Classification Trees

1 Introduction

One of the most common tasks in data mining is to build models for the prediction (or explanation) of the class of an object on the basis of some of its attributes. The term *object* should be interpreted very loosely here: it could be a customer, transaction, household, e-mail message,traffic accident, patient, handwritten digits and so on. Likewise, the class of such an object could be many things, for example:

- Good/bad credit for loan applicants.
- Respondent/non-respondent for mailing.
- Yes/no claim on insurance policy.
- Spam/no spam for e-mail message.
- The numbers 0 through 9 for handwritten digits.

We consider here the problem of learning a classification tree model from data. We start with an example concerning the classification of loan applicants into good and bad risks. After that we discuss the general theory of learning classification trees.

2 Example: credit scoring

In credit scoring, loan applicants are either rejected or accepted depending on characteristics of the applicant such as age, income and marital status. Repayment behaviour of the accepted applicants is observed by the creditor, usually leading to a classification as either a good or bad (defaulted) loan. Such historical information can be used to learn a new classification model on the basis of the characteristics of the applicant together with the observed outcome of the loan. This new classification model can then be used to make acceptance decisions for new applicants.

Figure 1 shows a classification tree that has been constructed from the data in table 1. First we explain how such a tree may be used to classify a new applicant. Later we explain in detail how such a tree can be constructed from the data.

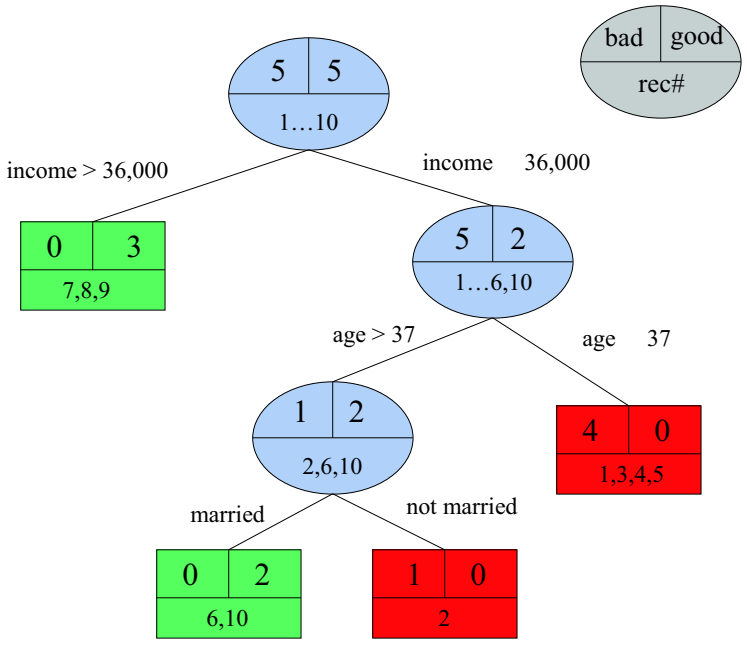

Figure 1: Tree built on credit scoring data

When a new applicant arrives he or she is "dropped down" the tree until we arrive at a leaf node, where we assign the associated class to the applicant. Suppose an applicant arrives and fills in the following information on the application form:

> age: 42, married?: no, own house?: yes, income: 30,000, gender: male.

In order to assign a class to the applicant we start at the root node of the tree and perform the associated test on income. Here we go to the right and we arrive at a new test on age. The applicant is sent to the left where we arrive at the final test on marital status. Since the applicant is not married he's sent to the right and we arrive at a leaf node with class label "bad". We predict that the applicant will default and therefore he's rejected.

How do we construct a tree from the data? The general idea is very simple: we use the historical data to find tests that are very informative about the class label. Compare the first test in the tree to the alternative test on gender shown in figure 2. The figure shows that there are five men in our data set and

Record	age	married?	own house	income	gender	class
1	22	no	no	28,000	male	bad
2	46	no	yes	32,000	female	bad
3	24	yes	yes	24,000	male	bad
4	25	no	no	27,000	male	bad
5	29	yes	yes	32,000	female	bad
6	45	yes	yes	30,000	female	good
7	63	yes	yes	58,000	male	good
8	36	yes	no	52,000	male	good
9	23	no	yes	40,000	female	good
10	50	yes	yes	28,000	female	good

Table 1: Bank credit data

three (60%) of them defaulted. There are also five women and two (40%) of them defaulted. What information concerning the class label have we gained by asking the gender? Not so much. In the data set 50% of the applicants defaulted. If we know nothing about an applicant and we had to guess the class label, we could say bad (or good) and we would be right 50% of the times. If we know the gender we can improve the prediction somewhat. If it's a male we should guess that it's a bad loan and we would be right about 60% of the times. On the other hand, if it's a female we should guess that it's a good loan and we would again be right 60% of the times. Our knowledge of the gender would improve the predictive accuracy from 50% to 60%. Now compare this to the first test in the tree in figure 1. All three applicants with income above 36,000 are good loans, and for the applicants with income below 36,000 5 out of 7 are bad loans. Using the income of an applicant, we would only make 2 mistakes out of 10, i.e. a predictive accuracy of 80%. Clearly, the test on income gives us more information about the class label than the test on gender. This is the way we build a tree. We use the historical data to compute which test provides us with the most information about the class label. When we have found this test, we split up the data in two groups that correspond to the two possible test outcomes. Within each of the resulting two groups we again look for the test that provides the most additional information on the class label. We continue in this fashion until all groups contain either only good or only bad loans.

Figure 3 gives a visual representation of the splits on income and age of the tree in figure 1. In this figure each applicant is indicated by its class label located on the (age,income) coordinate of the applicant. For example, the "good" label in the upper right corner corresponds to record number 7 in table 1. The tree algorithm divides the space into rectangular areas, and strives toward rectangles that contain applicants of a single class. The first split in the tree is indicated by the horizontal dashed line at income = 36. We see in the picture that the rectangle above the dashed line only contains good loans, whereas the lower rectangle contains good as well as bad loans. The second split in the tree is

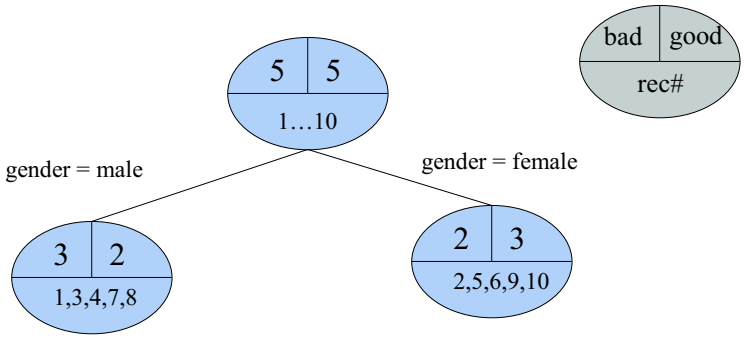

Figure 2: Test on gender

indicated by the vertical line at age=37 in the picture. This split subdivides the lower rectangle into two rectangles. The one to the left of age=37 only contains bad loans, and so we are done there. The rectangle to the right of age=37 contains two bad loans and one good loan. The final split in the tree subdivides these cases still further on the basis of marital status.

