Decision Tree

1. *Recursive Greedy Decision Tree Learning*

Decision tree uses greedy algorithm to construct the tree, it keeps partitioning data until stopping criteria are met (instances in a node belong to the same class A.K.A the node is pure; no available splits can provide positive gain, or the minimum number of instances in a node is achieved).

The algorithm for making a tree is:

Top-down decision tree learning

```
MakeSubtree(set of training instances D)
    C = DetermineCandidateSplits(D)
    if stopping criteria met
        make a leaf node N
        determine class label/probabilities for N
    else
        make an internal node N
        S = FindBestSplit(D, C)
        for each outcome k of S
            D_k = subset of instances that have outcome k
            k^th child of N = MakeSubtree(D_k)
    return subtree rooted at N
```

My actual implementation is slightly different:

```
BuildTree(Data, m):
If all the instances in the data belong to the same class:
        Return Leaf(Data)
BestSplit, BestInfoGain = FindBestSplit(Data)
If number of instances in Data < m or BestInfoGain <= 0:
        Return Leaf(Data)
Else:
        N = MakeNode(BestSplit, Data)
        For each outcome of BestSplit:
                Dk = Subset of data that have outcome k
                Kth child of N = BuildTree(Dk, m)
```

The stopping criteria are: if all the instances in the data belong to the same class, or if the number of instances in the data is less than input value m, or no currently available split provide positive information gain.

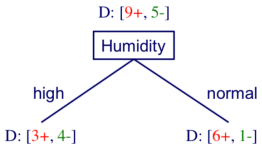
2. *Metric for Deciding Which Feature to split on*

The metric ID3 decision tree uses to choose split is information gain, given a current data set (or sub data set for a node in a tree), it is calculated by the sum of entropy of the class label subtracted by the sum of entropy of all the feature's labels:

$$\text{InfoGain}(D,S) = H_D(Y) - H_D(Y \mid S)$$

Information gain example

- What's the information gain of splitting on Humidity?



$$H_D(Y) = -\frac{9}{14}\log_2\left(\frac{9}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) = 0.940$$

$$H_D(Y \mid \text{high}) = -\frac{3}{7}\log_2\left(\frac{3}{7}\right) - \frac{4}{7}\log_2\left(\frac{4}{7}\right) = 0.985$$

$$H_D(Y \mid \text{normal}) = -\frac{6}{7}\log_2\left(\frac{6}{7}\right) - \frac{1}{7}\log_2\left(\frac{1}{7}\right) = 0.592$$

$$\begin{aligned}\text{InfoGain}(D,\text{Humidity}) &= H_D(Y) - H_D(Y \mid \text{Humidity}) \\ &= 0.940 - \left[\frac{7}{14}(0.985) + \frac{7}{14}(0.592)\right] \\ &= 0.151\end{aligned}$$

Where HD(Y|Humidity) is called conditional entropy:

$$H(Y \mid X) = \sum_{x \in \text{values}(X)} P(X = x)\, H(Y \mid X = x)$$

where

$$H(Y \mid X = x) = -\sum_{y \in \text{values}(Y)} P(Y = y \mid X = x)\, \log_2 P(Y = y \mid X = x)$$

3. *Problem with Information Gain*

Decision trees using information gain are biased towards tests with many outcomes (e.g. ask a question where the feature is nominal and have a lot of values).

To address this issue, C4.5 decision tree uses a splitting criterion called gain ratio. It's introduced to penalize tests with a lot of outcomes:

consider the potential information generated by splitting on S

$$\text{SplitInfo}(D,S) = -\sum_{k \in \text{ outcomes}(S)} \frac{|D_k|}{|D|}\log_2\left(\frac{D_k}{D}\right)$$

use this to normalize information gain

$$\text{GainRatio}(D,S) = \frac{\text{InfoGain}(D,S)}{\text{SplitInfo}(D,S)}$$

S is the feature the question intends to split on, |Dk| is the number of instances in the training data that have feature value = k , |D| is the number of training data. When the feature has a lot of feature values, the SplitInfo (also called split entropy) is going to be large, and the gain will be penalized.

