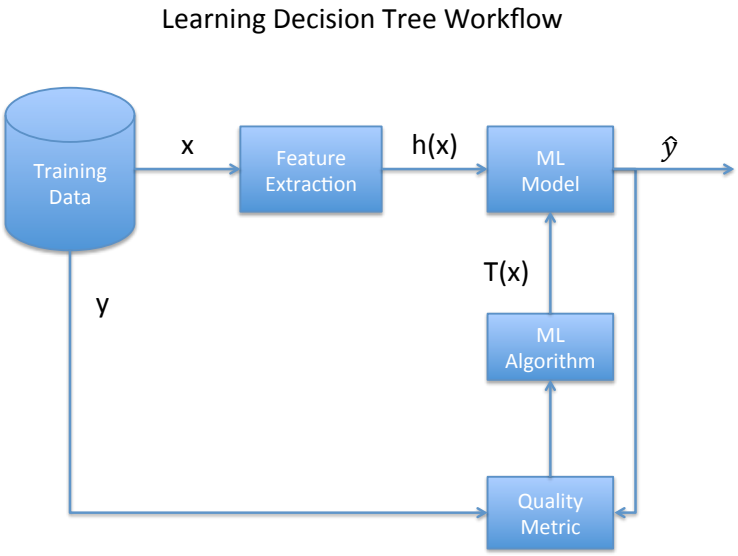
Classification Week 3 - Decision Trees

Intuition behind decision trees – Predicting Loan Defaults

Given an input x_i, which is a set of answers to a loan application, then the model that traverses the decision tree is $T(x_i)$ and it produces the predicted class of the loan application, $\hat{y}_i$, either safe or risky.

Task of learning decision trees from data



The data might look like this;

h1(x) credit	h2(x) term yrs	h3(x) income	y loan status
excellent	3	high	safe
fair	5	low	risky
fair	3	high	safe
poor	5	high	safe
excellent	3	low	safe
fair	5	low	safe
poor	3	high	risky
poor	5	low	risky
fair	3	high	safe

The goal is to learn a decision tree from these input features. Generally, given N observations $(\vec{x}_i, y_i)$ we want to optimize a quality metric on the training data to find the best decision tree T(x). Our quality metric will be classification error;

$$classification\ error = \frac{count(incorrect)}{number\ of\ inputs}$$

The best possible value is 0.0
The worst possible value is 1.0

This is the complement of classification accuracy.

This is an NP-hard problem; there are an exponentially large number of possible trees. So instead of generating every single tree and choosing the best one (based on classification accuracy), we choose the best partial tree at each step.

Recursive greedy algorithm

- Step 1: Start with an empty tree with a single node that contains all the data. This is the root node.
- Step 2: Select a feature on which we will split the data, create new child nodes with data filtered according to the split feature values.
- For each split node
    a. Step 3: if nothing more to do, make predictions
    b. Step 4: otherwise, goto step 2 (recursively) and continue splitting data.

In the diagram below, we can see the root node and the first split. Credit rating was chosen as the first split feature, so the nodes have data separated according to the 3 credit rating classes. Here we can see that the node where credit = excellent has only loan status of safe. So in step 3 we would be done with that node, we could predict safe. However, the other two nodes have a mix of loan status values, so we may want to split them again.

h1(x) credit	h2(x) term yrs	h3(x) income	y loan status
excellent	3	high	safe
fair	5	low	risky
fair	3	high	safe
poor	5	high	safe
excellent	3	low	safe
fair	5	low	safe
poor	3	high	risky
poor	5	low	risky
fair	3	high	safe

credit = excellent

h1(x) credit	h2(x) term yrs	h3(x) income	y loan status
excellent	3	high	safe
excellent	3	low	safe

credit = fair

h1(x) credit	h2(x) term yrs	h3(x) income	y loan status
fair	5	low	risky
fair	3	high	safe
fair	5	low	safe
fair	3	high	safe

credit = poor

h1(x) credit	h2(x) term yrs	h3(x) income	y loan status
poor	5	high	safe
poor	3	high	risky
poor	5	low	risky