3 Building classification trees

3.1 Impurity Measures and the Quality of a split

In the previous section we saw that in constructing a tree it makes sense to choose a split (test) that provides us the most information concerning the class label. An alternative formulation is to say we should choose a split that leads to subsets that contain predominantly cases of one class. These are still vague notions, and in this section we will look at different ways of formalizing them.

The objective of tree construction is to finally obtain nodes that are pure in the sense that they contain cases of a single class only. We start with quantifying the notion of *impurity* of a node, as a function of the relative frequencies of the classes in that node:

$$i(t) = \phi(p_1, p_2, \ldots, p_J)$$

where the $p_j (j = 1, \ldots, J)$ are the relative frequencies of the J different classes in that node.

Figure 3: Visual representation of splits on age and income

Before discussing alternative ways of doing that, we list three sensible requirements of any acceptable quantification of impurity:

1. An impurity measure of a node should be at a maximum when the observations are distributed evenly over all classes in that node, i.e. at

$$(\frac{1}{J}, \frac{1}{J}, \ldots, \frac{1}{J})$$

2. An impurity measure of a node should be at a minimum when all observations belong to a single class in that node, i.e. at

$$(1, 0, \ldots, 0), (0, 1, 0, \ldots, 0), \ldots, (0, 0, \ldots, 1)$$

3. ϕ is a symmetric function of $p_1, \ldots, p_J$.

Given these requirements there are still many alternatives for defining impurity, and different implementations of tree construction algorithms use different measures. Whatever impurity we finally choose, it makes sense to define the

quality of a split (test) as the reduction of impurity that the split achieves. Hence we define the quality of split s in node t as

$$\Delta i(s,t) = i(t) - \pi(\ell)i(\ell) - \pi(r)i(r)$$

where $\pi(\ell)$ is the proportion of cases sent to the left by the split, and $\pi(r)$ the proportion of cases sent to the right. We discuss some well known impurity measures.

3.1.1 Resubstitution error

Perhaps the most obvious choice of impurity measure is the so-called resubstitution error. It measures what fraction of the cases in a node is classified incorrectly if we assign every case to the majority class in that node. That is

$$i(t) = 1 - \max_j p(j|t)$$

where $p(j|t)$ is the relative frequency of class j in node t (check that resubstitution error fullfils the requirements listed earlier). For the important case of two-class problems we denote the classes by 0 and 1; $p(0)$ denotes the relative frequency of class 0 and $p(1)$ must be equal to $1-p(0)$ since the relative frequencies must sum to 1. In figure 4 the dashed line shows the graph of resubstitution error as a function of the class distribution for two-class problems (actually it is scaled to have a maximum of 1 at $p(0) = p(1) = 0.5$; what is its true maximum?)

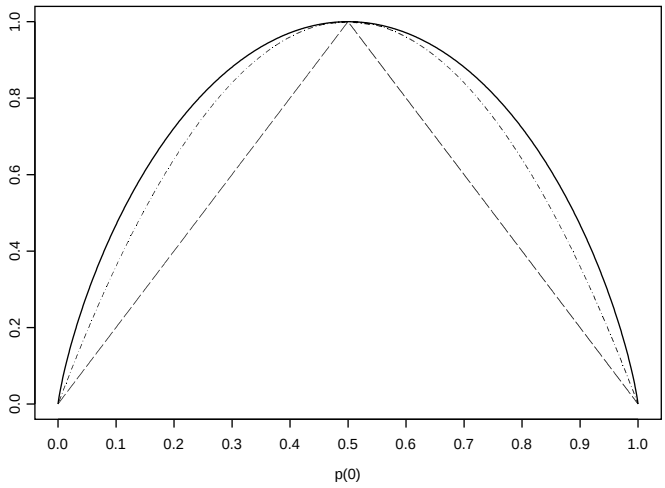

Figure 4: Graph of entropy (solid line), gini-index (dot-dash) and resubstitution error (dashed line) for two-class problem.

The quality (impurity reduction) of a split becomes (after some algebra)

$$\Delta i(s,t) = \max_j p(j|\ell)\pi(\ell) + \max_j p(j|r)\pi(r) - \max_j p(j|t)$$

Question 1 *What is the impurity reduction of the first split in figure 1 if we use resubstitution error as impurity measure?*

Despite its intuitive appeal, resubstitution error has some shortcomings as a measure of the quality of a split. Consider for example the two splits displayed in figure 5. According to resubstitution error these splits are equally good, because both yield an error of 200 out of 800. Yet we tend to prefer the split at the right because one of the resulting nodes is pure. We would like our measure of impurity to reflect this preference. What this means is that as we move from the maximum at $p(0) = 1/2$ toward the minimum at $p(0) = 1$, ϕ should decrease *faster* than linearly. Likewise, as we move from the minimum at $p(0) = 0$ toward the maximum at $p(0) = 1/2$, ϕ should increase *slower* than linearly. Equivalently, this requires that ϕ be strictly concave. If ϕ has a continuous second derivative on [0,1], then the strict concavity translates into $\phi''(p(0)) < 0, 0 < p(0) < 1$. Thus we define the class $\mathcal{F}$ of impurity functions (for two-class problems) that satisfy

1. $\phi(0) = \phi(1) = 0$,
2. $\phi(p(0)) = \phi(1 - p(0))$,
3. $\phi''(p(0)) < 0, 0 < p(0) < 1$.

Next we discuss two impurity measures that belong to this class.

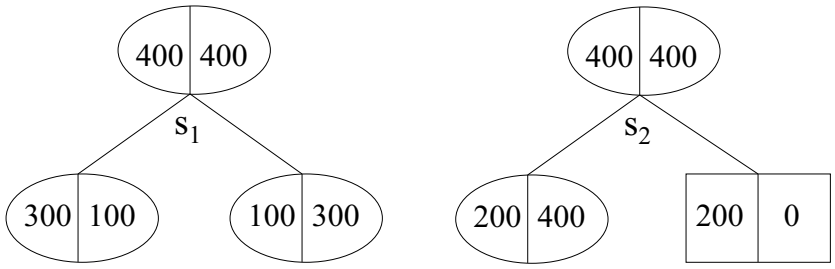

Figure 5: These splits are equally good according to resubstitution error

3.1.2 Gini-index

For the two-class case the Gini-index is

$$i(t) = p(0|t)p(1|t) = p(0|t)(1 - p(0|t))$$

The graph of the gini-index is displayed in figure 4 as the dot-dashed line, scaled to have a maximum value of 1.

