positive examples to prevent us from finding the same rule again. The task remains to implement the FindBestCond routine:

```
FindBestCondition(examples, class)
    MGC ← true   ' most general condition
    Star ← MGC
    Newstar ← {}
    Bestcond ← null
    While Star is not empty  (or loopcount < MAXCONJUNCTS)
        For each rule R in Star
            For each possible feature F
                R' ← specialization of Rule formed by adding F as an
                    Extra conjunct to Rule (i.e. Rule' = Rule AND F)
                    Removing null conditions (i.e. A AND NOT A)
                    Removing redundancies (i.e. A AND A)
                    And previously generated rules.
                If EntropyTest(R',class) better than
                EntropyTest(Bestcond, class)
                    Bestcond ← R'
                Add R' to Newstar
                If size(NewStar) > MAXRULESIZE then
                    Remove worst in Newstar
                    until Size=MAXRULESIZE
            Star ← Newstar
    Return Bestcond
```

Initially, FindBestCondition will start with Star containing only the default "true" rule. To this we add specializations by testing the rules with one feature. For example if our features are X, Y and Z, then on the first pass we would generate "IF X then Class", "IF Y then Class" and "IF Z then Class". All of these rules would be tested to see which has the lower entropy (i.e. performance on the training set; other metrics can also be used, such as the error rate using this rule).

Next, we keep track of the best rule so far and only remember the MAXRULESIZE best rules. If MAXRULESIZE = 2, then we might only keep the rules "IF X then Class" and "IF Y then Class". By limiting the rule size, we enforce a narrow "beam" in which we are searching through the rule space. To the top rules, we specialize them further, AND'ing each rule with all features. We would now have the rules:

IF X and Y then class
IF X and Z then class
IF Y and Z then class

The process repeats. The next loop we would generate rules with three conjuncts, until we run out of features or reach our MAXCONJUNCTS limit. This routine is quite compute intensive, especially for a large number of features and sample cases.

The CN2 algorithm has been tested on many sample machine learning tasks (heart disease test data, plant classification, etc.) In many cases it not only produced more readable rules, but also outperformed decision trees.