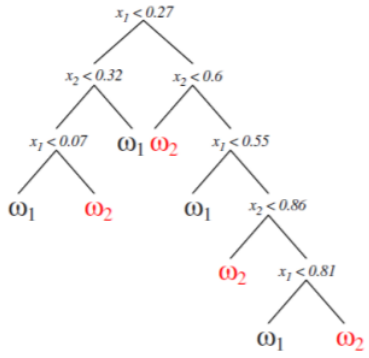
1 Decision Trees

A **decision tree** is a type of classifier which classifies things by repeatedly posing questions of the form, "Is feature x_i greater than value v?" You can visualize such a thing as a tree, where each node consists of a split-feature, split-value pair (x_i, v) and the leaves are possible classes. Here's a picture:



Note that a class may appear multiple times at the leaves of the decision tree. Picking a split-feature, split-value pair essentially *carves up* the feature space into chunks, and you can imagine that having enough of these pairs lets you create an arbitrarily complex classifier that can perfectly overfit any training set (unless two training points of different classes coincide).

When we consider the task of building this tree, we want to ask ourselves a couple of things:

- How do we pick the split-feature, split-value pairs?
- How do we know when to stop growing the tree?

Let's address the first point. Intuitively, when we pick a feature to split on, we want choose the one which maximizes *how sure we are of the classes of the two resulting chunks*. If we had a bunch of data points and the barrier represented by the line $x_i = v$ perfectly split them such that all the instances of the positive class were on one side and all the instances of the negative class were on the other, then the split (x_i, v) is a pretty good one.

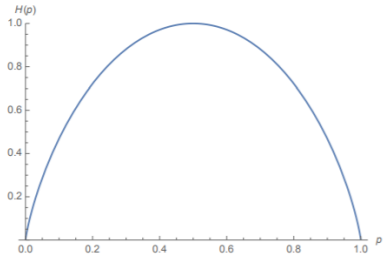
Now, to concretely talk about the idea of "how sure" we are, we'll use the ideas of **surprise** and **entropy**. The numerical surprise of observing that a random variable X takes on value k is:

$$\log \frac{1}{P(X=k)} = -\log P(X=k)$$

The entropy of a random variable $H(X)$ is:

$$\begin{aligned} H(X) &= E[\text{surprise}] \\ &= E_X[\log \frac{1}{P(X=k)}] \\ &= \sum_k P(X=k) \log \frac{1}{P(X=k)} \end{aligned}$$

Here's a graph of entropy vs. $P(X=k)$ for a *binary* discrete random variable X (as in, $k \in \{0,1\}$):



Observe that (1) it's convex and (2) it's maximized when the probability distribution is *even* with respect to the outcomes. That is to say, a coin that is completely fair ($P(X=0) = P(X=1) = 0.5$) has more entropy than a coin that is biased. This is because we are less sure of the outcome of the fair coin than the biased coin *overall*. Even though we are more surprised when the biased coin comes up as its more unlikely outcome, the way that entropy is defined gives a higher uncertainty score to the fair coin. In general, a random variable has more entropy when the distribution of its outcomes is closer to uniform and less entropy when the distribution is highly skewed to one outcome.

An assumption

By the way, it's strange that entropy is defined for random variables when there are no random variables in a training set. To address this, we make the assumption that a random variable X on the possible classes is representative of our training set. The distribution of X is defined by our training points (x_i, y_i), where $P(X = c_j) = \frac{n_j}{n}$ with n_j being the number of training points whose y_i is c_j.

So now we know that when we choose a split-feature, split-value pair, we want to *reduce* entropy in some way. First, let $Y_{f_i,v}$ be an indicator function that is 1 for a data point if its feature f_i is less than v. Note that Y is not a random variable, though we will use it in some notations that make it appear like one. When we consider splitting one set of points represented by random variable X with split-key (f_i, v), there are a few entropies we should consider.

- $H(X)$
- $H(X|Y=1)$. That is, the entropy of the set of points whose f_i is less than v.
- $H(X|Y=0)$. That is, the entropy of the set of points whose f_i is not less than v.

In some ways $H(X)$ is non-negotiable: we start with the set of points represented by X, and they have some entropy, and now we wish to carve up those points in a way to minimize the entropy remaining. Thus, the thing we want to minimize is a weighted average of the two sides of the split, where the weights are (proportional to) the sizes of two sides:

$$\begin{aligned}\textbf{Minimize } & P(X < v)H(X|Y=1) + P(X \geq v)H(X|Y=0) \\ = & P(Y=1)H(X|Y=1) + P(Y=0)H(X|Y=0)\end{aligned}$$

An equivalent way of seeing this is that we want to maximize the information we've learned, which represented by how much $H(X)$ gets reduced after learning Y:

$$\textbf{Maximize } H(X) - \underbrace{P(Y=1)H(X|Y=1) + P(Y=0)H(X|Y=0)}_{H(X|Y)}$$

.