Question 2 *Check that the gini-index belongs to $\mathcal{F}$.*

Question 3 *Check that if we use the gini-index as an impurity measure, split s_2 in figure 5 is preferred.*

Remark 1 *This is the variance of a Bernoulli random variable defined by drawing (with replacement) an example at random from this node, and observing its class. Hence we can view impurity reduction as variance reduction in this case. The objective is to obtain nodes with zero variance in the class label.*

The generalisation to problems with more than two classes for the gini-index is

$$i(t) = \sum_j p(j|t)(1 - p(j|t))$$

3.1.3 Entropy

Another well-known impurity measure that is used for example in the programs ID3 and C4.5 ([Qui93]) is the entropy. For the two-class case the entropy is

$$i(t) = -p(0|t)\log p(0|t) - p(1|t)\log p(1|t) = -p(0|t)\log p(0|t) - (1-p(0|t))\log(1-p(0|t))$$

The graph of entropy is displayed in figure 4 as the solid line (log with base 2).

Question 4 *Check that entropy impurity belongs to $\mathcal{F}$.*

Remark 2 *This is the average amount of information generated by drawing (with replacement) an example at random from this node, and observing its class. If a node is pure, observing the class label yields zero information.*

The generalisation to problems with more than two classes for entropy is

$$i(t) = -\sum_j p(j|t)\log p(j|t)$$

3.2 The set of splits considered

We have looked at different criteria for assessing the quality of a split. In this section we look at which splits are considered in the first place. We denote the set of explanatory variables (features) by $x_1, x_2, \ldots, x_p$. Variables may be numeric (ordered) or categorical. The set of splits considered is defined as follows:

1. Each split depends on the value of only a *single* variable.
2. If x is numeric, we consider all splits of type $x \leq c$ for all c ranging over $(-\infty, \infty)$.
3. If x is categorical, taking values in $V(x) = \{b_1, b_2, \ldots, b_L\}$, we consider all splits of type $x \in S$, where S is any non-empty proper subset of $V(x)$.

3.2.1 Splits on numeric variables

We can easily see there is only a finite number of distinct splits of the data. Let n denote the number of examples in the training sample. Then, if x is ordered, the observations in the training sample contain at most n distinct values $x_1, x_2, \ldots, x_n$ of x. This means there are at most $n-1$ distinct splits of type $x \leq c_m$, $m = 1, \ldots, n' \leq n$, where the c_m are taken halfway between consecutive distinct values of x.

Remark 3 *Note that any split between the same consecutive values of x yields the same partition of the data and therefore has the same quality computed on the training sample.*

Example 1 *Let's see how the best split on income is computed in our credit scoring example. In table 2 we have listed the impurity reduction (using the gini-index as impurity measure) achieved by all possible splits on income in the root node. For example, in the third row of the table we compute the quality of the split income $\leq$ 29 (halfway between 28 and 30). Four observations are sent to the left of which one Good and three Bad, so the impurity of the left child is (1/4)(3/4)=3/16. Six observations are sent to the right, of which two are Bad and four are Good, so the impurity of the right child is (2/6)(4/6)=8/36=2/9. The impurity of the root node is (1/2)(1/2) = 1/4, so the impurity reduction of the split is*

$$1/4 - (4/10)(3/16) - (6/10)(2/9) \approx 0.04$$

From table 2 we conclude that the best split on income is between 32 and 40, and since we always choose the midpoint, we get: income $\leq$ 36.

3.2.2 Splits on categorical variables

For a categorical variable x with L distinct values there are $2^{L-1} - 1$ possible splits to consider. There are $2^L - 2$ non-empty proper subsets of $V(x)$ (i.e. the

Income	Class	Quality (split after) 0.25−
24	B	0.1(1)(0)+0.9(4/9)(5/9) = 0.03
27	B	0.2(1)(0) + 0.8 (3/8)(5/8) = 0.06
28	B,G	0.4(3/4)(1/4) + 0.6(2/6)(4/6) = 0.04
30	G	0.5(3/5)(2/5) + 0.5(2/5)(3/5) = 0.01
32	B,B	0.7(5/7)(2/7) + 0.3(0)(1) = 0.11
40	G	0.8(5/8)(3/8) + 0.2(0)(1) = 0.06
52	G	0.9(5/9)(4/9) + 0.1(0)(1) = 0.03
58	G	

Table 2: Computation of the best split on income

empty subset and $V(x)$ itself are no splits at all). But the splits $x \in S$ and $x \in S^c$, where S^c is the complement of S with respect to $V(x)$, are clearly the same. So the number of different splits is only

$$\tfrac{1}{2}(2^L - 2) = 2^{-1}2^L - 1 = 2^{L-1} - 1$$

Example 2 *Consider the categorical variable* marital status *with possible values* {*single,married,divorced*}*. The number of distinct splits is* $2^2 - 1 = 3$*. They are: marital status* $\in$ {*single*}*, marital status* $\in$ {*married*}*, and marital status* $\in$ {*divorced*}*. The split marital status* $\in$ {*married, divorced*} *is equivalent to marital status* $\in$ {*single*}*.*

3.3 The basic tree construction algorithm

We finish this section with an overview of the basic tree construction algorithm (see Table 3). The algorithm maintains a list of nodes (a node is a set of examples) that have to be considered for expansion. Initially we put the set of training examples on this list. We select a node from the list and call it the current node. If the node contains examples from different classes (i.e. its impurity is larger than zero), then we find the best split and apply this split to the node. The resulting child nodes are added to the list. The algorithm finishes when there are no more nodes to be expanded.

3.4 Overfitting and pruning

Did you ever notice that economists and stock analysts always have a perfect explanation of an event, *once they know the event has happened*? This is an example of what is called the "silly certainty of hindsight". After some stock has gone down (or up) they can always explain that this had to happen because of a number of very compelling reasons, but could they have *predicted* the stock movement?

In data mining we also have to be careful not to fall into the trap of the silly certainty of hindsight. Once we know which applicants have defaulted

Algorithm: Construct tree

```
nodelist ← {training sample}
Repeat
    current node ← select node from nodelist
    nodelist ← nodelist − current node
    if impurity(current node) > 0
    then
        S ← candidate splits in current node
        s* ← arg max_{s∈S} impurity reduction(s,current node)
        child nodes ← apply(s*,current node)
        nodelist ← nodelist ∪ child nodes
    fi
Until nodelist = ∅
```

Table 3: Basic Tree Construction Algorithm

and which not, we can construe some complicated model that gives a perfect "explanation". Unless there are two applicants with the same attribute values, but different class label, we can build a tree that classifies every applicant used to construct the tree correctly (if necessary by creating a separate leaf node for each applicant). But is this a good model? Do we really think that we can predict the outcome for a group of new applicants with 100% accuracy? No, in fact we know we can't. Even though we can construct a model that "predicts" the data used to construct the model perfectly, its performance on new data is likely to be much worse. By fitting the model perfectly to the data, we have "overfitted" the model to the data, and have in fact been modelling noise. Think of it this way: we may have two applicants with the same characteristics except one earns 2000 euro a month and the other 2010. The one who earns 2010 euro's defaulted and the other guy didn't. Do we really think the second applicant defaulted because he earns 10 euro's more? No, of course not. More likely there is some other reason that is perhaps not recorded in our database. Perhaps the guy was a gambler, a question we didn't ask on the application form.