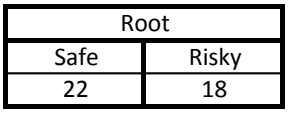
The key things we need to make clear

- What to split on at a node
- When to stop the recursion on a node

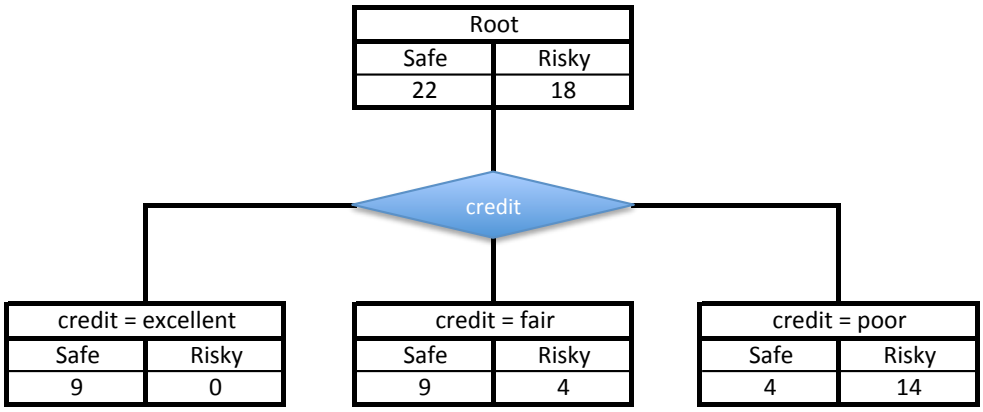
Learning a decision stump

A decision stump is a single node in the decision tree. This can also be thought of as a one level decision tree. The overall task turns out to be learning on each stump.

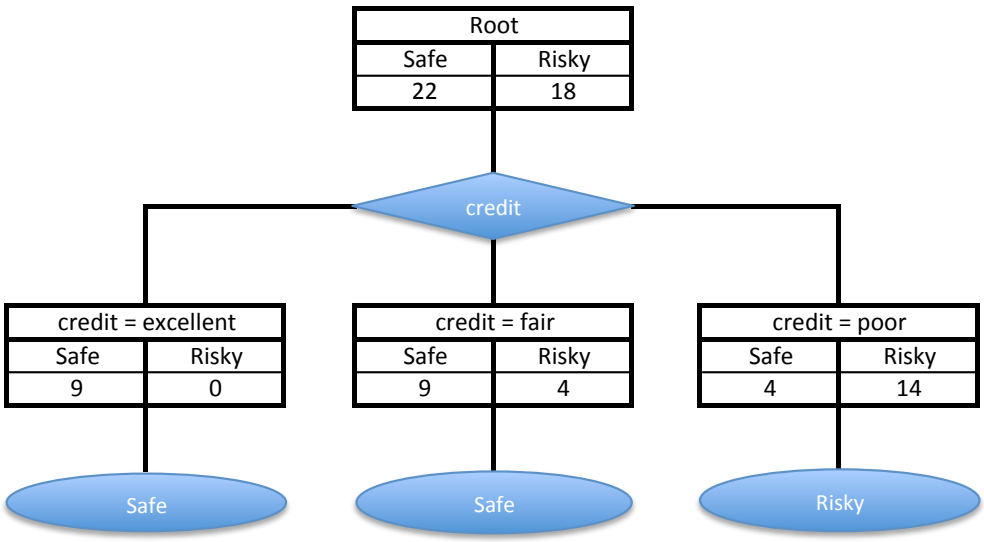
If we are given N = 40 data points of which 22 are classified safe and 18 are risky, then we can say our root node looks like this;



Now is splitting on credit rating creates a set of intermediate nodes, then the decision tree might look like this;

