1 Decision Trees (cont.)

1.1 Writing the code

Stephanie wrote code to support the decision tree, making separate classes for the tree node and the tree itself. She chose to use recursive methods in the node class to execute all the necessary logic.

- DecisionTreeNode class
  - Stores the indices of the training points that “made it” to this node.
  - Stores the number of training points in each class that “made it” to this node.
  - Stores a reference to the full set of data and class values.
  - Stores the entropy for the node.
  - Stores the class of the node (If this is a leaf node, then this is what will be used to do the classification).
  - Stores error of the node (we will talk about this later – it becomes important when we prune).
  - Has a method named become decision node which generates child nodes. It calls helper methods find split point and compute outinfo of this split point to find the feature and threshold that create the best split value for the tree. This will be the value that produces the information gain. (Note that I did not make this method recursive, because it is designed to be used to build either 1R trees or trees with more depth.
  - Has a method named classify which takes in a data point. If this is a leaf node, it return the node’s class. Otherwise, it calls the method recursively on the appropriate child node.

The OneRuleDecisionTree class has a DecisionTreeNode as its root. To build a 1R tree, you pass in data, it creates a node, and then makes it become a decision node. The classify method

1.2 Simple Tree with more than 1 decision

There is no reason we need to stop with just one decision node and 2 leaf nodes. We could attempt to turn each leaf node into a decision node, expanding our tree as we go.
We have a helper method for DecisionTreeNode called `become_decision_node` that finds the best possible
decision and makes the child nodes, if possible. If those children are made, then the left child will
be referenced by `self.kids[0]` and the right child will be referenced by `self.kids[1]`.

To build a tree with as many branches as we need to get pure nodes (nodes with an entropy of 0),
we can use this recursive method:

```python
def make_simple_tree(self):
    if self.my_entropy == 0.0:
        return
    self.become_decision_node()
    for kid in self.kids: # this will do nothing if there were no kids created.
        kid.make_simple_tree()
```

While the above procedure creates a perfectly fine decision tree, it does not balance generalization
with specificity. The best decision tree algorithms use additional information to choose when to
subdivide a node, and they prune branches from trees once they are completed. Otherwise, every
single detail in the training data will contribute to the tree, whether or not it is relevant to the
task. When the decision tree (or any classifier) captures all the noise of the data, instead of the
underlying relationship between the classes and the features, it is called **over-fitting**.

It is important to note two attributes of the process:

- The decision about whether to build a node, and what question to ask is part of a local
  search process, or a **greedy search** process. That means the tree is built through a series
  of decisions that do not take into account global information about the structure of the tree,
  but consider only the immediate value of a question on the components of the training set
  going through that branch.

- The decision process itself is heuristic, which means there is no provably correct method of
  picking a decision tree node. Experience and experimentation have yielded an algorithm that
  generally performs well within a greedy search process. However, the free lunch theorem sug-
  gests that certain parameters of the tree building process should be evaluated when building
  a tree for a particular task.

### 1.3 Pruning

Pruning decision trees is the trickiest part of building useful trees. Almost all data sets contain some
noise, so creating a perfect decision may be either impossible or undesirable from a generalization
standpoint. It may be better to stop subdividing a particular branch of the tree well before all of
the attributes have been used or the branch is pure. Pruning uses two different methods of reducing
the complexity of trees.

- Subtree replacement: eliminate a subtree and replace it with a single leaf node
- Subtree raising: replace a parent node by one of its subtrees

Subtree replacement, which is basically a traditional tree pruning operation, is the most common
and effective method of decision tree pruning. The basic idea is to prune away leaves if their parent
node would be “good enough”.. The approach is to look at the expected error rate of a subtree
and compare it to the expected error rate of replacing the subtree with a single leaf node. If the expected error rate of the subtree is not significantly better than the error rate of the single leaf node, then prune away the leaves.