How do we avoid overfitting when we construct a classification tree? Two basic approaches have been tried:

- Stopping Rules: don't expand a node if the impurity reduction of the best split is below some threshold.
- Pruning: grow a very large tree and merge back nodes.

The disadvantage of a stopping rule is that sometimes you first have to make a weak split to be able to follow up with a good split.

Example 3 *Suppose we want to build a tree for the logical xor, which is defined as follows*

P	Q	P *xor* Q
1	*1*	*0*
1	*0*	*1*
0	*1*	*1*
0	*0*	*0*

Let P and Q be the attributes, and P xor Q the class label to be predicted. In the top node we have 2 examples of each class. A split on P in the top node yields no impurity reduction, and neither does a split on Q. If we make the split on either attribute however, we can follow up with a split on the other attribute in both of the child nodes to obtain a tree with no errors.

Because of this problem with stopping rules, we have to look at other ways of finding the right-sized tree. The basic idea of pruning is to initially grow a large tree, and then to prune this tree (merge back nodes) to obtain a smaller tree of the right size. What is the right size? Since we want to have a tree that has good performance on new data (i.e. data not used to construct the tree), we divide the available data into a sample used for building the tree (the training sample) and another sample for testing the performance of the tree (the test sample). We look at different trees constructed on the training sample, and select the one that performs best on the test sample. In the next section we discuss cost-complexity pruning, a well-known pruning algorithm to determine the tree of the right size.

3.4.1 Cost-complexity pruning

After building the large tree we can look at different pruned subtrees of this tree and compare their performance on a test sample. To prune a tree T in a node t means that t becomes a leaf node and all descendants of t are removed.

Figure 8 shows the tree that results from pruning the tree in figure 6 in node t_2. The branch T_{t_2} consist of node t_2 and all its descendants. The tree obtained by pruning T in t_2 is denoted by $T - T_{t_2}$.

A pruned subtree of T is any tree that can be obtained by pruning T in 0 or more nodes. If T' is a pruned subtree of T, we write this as $T' \leq T$ or alternatively $T \geq T'$.

The problem we face now is that the number of pruned subtrees may become very large and it may not be feasible to compare them all on a test set.

Remark 4 *More precisely, let $|\tilde{T}|$ denote the number of leaf nodes of binary tree T. Then the number of pruned subtrees of T is $\lfloor 1.5028369^{|\tilde{T}|} \rfloor$. So, for a tree with 25 leaf nodes (which is not unusually large) we would already have to compare 26,472 pruned subtrees.*

The basic idea of cost-complexity pruning is not to consider all pruned subtrees, but only those that are the "best of their kind" in a sense to be defined below.

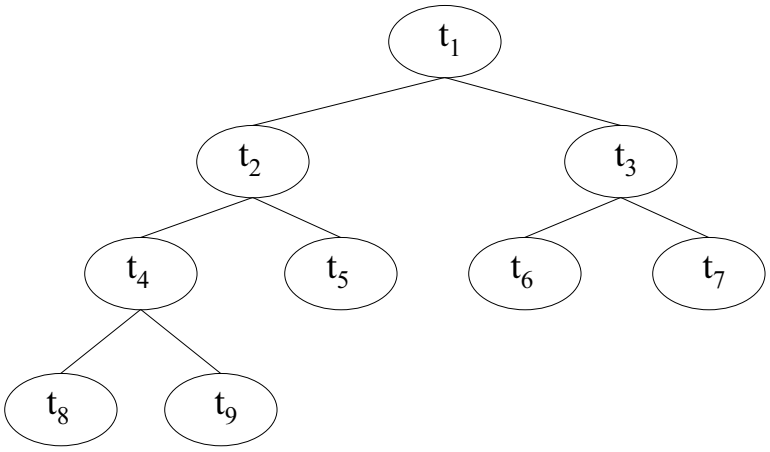

Figure 6: Tree T with leaf nodes $\tilde{T} = \{t_5, t_6, t_7, t_8, t_9\}, |\tilde{T}| = 5$

Let $R(T)$ denote the fraction of cases in the training sample that are misclassified by T ($R(T)$ is called the resubstitution error of T).

We define the total cost $C_\alpha(T)$ of tree T as

$$C_\alpha(T) = R(T) + \alpha|\tilde{T}|$$

The total cost of tree T consists of two components: resubstitution error $R(T)$, and a penalty for the complexity of the tree $\alpha|\tilde{T}|$. In this expression $\tilde{T}$ denotes the set of leaf nodes of T, and α is the parameter that determines the complexity penalty: when the number of leaf nodes increases with one (one additional split in a binary tree), then the total cost (if R remains equal) increases with α. Depending on the value of $\alpha(\geq 0)$ a complex tree that makes no errors may now have a higher total cost than a small tree that makes a number of errors.

We denote the large tree that is to be pruned to the right size by T_{max}. If we fix the value of α, there is a smallest minimizing subtree $T(\alpha)$ of T_{max} that fulfills the following conditions:

1. $C_\alpha(T(\alpha)) = \min_{T \leq T_{max}} C_\alpha(T)$
2. If $C_\alpha(T) = C_\alpha(T(\alpha))$ then $T(\alpha) \leq T$.

The first condition says there is no subtree of T_{max} with lower cost than $T(\alpha)$, at this value of α. The second condition says that if there is a tie, i.e. there is more than one tree that achieves this minimum, then we pick the smallest tree (i.e. the one that is a subtree of all others that achieve the minimum).