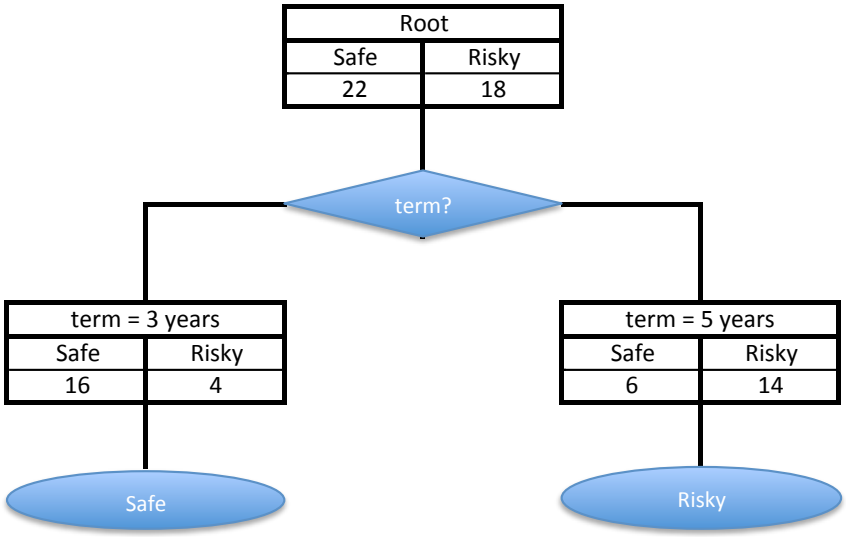


At this point, we can choose to make a prediction of the class of the loan at each intermediate node. We would do this using a majority value prediction; whichever class has more then that is our prediction. In the above case, the predictions would look like;



Selecting best feature to split on

In the prior section, we showed what splitting on credit would like like. However, there are other features we could have chose to split on. For instance, we could have chosen to split on the term of the loan in years. Such a split might have looked like this;



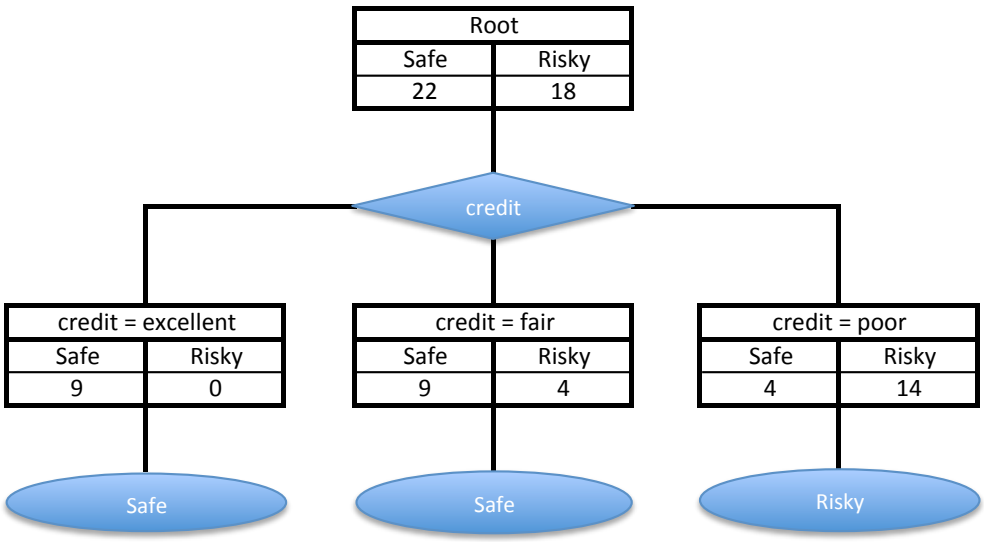
So the question then is, which is better to do that split on, credit rating or the term of the loan? The short answer is that we would try both splits and compare their classification error, then choose the one with the lowest classification error.