4. Problem with the greedy algorithm that is used to generate decision trees

Because decision tree uses a recursive greedy algorithm to generate decision trees, the tree oftentimes overfit the data. In other words, the tree will be a little too specific to the training set and fail to generalize for unseen test data. A more formal way to define overfitting is:

Overfitting

- consider error of model h over
  - training data: $error_D(h)$
  - entire distribution of data: $error(h)$

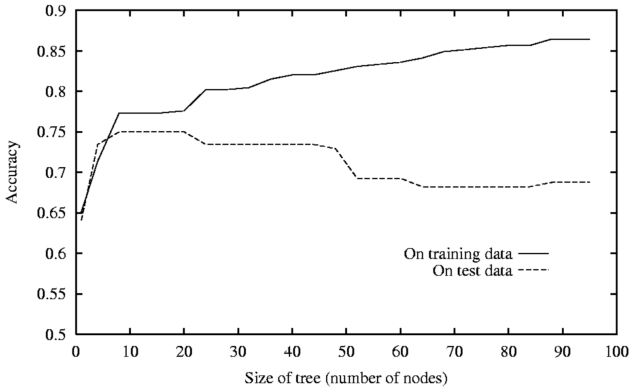
- model $h \in H$ *overfits* the training data if there is an alternative model $h' \in H$ such that

$$error(h) > error(h')$$

$$error_D(h) < error_D(h')$$

Overfitting will cause accuracy to be high on training set while low on test set:

Overfitting in decision trees



To address this problem, we can either choose to stop early when building the tree or prune back after finished building the tree. Quite often, pruning is preferred since it is hard to tell when a tree algorithm should stop because it is impossible to tell if the addition of a single extra node will dramatically decrease error.

The simplest and most intuitive way of pruning is **reduced error pruning.** For each node **starting at the leaves**, replace the node with its majority class. If the resulting decision does not affect the accuracy on validation set, keep the change. Keep doing this for every node.

Another approach is called **cost complexity pruning**, the pruning procedure is defined as:

Cost complexity pruning generates a series of trees $T_0 \dots T_m$ where T_0 is the initial tree and T_m is the root alone. At step i, the tree is created by removing a subtree from tree $i-1$ and replacing it with a leaf node with value chosen as in the tree building algorithm. The subtree that is removed is chosen as follows:

1. Define the error rate of tree T over data set S as $\mathrm{err}(T,S)$.
2. The subtree that minimizes $\frac{\mathrm{err}(\mathrm{prune}(T,t),S) - \mathrm{err}(T,S)}{|\mathrm{leaves}(T)| - |\mathrm{leaves}(\mathrm{prune}(T,t))|}$ is chosen for removal.

The function $\mathrm{prune}(T,t)$ defines the tree gotten by pruning the subtrees t from the tree T. Once the series of trees has been created, the best tree is chosen by generalized accuracy as measured by a training set or cross-validation.

5. Random Forests

One problem with decision trees is that they don't predict as well as other machine learning algorithms, this is mostly due to the greedy nature of the tree construction algorithm, small changes to the input training data might cause the tree to look very differently due to the hierarchical nature of the top-down tree growing process. We say that decision trees have **high variance**.

One way to address this problem is that we train multiple trees on different subsets of the data with different subsets of the features, then compute the ensemble:

$$f(\mathbf{x}) = \sum_{m=1}^{M} \frac{1}{M} f_m(\mathbf{x})$$

Where fm is the m'th tree. This technique is called "bagging", which stands for "bootstrap aggregating".

The reason we want to traing on only a subset of training features is that simply rerunning the training algorithm on subset of data can result in highly correlated predicators, which limits the amount of variance reduction that is possible.

Experiment: Random forest works pretty compared to just a single decision tree. Using a decision tree (stopping criteria include instances under a node less than 30) gives us accuracy of 0.7575757575757576. Multiple of run of the random forest give us accuracy of 0.8333333333333334, 0.8106060606060606, 0.8181818181818182, 0.8106060606060606, 0.8333333333333334, whose average is 0.8212121, which is pretty good for a simple implementation of the random forest.