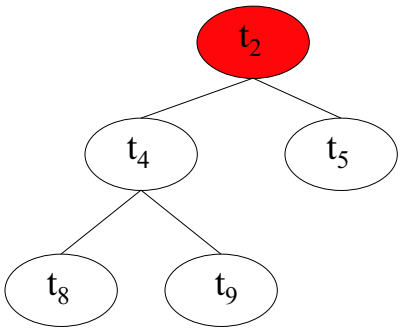

Figure 7: Branch T_{t_2}: $\tilde{T}_{t_2} = \{t_5, t_8, t_9\}, |\tilde{T}_{t_2}| = 3$

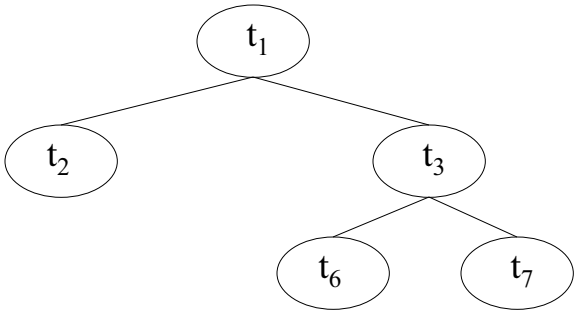

Figure 8: Tree $T - T_{t_2}$ resulting from pruning T in t_2

It can be shown that for every value of α there is such a smallest minimizing subtree. This is not trivial. What this says is that it cannot occur that we have two trees that achieve the minimum, but are incomparable, i.e. neither is a subtree of the other. We will not prove this result.

Although α goes through a continuum of values, there is only a finite number of subtrees of T_{max}. We can construct a decreasing sequence of subtrees of T_{max}

$$T_1 > T_2 > T_3 > \ldots > \{t_1\}$$

(where t_1 is the root node of the tree) such that T_k is the smallest minimizing subtree for $\alpha \in [\alpha_k, \alpha_{k+1})$. This is an important result, because it means we can obtain the next tree in the sequence by pruning the current one. This allows the specification of an efficient algorithm to find the smallest minimizing subtrees at different values of α.

The first tree in the sequence, T_1 is the smallest subtree of T_{max} with the

same resubstitution error as T_{max} (i.e. $T_1 = T(\alpha = 0)$).

Remark 5 *If we continue splitting until a node contains observations of a single class, as outlined in the basic tree construction algorithm, then $T_1 = T_{max}$. Usually we apply a different stopping criterion however, e.g. we also stop splitting when the number of observations in a node is below a certain threshold. Furthermore, we may have a node with observations from different classes, but with all attribute vectors identical. In that case there is no possibility of splitting the node any further. With these more complex stopping rules, there may be subtrees of T_{max} that have the same resubstitution error.*

The algorithm to compute T_1 from T_{max} is straightforward. Find any pair of leaf nodes (with a common parent) that can be merged back (i.e. pruned in the parent node) without increasing the resubtitution error . Continue until no more such pair can be found. Thus we obtain a tree that has the same total cost as T_{max} at $\alpha = 0$, but since it is smaller it is preferred over T_{max}.

Algorithm: Compute T_1 from T_{max}
$T' \leftarrow T_{max}$
Repeat
  Pick any pair of terminal nodes ℓ and r with common parent t in T'
  such that $R(t) = R(\ell) + R(r)$, and set
  $T' \leftarrow T' - T_t$ (i.e. prune T' in t)
Until no more such pair exists
$T_1 \leftarrow T'$

How do we find the trees in the sequence and the corresponding values of α? Let T_t denote the branch of T with root node t. At which value of α does $T - T_t$ become better than T? If we were to prune in t, its contribution to the total cost of $T - T_t$ would become $C_\alpha(\{t\}) = R(t) + \alpha$, where $R(t) = r(t)p(t)$, $r(t)$ is the resubstitution error at node t, and $p(t)$ is the proportion of cases that fall into node t.

The contribution of T_t to the total cost of T is $C_\alpha(T_t) = R(T_t) + \alpha|\tilde{T}_t|$, where $R(T_t) = \sum_{t' \in \tilde{T}_t} R(t')$. $T - T_t$ becomes the better tree when $C_\alpha(\{t\}) = C_\alpha(T_t)$, because at that value of α they have the same total cost, but $T - T_t$ is the smallest of the two. When $C_\alpha(\{t\}) = C_\alpha(T_t)$, we have

$$R(T_t) + \alpha|\tilde{T}_t| = R(t) + \alpha$$

Solving for α we get

$$\alpha = \frac{R(t) - R(T_t)}{(|\tilde{T}_t| - 1)}$$

So for any node t in T_1, if we increase α, then when

$$\alpha = \frac{R(t) - R(T_{1,t})}{(|\tilde{T}_{1,t}| - 1)}$$

the tree obtained by pruning in t becomes better than T_1. The basic idea is to compute this value of α for each node in T_1, and then to select the "weakest links" (there may be more than one), i.e. the nodes for which

$$g(t) = \frac{R(t) - R(T_{1,t})}{(|\tilde{T}_{1,t}| - 1)}$$

is the smallest. We prune T_1 in these nodes to obtain T_2, the next tree in the sequence. Then we repeat the same process for this pruned tree, and so on until we reach the root node.

Here we give an outline of the algorithm for cost-complexity pruning. We start at T_1, the smallest minimizing subtree of T_{max} for $\alpha = 0$. Then we repeat the following until we reach the root node: For each node t in the current tree T_k, we compute $g_k(t)$, the value of α at which $T_k - T_t$ becomes better than T_k. Then we prune T_k in all nodes for which g_k achieves the minimum to obtain T_{k+1}.

Algorithm: Compute tree sequence

$T_1 \leftarrow T(0)$
$\alpha_1 \leftarrow 0$
$k \leftarrow 1$
While $T_k > \{t_1\}$ do
  For all non-terminal nodes $t \in T_k$
    $g_k(t) \leftarrow \frac{R(t) - R(T_{k,t})}{(|\tilde{T}_{k,t}| - 1)}$
  $\alpha_{k+1} \leftarrow \min_t g_k(t)$
  Visit the nodes in top-down order and prune
  whenever $g_k(t) = \alpha_{k+1}$ to obtain T_{k+1}
  $k \leftarrow k + 1$
od

In pruning the tree, we visit the nodes in top-down order to avoid considering nodes which themselves will be pruned away.

Example 4 *As an example, let's compute the sequence of subtrees and corresponding values of α for the tree depicted in figure 9.*

It is easy to see that $T_1 = T_{max}$, since merging two leaf nodes (by pruning in t_3 or t_5) leads to an increase in resubstitution error. Next we compute $g_1(t)$ for all nodes t in T_1; $g_1(t)$ is the value of α at which the total cost of the tree obtained by pruning in t becomes equal to the total cost of T_1. We compute: $g_1(t_1) = 1/8, g_1(t_2) = 3/20, g_1(t_3) = 1/20, g_1(t_5) = 1/20$. We show how $g_1(t_5)$ was computed in detail: $R(t_5) = 10/200 = 1/20$ since in t_5 10 cases are classified incorrectly and we have 200 cases in total. An alternative way to compute $R(t_5)$ is through $R(t) = r(t) \times p(t)$. Then we get $R(t_5) = 1/7 \times 70/200 = 70/1400 = 1/20$. $R(T_{1,t_5}) = 0$ since both leaf nodes below t_5 in T_1 have 0 error. So we compute

$$g_1(t_5) = \frac{R(t_5) - R(T_{1,t_5})}{|\tilde{T}_{1,t_5}| - 1} = \frac{1/20 - 0}{2 - 1} = \frac{1}{20}$$