So we can look at the classification error in our first node. In the root there are 22 safe and 18 risky, so we would, based on majority value prediction, predict safe for the 40 loans at the node. Of course, we know that 18 were in fact risky, so those are wrongly classified by the majority value prediction. So our classification error at the root is $\frac{18}{40} = 0.45$. We only had two choices, so this is a very large error (with random data, we would only expect a 0.50 classification error and we are very close to that).

But this does give us our general procedure;

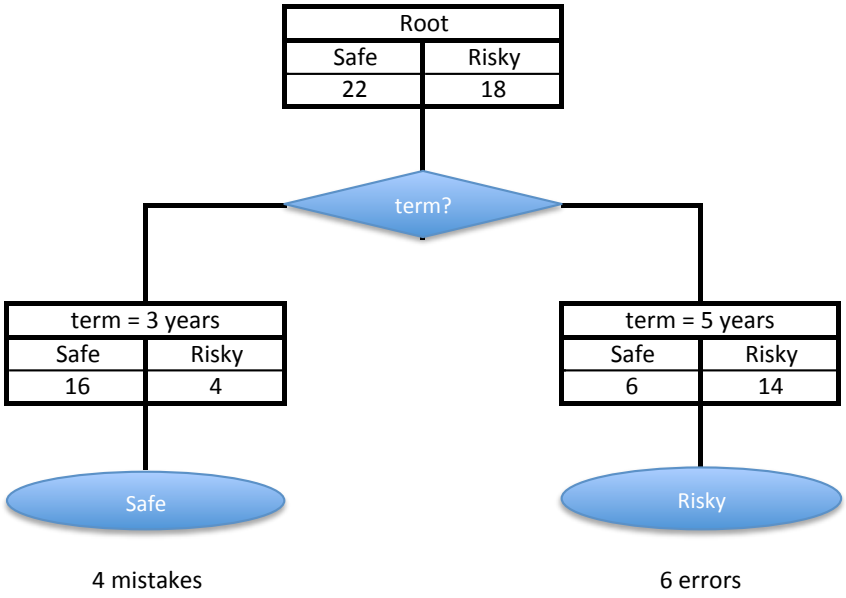
1. predicted y = class of majority in the node
2. classification error = non-majority class inputs / total inputs at node.

Now if we look at the credit history split we find 8 mistakes;



And so the classification error for this split is (0+4+4)/40 = 0.20. This is much better than at the root node. So without looking at the other features we could have split on, splitting on credit history improves our predictions, so we know we would likely do this split even if the others were poor.

But if we do look at splitting on term;



So the classification error if we split on term is (4+6)/40 = 0.25, which is a higher error than if we split on credit history. So we would do better to split on the credit history rather than the term of the loan if we did not intend to go any further.

A greedy algorithm will make this choice at each node and keep it. So the greedy choice at each node is;

At each node with a subset of data M;

- for each feature $h_j(x)$
  - Split the data of M according to the feature $h_j(x)$
  - Compute the classification error of the split
- Choose the feature split $h_*(x)$ with the lowest classification error

For each of these nodes we have to decide if we do another level of recursion (we learn another decision stump) or do we stop.

When to stop recursing

This turns out to be very easy. At any node we stop

- If all y values agree. For instance, in the credit history stump, the credit = excellent node has only loan status = safe, so there is not reason to do another split.
- If we run out of features. Clearly we can't split if there a no remaining features to split on. (in otherwords, we would never split on a feature we have already split on).

Once we are done, the overall classification error for the decision tree is calculated by counting all the mistaken classifications at all the leaf nodes and dividing this by the total number of inputs.

Decision Tree Classification Error = total mistakes at leaf nodes / total inputs.