We can estimate the error rate of a node by using the formula for occurrences of a class resulting from a Bernoulli process. A Bernoulli process models coin flipping multiple times, according to some (unknown, true) probability. We want to use our knowledge of the observed error rate (the fraction of data points in the node that are not in the majority class) to estimate the true error rate. To do that, we find the upper bound of a confidence interval around it. Given a confidence limit $c$, we can use a lookup table to find the $z$-value that corresponds to it. For example, a 25% confidence limit is a $z$-value of 0.69, or 0.69 standard deviations. Because we end up using training data to estimate the expected error rate (not a great idea) we have to use a pessimistic estimate of the error, which is basically an upper bound on the error (with $c$ confidence). If $f$ is the error rate, $N$ is the number of samples, and $z$ is the $z$-value corresponding to our chosen confidence, the upper bound on the true error is given by (1).

$$e = f + rac{z^2}{2N} + z\sqrt{f \frac{f}{N} - f^2 \frac{N}{4N^2}}$$

(1)

If a node has 5 examples of class A and 14 examples of class B, then replacing the decision node by a leaf that guesses B results in 5 errors, or an error rate of $f = 5/19$. Using a confidence of 25%, or $z = 0.69$, the according to (1), the upper bound on the error at the node is 0.338. We can then calculate the upper bound of the error on each child node (which should be a leaf), combine the estimates using a weighted average, and compare the result to the estimated error of the node itself. If the combined error of the leaves is less than the error at the node, then we do not prune away the children. If the error of the children is greater than the estimated error of the parent, however, then we can prune away the children and replace the node with a leaf. We work from the bottom of the tree (the leaves) up, pruning as we go. (Note that we do not prune away entire subtrees – just leaves)

Pruning is an essential part of improving the performance of decisions on independent test data. Pruning can also generate much simpler subtrees, improving both the ability of people to understand the rules and the overall speed of the classifier.

What you need to know to code it up

- Add code so that when you create a new node, you add a field called my_error, which is the upper bound on the true error. Compute it accordion to equation (1) with $f$ as the error rate estimate from the node, $N$ as the number of training points that make to that node, and $z = 0.69$.

- Write a bottom-up recursive routine that prunes children (that are leaves) if their weighted average error (the my_error field) is higher than the error of the parent. This may be done with an additional set of training data that was not used to build the tree.

Here is the code for the pruning algorithm. It will be a DecisionTree node method.

```python
def prune_tree(self):
    if len(self.kids) == 0:
        return # this is a leaf node, no kids to prune
```
```python
self.kids[0].prune_tree()
self.kids[1].prune_tree()
if not self.kids[0].is_leaf() or not self.kids[1].is_leaf():
    return  # my kids aren't leaves, so I am not a candidate for removal
# compute the weighted average of the kid's error.
kids_errors = 0.0
for kid in self.kids:
    kids_errors += kid.my_error * float(len(kid.my_indices))/float(len(self.my_indices))
if self.my_error < kids_errors:
    # if my error is smaller than the weighted error of my kids, then
    # I will prune them.
    self.kids = []
```

### 1.4 Subtleties

#### 1.4.1 Too many branches

One problem with using information gain directly is that it tends to prefer decision trees that subdivide the task into many small pieces. A method often used to counter this tendency is to express the information gain as the ratio of the node’s information gain to the entropy of how the data gets split by the question (intrinsic information). The entropy of the data split is simply the information content of the subdivision of the data into the branches regardless of class. A question that subdivides all of the incoming training samples into equal parts has a high entropy. A question that sends a large number of samples into one branch and few into another has a low entropy. Using the ratio of information gain to intrinsic information favors nodes that produce fewer divisions, but balances that with the ability of the question to subdivide the task in a useful manner.

The intrinsic information in the 3-way split is $E(6, 7, 7) = 1.718$. The intrinsic information in the 2-way split is $E(10, 10) = 1$. Therefore, even if the two questions produced similar information gains, the two-way split gives a smaller denominator and should be preferred because it gives a larger information gain ratio.

In some cases the gain ratio overcompensates, so a heuristic is to select the node with the largest gain ratio, so long as its information gain is at least as big as the average information gain for all nodes considered. That way a node with a small information gain, but a large gain ratio, does not dominate over a node that has a higher than average information gain, but not as equal a division of the data.

**Incorporating the subtlety:** Instead of using the information gain in our algorithms, we may want to use the gain ratio.

#### 1.4.2 Missing values

Missing values can present problems when building a decision tree. As noted earlier, one solution to missing attributes is to send the instance down two branches and then combine the results of the resulting leaves using a weighted average. It is possible to do the same thing when building the decision tree by using non-integer branch weights when calculating the information gain for a node. For sub-trees after a split decision, the training sample still contributes via its likelihood of going down that branch.
These notes were adapted from those of Bruce Maxwell.