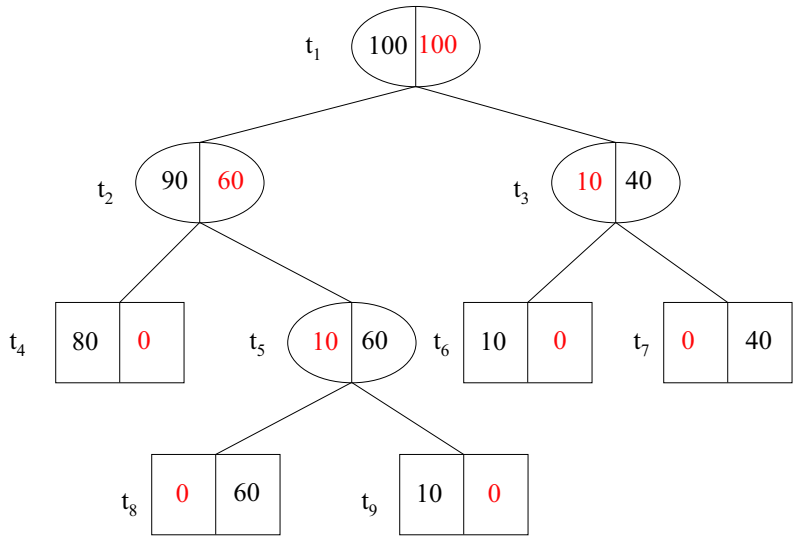

Figure 9: Tree T_{max}

Nodes t_3 and t_5 both achieve the minimal value of g_1, so we obtain T_2 from T_1 by pruning in both nodes. We then obtain the tree in figure 10.

Next we compute the g-values for T_2: $g_2(t_1) = 2/10, g_2(t_2) = 1/4$. The minimum is achieved by t_1, so we prune in t_1 and we have reached the root of the tree ($T_3 = \{t_1\}$).

The sequence of α-values has become: $\alpha_1 = 0$, $\alpha_2 = 1/20$, $\alpha_3 = 2/10$. Thus T_1 is the best tree for $\alpha \in [0, \frac{1}{20})$, T_2 is the best tree for $\alpha \in [\frac{1}{20}, \frac{2}{10})$ and $T_3 = \{t_1\}$ for $\alpha \in [\frac{2}{10}, \infty)$.

3.5 Selection of the final tree

3.5.1 Train and Test

The most straightforward way to select the final tree from the sequence created with cost complexity pruning is to pick the one with the lowest error rate on a test set. We denote the estimated error rate of tree T on a test sample by $R^{ts}(T)$.

Remark 6 *The standard error of R^{ts} as an estimate of the true error rate R^* is*

$$SE(R^{ts}) = \sqrt{\frac{R^{ts}(1 - R^{ts})}{n_{test}}},$$

where n_{test} is the number of examples in the test set.

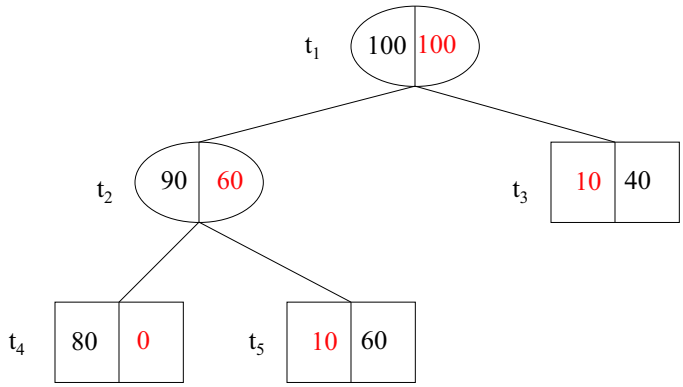

Figure 10: Tree T_2

Experiments have shown that usually there is quite a long subsequence of trees with error rates close to each other, as illustrated in figure 11.

Again, experiments have shown that the tree size that achieves the minimum within this long flat valley, is quite sensitive to the examples we happen to select for testing. To reduce this instability CART/Rpart [BFOS84] use the 1-SE rule: select the smallest tree with R^{ts} within one standard error of the minimum. In figure 11 we have depicted the tree that achieves the lowest R^{ts} with the interval $[R^{ts}, R^{ts} + SE]$ as a vertical line next to it. The two trees to the left of it have a value of R^{ts} within this interval, and the leftmost (the smallest one) is selected as the final tree.

3.5.2 Cross-validation

When the data set is relatively small, it is a bit of a waste to set aside part of the data for testing. A way to avoid this problem is to use *cross-validation*. In that case we can proceed as follows.

Construct a tree on the full data set, and compute $\alpha_1, \alpha_2, \ldots, \alpha_K$ and $T_1 > T_2 > \ldots > T_K$. Recall that T_k is the smallest minimizing subtree for $\alpha \in [\alpha_k, \alpha_{k+1})$.

Now we want to select a tree from this sequence, but we have already used all data for constructing it, so we have no test set to select the final tree.

The trick is that we are going to estimate the error of a tree T_k from this sequence in an indirect way as follows.

Step 1

Set $\beta_1 = 0, \beta_2 = \sqrt{\alpha_2 \alpha_3}, \beta_3 = \sqrt{\alpha_3 \alpha_4}, \ldots, \beta_{K-1} = \sqrt{\alpha_{K-1} \alpha_K}, \beta_K = \infty$. β_k is considered to be a typical value for $[\alpha_k, \alpha_{k+1})$, and therefore as the value corresponding to T_k.

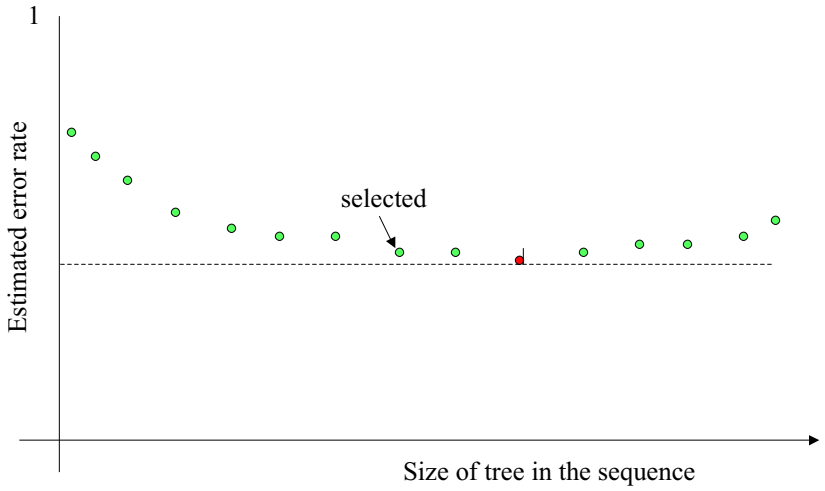

Figure 11: Estimated error rate as a function of tree size

Step 2

Divide the data set in v groups $G_1, G_2, \ldots, G_v$ (of equal size) and for each group G_j

1. Compute a tree sequence with cost-complexity pruning on all data *except* G_j, and determine $T^{(j)}(\beta_1), T^{(j)}(\beta_2), \ldots, T^{(j)}(\beta_K)$ for this sequence.