Making predictions with decision trees

We can do this with a recursive method:

predict(tree_node, input)

- if tree_node is leaf
  - return majority class of inputs in leaf
- else
  - next_node = child node of tree_node whose feature value agrees with the input
  - return predict(next_node, input)

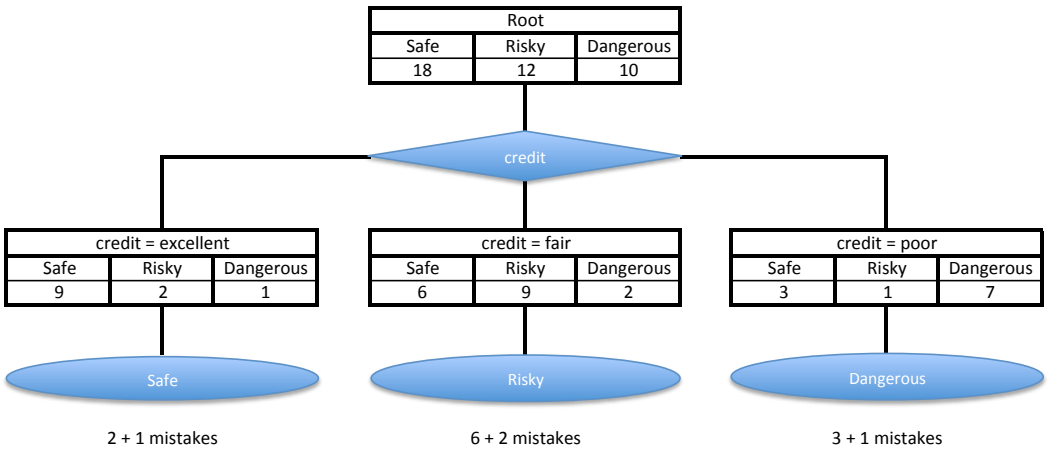
We have some input for which we want to make a predicted output. We use the root node and that input in the initial call;

prediction = predict(root, input);

Multiclass classification with decision trees

The prior example involved binary classification; a loan application was classified as safe or risky. However, the exact same algorithm works for multiple class outputs, we just need to understand how to calculate the classification error with multiple classes. Remember that classification error is the number of mistakes only the total number in inputs. In a multiclass setting, only one class is correct and all the others will be mistakes.

Here is what a split might look like if we added a loan application class of dangerous.



The classification error is base on the total number of mistaken classes, so;

$$\text{credit split error} = ((2+1) + (6+2) + (3+1)) / 40 = 15 / 40 = 0.275$$

We can also produce prediction probabilities at each node. For instance, for an input whose credit is poor, we would end up at the rightmost node and so predict that the loan application was Dangerous, because that is the majority value prediction at that node. But we can go further than this prediction; we can assign a probability to it. This is the number in the class over the total inputs at the node. In this case;

$$P(y = \text{Dangerous} \mid w) = 7 / (3 + 1 + 7) = 0.64$$

Real Valued Features

In our model, income is an example of a real-valued feature. If we chose to use each unique value in income as a category, almost all categories would have just a single input value. Such a small set of data for a prediction would lead to overfitting. We would have 100% training accuracy with 100% probability, but this would almost certainly not generalize.

One way to handle real-valued inputs is to group them into a small number of discrete categories. For instance, we could turn income into a binary class by assigning values to a "high income" category where income >= $60,000 and to a "low income" category where income < $60,000. Now the algorithm is working with categorical data rather than continuous numerical data. This is a very natural way to make real-valued inputs work with the greedy algorithm.

Such binary categorization is similar to using a constant model in regression. The split produces a decision boundary that is a straight line that is perpendicular to an axis. Values are on one side of the line or the other, so the decision boundary is very simple.

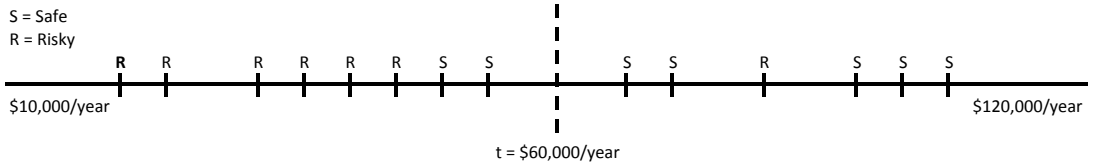
There must be some process for choosing how the continuous numeric values are grouped into categories. This is done at feature extraction time, before learning the tree. Clearly, making this choice is important to the accuracy of the model.

