# CS 486 — Lecture 8: Decision Trees

## 1 Stopping Criterion

### 1.1 No More Features Left

- What do we do if we run out of features during our decision tree construction?
- This occurs if some data in the dataset have the same features but their final classes differ.
- This is due to noise or some factor that we did not observe/track.
- As such, we can't make a deterministic choice just based on the dataset.
- One way we can deal with it is use a majority decision.
- The other possibility is to just randomly make a decision via probabilistic decisions (ie: if 2 data points are "yes", and 1 is "no", then $\frac{2}{3}$ chance to select "yes").

### 1.2 No More Examples Left

- Occurs if we hit a node with no entries with classes in the dataset.
- Since we never hit this combination of data, we don't know how to predict this!
- How can we handle this case? One way is to use a similar result; we can use the parent node to make a decision. We are likely to not be able to make a deterministic choice in this case so like earlier, we can use probability or majority to decide on what option to choose.

## 2 Entropy

- We can write a decision tree learner as such (in noisy/indecisive cases we return a majority):

```
def learner(examples, features):
    if all examples are in the same class:
        return class label
    else if no features left:
        return majority decision
    else if no examples left:
        return majority decision at the parent node
    else:
        choose a feature f
        for each value v of feature f:
            build edge with label v
            build subtree using examples where the value of f is v
```

- We can see our 3 base cases and the recursive case (the `else` case).
- So how do we choose which feature to test at each step?
- We want to minimize the number of tests to create a shallow/small tree.
- Since finding the global optimal order of testing features is difficult (NP), we use a greedy search instead.
- At each step, test a feature that makes the biggest difference to the classification.
- We want to make a decision ASAP. So, find a feature that reduces uncertainty as much as possible.

- The information content of a feature is just the uncertainty before testing the feature minus the uncertainty after testing said feature.
- We quantify uncertainty using the idea of entropy.
- Given a distribution $P(c_1), \ldots P(c_k)$ over $k$ outcomes $c_1, \ldots, c_k$, the entropy of the distribution is

$$I(P(c_1), \ldots P(c_k)) = -\sum_{i=1}^{k} P(c_i) \log_2(P(c_i))$$

- For example, the distribution of (0.5, 0.5) is 1 bit of uncertainty.
- Or the distribution of (0.01, 0.99) is approximately 0.08 bits of uncertainty, which is low.
- So let's consider a distribution of $(p, 1-p)$:
  - The max entropy is 1 when $p = 0.5$.
  - The min entropy is 0 when $p = 0$ and $p = 1$.
  - Note that in the case $p = 0$, $\log_2(0)$ is undefined...but we actually state that $I(0, 1) = I(1, 0) = 0$ by definition.
  - This means that the plot of this distribution in regards to $p$ forms a semicircle.

# 3 Calculating Expected Information Gain

- A feature has $k$ values $v_1, \ldots, v_k$.
- Before we test the feature, we have $p$ positive and $n$ negative examples.
- After testing the feature, for each value $v_i$ of the feature, we have $p_i$ positive and $n_i$ negative examples.
- So given all of this, how do we measure the changing uncertainty?
- We call this the expected information gain.
- The entropy before testing the feature is binary, with a distribution of $H_{\text{before}} = I(\frac{p}{p+n}, \frac{n}{p+n})$.
- After testing the feature, for each branch, we can calculate the corresponding entropy as before.
- So, we get that the expected entropy after testing the feature is $\sum_{i=1}^{k} \frac{p_i+n_i}{p+n} \times I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p+i+n_i})$.
- That is, the expected information gain, or entropy reduction, is $H_{\text{before}} - H_{\text{after}}$.
- We can generalize this to more than 2 classes.

# 4 Handling Real-Valued Features

- So what do we do with real values? For example, temperature by floats.
- So how do we handle discrete features? We can allow binary splits only, but we can also allow multi-way splits.
- This makes the tree more complex, but also makes it shallower.
- Info gain prefers a variable with a larger domain, so this may add some bias to some results.
- Note we can always convert a tree with multi-way splits into a binary split tree...

- Binary-split-only trees are deeper but simpler and more compact.
- One way to convert a real feature to a discrete feature is to group into ranges.
- This is easy to do but you may lose valuable information, may make the tree more complex, and is a bit hard to do without any information.
- Another option is to always do binary tests and dynamically pick a split point.
- A general algorithm to split is to look at points and see if their class changes. If it does, then it might be worth splitting there (ie: if one has "yes" and the other "no" then it might be worth splitting).

# 5 Overfitting

- Our decision tree algorithm is a "perfectionist" — it will keep growing the tree until it *perfectly* fits our training set.
- This could mean it cannot generalize well!
- One potential solution is to *not* grow a full tree. We could require a minimum number of examples per node, require a threshold of information gain, have a max depth, post-prune a tree, etc.
- Post-pruning is generally a better idea; sometimes there are multiple features which, when working together, are informative, but the feature alone is not useful.
- For example, suppose the function is on 2 bits, and it applies the XOR. Knowing just one bit tells us nothing, but knowing two bits gives a perfect result.
- If we stop early, then we might not take advantage of looking at each bit separately as it may look useless.
- Meanwhile, with post-pruning, we may have properly split by both bits and we can then see later that this branch is good and we don't prune it.