2. Compute the error of $T^{(j)}(\beta_k)$ on G_j.

Remark 7 *Note that $T^{(j)}(\beta_k)$ is the smallest minimizing subtree from the sequence built on all data except G_j, for $\alpha = \beta_k$.*

Step 3

For each β_k, sum the errors of $T^{(j)}(\beta_k)$ over G_j $(j = 1, \ldots, v)$. Let β_h be the one with the lowest overall error. Since β_h corresponds to T_h, we select T_h *from the tree sequence constructed on all data* as the final tree. Use the error rate computed with cross-validation as an estimate of its error rate.

Alternatively, we could again use the 1-SE rule in the last step to select the final tree from the sequence.

It is important to note that in the procedure described here we are effectively using cross-validation to select the best value of the complexity parameter from the set $\beta_1, \ldots, \beta_K$. Once the best value has been determined, the corresponding tree from the original cost-complexity sequence is returned.

4 Handling missing data in trees

Most data mining algorithms and statistical methods assume there are no missing values in the data set. In practical data analysis this assumption is almost never true. Some common reasons for missing data are:

1. respondents may not be willing to answer some question
2. errors in data entry
3. joining of not entirely matching data sets

The most common "solution" to handling missing data is to throw out all observations that have one or more attributes missing. This practice has some disadvantages however:

1. Potential bias: if examples with missing attributes differ in some way from completely observed examples, then our analysis may yield biased results.
2. Loss of power: we may have to throw away a lot of examples, and the precision of our results is reduced accordingly.

For tree-based models some ad-hoc procedures have been constructed to handle missing data both for training and prediction.

We have to solve the following problems if we want to construct and use a tree with incomplete data:

1. How do we determine the quality of a split?
2. Which way do we send an observation with a missing value for the best split? (both in training and prediction)

Note that an observation with a missing *class label* is useless for tree construction, and will be thrown away.

In determining the quality of a split, CART [BFOS84] simply ignores missing values, i.e.

$$\Delta i(s,t) = i(t) - \pi(\ell) i(\ell) - \pi(r) i(r)$$

is computed using *observed* values only. This "solves" the first problem, but now we still have to determine which way to send an observation with a missing value for the best split. To this end, CART computes so-called *surrogate splits*. A surrogate split is a split that is similar to the best split, in the sense that it makes a similar partition of the cases in the current node. To determine the value of an alternative split as a surrogate we make a cross-table (see Table 4).

In this table $\pi(\ell^*, \ell')$ denotes the proportion of cases that is sent to the left by both the best split s^* and alternative split s', and likewise for $\pi(r^*, r')$, so $\pi(\ell^*, \ell') + \pi(r^*, r')$ is the proportion of cases that is sent the same way by both splits. It is a measure of the similarity of the splits, or alternatively: it indicates how well we can predict which way a case is sent by the best split by looking at the alternative split.

$\pi(\ell^*, \ell')$	$\pi(\ell^*, r')$
$\pi(r^*, \ell')$	$\pi(r^*, r')$

Table 4: Cross-table for computing the value of alternative split s' as a surrogate for best split s^*.

Remark 8 *If $\pi(\ell^*, \ell') + \pi(r^*, r') < 0.5$ we can get a better surrogate by switching left and right for the alternative split. Furthermore it should be noted that the proportions in table 4 are computed on the cases where both the variable of the best and alternative split are observed.*

The alternative splits with $\pi(\ell^*, \ell') + \pi(r^*, r') > \max(\pi(\ell^*), \pi(r^*))$ are sorted in descending order of similarity. Now if the value of the best split is missing, try the first surrogate on the list, and if that one is missing as well try the second one, etc. If all surrogates are missing, use $\max(\pi(\ell^*), \pi(r^*))$.

Example 5 *In figure 12, we have depicted the best split at the left, and an alternative split on marital status on the right. The best split is the right child of the root of the credit scoring tree as shown in figure 1. What is the value of the alternative split on marital status as a surrogate? We can read from figure 12 that both splits send records 6 and 10 to the left, so $\pi(\ell^*, \ell') = \frac{2}{7}$. Both splits send records 1 and 4 to the right, so $\pi(r^*, r') = \frac{2}{7}$ as well. Its value as a surrogate is therefore $\pi(\ell^*, \ell') + \pi(r^*, r') = \frac{2}{7} + \frac{2}{7} = \frac{4}{7}$. Since $\max(\pi(\ell^*), \pi(r^*)) = \frac{4}{7}$ as well, the alternative split on marital status is not a good surrogate.*

Question 5 *In figure 13, we have depicted the best split at the left, and an alternative split on gender on the right. Is the split on gender a good surrogate?*

5 Computational efficiency

In this section we discuss some issues concerning the computational efficiency of tree construction algorithms.

5.1 Splitting on categorical attributes

We have seen that for a categorical attribute with L distinct values, the number of distinct splits is $2^{L-1} - 1$. For attributes with many distinct values, the number of possible splits may become quite large. For example, with 15 distinct values the number of splits is already $2^{14} - 1 = 16383$. For binary class problems and impurity measures that belong to class $\mathcal{F}$ there is a more efficient algorithm for finding the optimal split. Let $p(0|x = b_\ell)$ denote the relative frequency of class 0 for observations in the current node with $x = b_\ell$. Order the $p(0|x = b_\ell)$, that is,

$$p(0|x = b_{\ell_1}) \leq p(0|x = b_{\ell_2}) \leq \ldots \leq p(0|x = b_{\ell_L})$$

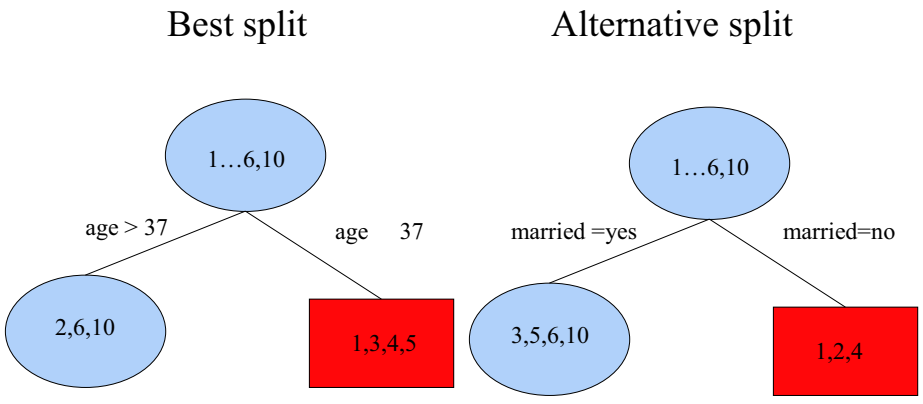

Figure 12: Alternative split on marital status

Then one of the L subsets

$$\{b_{\ell_1}, \ldots, b_{\ell_h}\}, \quad h = 1, \ldots, L-1,$$

is the optimal split. Thus the search is reduced from looking at $2^{L-1} - 1$ splits to $L-1$ splits. Intuitively, the best split should put all those categories leading to high probabilities of being in class 0 into one node, and the categories leading to lower class 0 probabilities in the other.