For general indicator functions Y, we want to maximize something called the **mutual information** $I(X;Y)$. This depends on the **conditional entropy** of X given Y:

$$\begin{aligned}H(X|Y=y) &= \sum_k P(X=k|Y=y) \log \frac{1}{P(X=k|Y=y)} \\ H(X|Y) &= \sum_y P(Y=y)H(X|Y=y) \\ I(X;Y) &= H(X) - H(X|Y)\end{aligned}$$

Note that I(X;Y) is nonnegative and it's zero only if the resulting sides of the split have the same distribution of classes as the original set of points. Let's say you were using a decision tree to classify emails as spam and ham. The precedng statement says, you gain no information if you take a set of (20 ham, 10 spam) and split it on some feature to give you sets of (12 ham, 6 spam); (8 ham, 4 spam) because the empirical distribution of those two resulting sets is equal to the original one.

Now that we have addressed how to pick splits, let's talk about when to stop growing the tree. We know that decision trees are powerful tools because they can be used to represent any arbitrarily complex decision boundaries. Thus, it is very easy to overfit them. One way to do this is to keep carving up the feature space until the leaves are entirely pure (as in, they only contain points of a single class). As a result, when training decision trees, we always want to keep in mind a couple of heuristics for stopping early:

- Limited depth: keep track of the depth of each split, and don't go past some depth t
- Information gain criteria: stop carving once your splits don't reward you with enough information (i.e., set a threshold for $I(X;Y)$)
- Pruning: This isn't a method for stopping early. Rather it's the strategy of growing your tree out as far as you can, and then *combining* splits back together if doing so reduces your validation error.

All of these thresholds can be tuned with cross-validation.

2 Random Forests

Another way to combat overfitting is the use of **random forests**. A random forest is a set of decision trees whose outputs are averaged together in some way (usually majority vote) to produce the final answer to a classification problem. In addition, each tree of the forest will have some elements of randomness to them. Let's think about why this randomness is necessary.

Imagine you took the same set of training points and you built k decision trees on them with the method of choosing splits described in the previous section. What could you say about these k trees? Answer: They would all be the exact same trees in terms of what splits are chosen! Why: because there is only one best split of a set of training points, and if we use the same algorithm on the same set of training points, we'll get the same result.

There are two options: (1) change our algorithm for each tree, or (2) change our training points. The algorithm is pretty good, so let's change the training points. In particular, we'll use the method of **bagging**: for each tree, we sample *with replacement* some constant number of training points to "bag" and use for training. This will avoid the problem of having identical decision trees...

...Or will it? Imagine that you had an *extremely* predictive feature, like the appearance of the word "viagra" for classifying spam, for your classification task. Then, even if you took a random subsample of training points, your k trees would still be very similar, most likely choosing to split on the same features. Thus, the randomness of our trees will come from:

- bagging: random subsample of training points per tree
- enforcing that a random subsample of *features* be used for each tree

Both the size of the random subsample of training points and the number of features per tree are hyperparameters intended to be tuned through cross-validation.

Remember why this is a good idea: One decision tree by itself is prone to overfitting to the training set. However, if we have a bunch of them that are diverse and uncorrelated, then we can be more sure that their average is closer to the true classifier.

3 Boosting

In random forests, we treated each member of the forest equally, taking a majority vote or an average over their outputs. The idea of **boosting** is to do this more carefully. Specifically, we want to:

- Weight each of the simple classifiers (i.e., trees in a random forest)
- Give higher weight to the "better" ones

What is a "better" member of the forest? The idea is: to improve our classifier, we want to classify more points correctly than we currently are. Some of our trees will be classifying things correctly that other trees are currently getting wrong, and we want to bring out that difference in our trees. The key ideas in doing this are as follows:

1. Give each training point a weight
2. Run the classifiers on the dataset, taking the average, and increase the weight of the misclassified points.

Algorithm: **AdaBoost** (Adaptive Boosting)

1. Initialize all weights of the training points to $\frac{1}{n}$.
2. Repeat as m goes from $1 \dots M$:
   (a) Build a classifier (however you wish).
   (b) Compute the weighted error $e_m = \frac{\sum_{i \text{ misclassified}} w_i}{\sum_i w_i}$.
   (c) Re-weight the training points.

$$w_i \leftarrow w_i * \begin{cases} \sqrt{\frac{1-e_m}{e_m}} \text{ if misclassified} \\ \sqrt{\frac{e_m}{1-e_m}} \text{ otherwise} \end{cases}$$