Another way, which allows splitting on the numerical feature any number of times, is to use threshold splits; pick one or more values as thresholds that are used when splitting values. For instance, branch on income by choosing a single threshold, such as income >= $60,000. Instead of a binary decision we could use multiple thresholds to group values into ranges. Threshold splits are determined while learning the tree, not when extracting values. The splits themselves are learned.

Furthermore, once we branch on a numeric value, we could branch on that same value again further down in the decision tree by picking new thresholds within the range of values assigned to that branch of the tree. This then allows for a more complex decision boundary for the given feature then we could get if we had assigned values to static categories at the outset.

Picking the best threshold to split on

We can visualize how the $60,000/year income threshold splits the data;



There are an infinite number of possible threshold values, t.
This graph shows t = $60,000/year.
Incomes to the left of the line are < t.
Incomes to the right of the line are > t.

In this case, if we split at $60,000, the majority class prediction would then be

- income < $60,000 predict Risky
- income >= $60,000 predict safe

When we choose this split we see that there are 2 inputs labeled safe that we would predict as risky, and one input labeled risky that we would predict as safe. Thus we have 3 mistakes of the 14 input points, so this threshold split would yield a classification error of 3/14 ~= 0.214

We can see that there are an infinite number of threshold values between any two adjacent input values. Splitting anywhere in between adjacent values yields the exact same result, so rather than consider all of these potential values, we only consider splits midway between two adjacent values. This leads to the following algorithm for picking a binary threshold for a real-valued input.

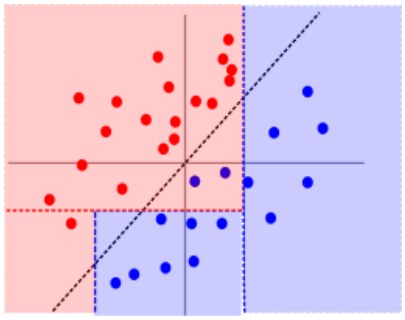
Threshold Split Selection Algorithm

- Step 1: Sort the values of the continuous-values feature $h_j(x)$, producing the set of sorted values $\{v_1, v_2, v_3, \ldots, v_N\}$
- Step 2:
  - For I = 1 to N-1
    - consider the split at $t_i = \frac{(v_1+v_2)}{2}$
    - calculate the classification error for the threshold split $h_j(x) \geq t_i$
  - Choose the t_* with the lowest classification error.

Unlike discrete categorical inputs, we can split on continuously valued inputs more than once. When splitting on a continuous feature a second time, deeper in the tree, we are limited to the range of values assigned to that branch of the tree in the prior threshold split of that feature (at prediction time, only data within that range will end up at this node in the tree). So our choice of threshold would be within the range available in the branch.

This then leads to 'cutting up' the input space using axis-perpendicular splits. Doing this over and over will lead to deep trees with lots of these threshold splits across the various continuous valued inputs.

Below is a plot, taken from this post http://stackoverflow.com/questions/4084668/questions-on-some-data-mining-algorithms, that shows data on two continuous inputs and compares an ideal regression line against how a decision tree might look. The diagonal line is the regression line. See how, if we continued to split on the two continuous values, we could create a model with the same classification error as the regression line. In effect, with enough axis aligned splits, we can approximate the diagonal line. More splits means a deeper tree.



It is very much worth viewing the "Visualizing Decision Boundaries" video. It shows that splitting on multiple continuous valued inputs can lead to deep trees with complex decision boundaries. The complex decision boundaries can lead to low or no training error, but they may not generalize well and so reflect an overfit model.

In logistic regression, large coefficients can indicate overfitting, so limiting the size of coefficients through regularization can help minimize overfitting. In the case of decision trees, deep trees can be an indication of overfitting and so limiting the depth of the tree can minimize overfitting.