5.2 Splitting on numerical attributes

We have seen that if x is a numeric variable with n distinct values, we have to compute $n-1$ splits to determine the optimal one. Let $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ denote the sorted distinct values of x. Let $x^{(i)}$ and $x^{(i+1)}$ be any two consecutive values of x. Let E denote the set of examples with $x = x^{(i)}$ or $x = x^{(i+1)}$. Fayyad and Irani [FI92] have shown that if all examples in E have the same class label, then the optimal entropy split cannot occur between $x^{(i)}$ and $x^{(i+1)}$.

This means we don't have to compute the split value for these cases. This can save a lot of computation.

Question 6 *For simplicity, suppose that all values of x are distinct and that we have two classes. In the best case, what reduction in computation is achieved by exploiting this rule? And in the worst case?*

The proof is rather lengthy, so we ommit it here. See [FI92] if you want to know the details.

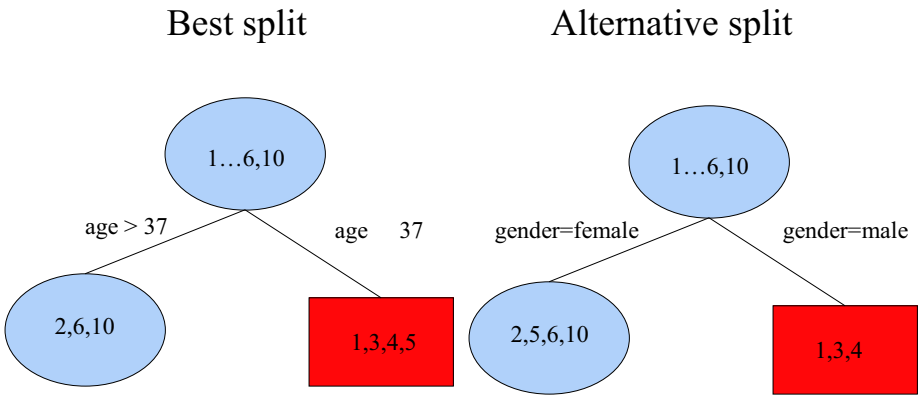

Figure 13: Alternative split on gender

5.3 Computational complexity of tree construction

We can make some rough statements about the computational complexity of constructing a classification tree. Let's assume all p attributes $x_1, \ldots, x_p$ are numeric. Initially we have to sort all attributes, as a preparation to computing the values of the splits. This takes $O(n \log n)$ time for each attribute.

During tree construction we have to evaluate all possible splits in each node. Let's assume we continue splitting until each leaf contains only one example. The time this takes depends on the way the tree is balanced. The best case occurs when in each node the best split divides the examples exactly in half, half of the examples go to the left and half of them go to the right. Then the resulting tree has depth $\log n$. At each level in the tree we have to consider $O(n)$ splits for each attribute, so per attribute this takes again $O(n \log n)$ time.

In the worst case we split of one example at a time and we get a tree of depth n. At each level in the tree we again have to consider $O(n)$ splits for each attribute, so per attribute this takes $O(n^2)$ time.

Consider the case where we have $n = 16$ observations. Let's see how many splits we have to compute depending on the balance of the tree. We start with the perfectly balanced tree depicted in figure 14, where the number of cases is given inside the node.

On the first level, there are $16 - 1 = 15$ splits to consider. On the second level $2 \times (7 - 1) = 14$, etc. In total we get $15 + 14 + 12 + 8 = 49$ possible splits. If n is a power of two, the general formula is

$$\sum_{\ell=1}^{\log(n)} n - 2^{\ell-1} = n \log n - \sum_{\ell=1}^{\log(n)} 2^{\ell-1} = n \log n - n + 1$$

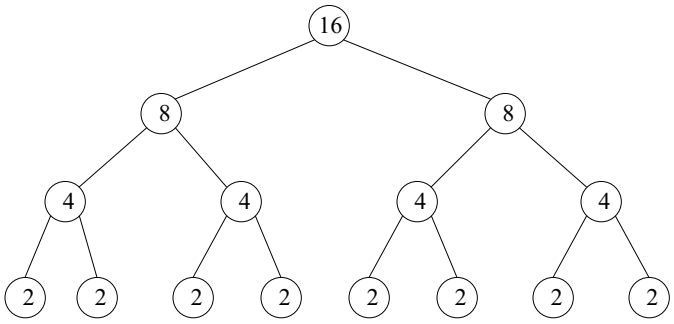

Figure 14: Balanced tree for $n = 16$

since $\sum_{k=1}^{m} 2^{k-1} = 2^m - 1$.

At the other extreme is the unbalanced tree of figure 15. The number of splits to consider for this tree is

$$15 + 14 + \ldots + 1 = \frac{15 \times 16}{2} = 120$$

and in general

$$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = \frac{n^2 - n}{2}$$

since $\sum_{i=1}^{m} i = \frac{1}{2}m(m+1)$.

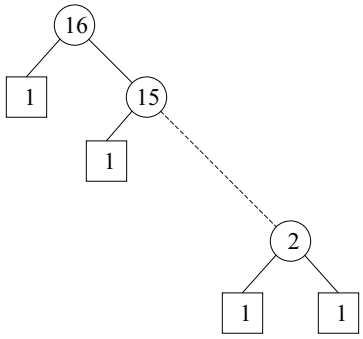

Figure 15: Unbalanced tree for $n = 16$

It's hard to make statements about the average case but intuitively it should be closer to the best case than to the worst case.

References

[BFOS84] L. Breiman, J.H. Friedman, R.A. Olshen, and C.T. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, California, 1984.

[FI92] U. Fayyad and K. Irani. On the handling of continuous-valued attributes in decision tree generation. *Machine Learning*, 8:87–102, 1992.

[Qui93] J.R. Quinlan. *C4.5 Programs for Machine Learning*. Morgan Kaufmann, San Mateo, California, 1993.