9
Avoiding Overfitting of Decision Trees

The Top-Down Induction of Decision Trees (TDIDT) algorithm described in previous chapters is one of the most commonly used methods of classification. It is well known, widely cited in the research literature and an important component of many successful commercial packages. However, like many other methods, it suffers from the problem of *overfitting* to the training data, resulting in some cases in excessively large rule sets and/or rules with very low predictive power for previously unseen data.

A classification algorithm is said to *overfit* to the training data if it generates a decision tree (or any other representation of the data) that depends too much on irrelevant features of the training instances, with the result that it performs well on the training data but relatively poorly on unseen instances.

Realistically, overfitting will always occur to a greater or lesser extent simply because the training set does not contain all possible instances. It only becomes a problem when the classification accuracy on unseen instances is significantly downgraded. We always need to be aware of the possibility of significant overfitting and to seek ways of reducing it.

In this chapter we look at ways of adjusting a decision tree either while it is being generated, or afterwards, in order to increase its predictive accuracy. The idea is that generating a tree with fewer branches than would otherwise be the case (known as *pre-pruning*) or removing parts of a tree that has already been generated (known as *post-pruning*) will give a smaller and simpler tree. This tree is unlikely to be able to predict correctly the classification of some of the instances in the training set. As we already know what those values should be this is of little or no importance. On the other hand the simpler tree may be

M. Bramer, *Principles of Data Mining*, Undergraduate Topics
in Computer Science, DOI 10.1007/978-1-4471-7307-6_9

able to predict the correct classification more accurately for unseen data — a case of 'less means more'.

We will start by looking at a topic that at first sight is unrelated to the subject of this chapter, but will turn out to be important: how to deal with inconsistencies in a training set.

9.1 Dealing with Clashes in a Training Set

If two (or more) instances in a training set have the same combination of attribute values but different classifications the training set is inconsistent and we say that a *clash* occurs.

There are two main ways this can happen.

1. One of the instances has at least one of its attribute values or its classification incorrectly recorded, i.e. there is noise in the data.
2. The clashing instances are both (or all) correct, but it is not possible to discriminate between them on the basis of the attributes recorded.

In the second case the only way of discriminating between the instances is by examining the values of further attributes, not recorded in the training set, which in most cases is impossible. Unfortunately there is usually no way except 'intuition' of distinguishing between cases (1) and (2).

Clashes in the training set are likely to prove a problem for any method of classification but they cause a particular problem for tree generation using the TDIDT algorithm because of the 'adequacy condition' introduced in Chapter 4. For the algorithm to be able to generate a classification tree from a given training set, it is only necessary for one condition to be satisfied: no two or more instances may have the same set of attribute values but different classifications. This raises the question of what to do when the adequacy condition is not satisfied.

It is generally desirable to be able to generate a decision tree even when there are clashes in the training data, and the basic TDIDT algorithm can be adapted to do this.

9.1.1 Adapting TDIDT to Deal with Clashes

Consider how the TDIDT algorithm will perform when there is a clash in the training set. The method will still produce a decision tree but (at least) one of the branches will grow to its greatest possible length (i.e. one term for each of

the possible attributes), with the instances at the lowest node having more than one classification. The algorithm would like to choose another attribute to split on at that node but there are no 'unused' attributes and it is not permitted to choose the same attribute twice in the same branch. When this happens we will call the set of instances represented by the lowest node of the branch the *clash set*.

A typical clash set might have one instance with classification *true* and one with classification *false*. In a more extreme case there may be several possible classifications and several instances with each classification in the clash set, e.g. for an object recognition example there might be three instances classified as *house*, two as *tree* and two as *lorry*.

Figure 9.1 shows an example of a decision tree generated from a training set with three attributes x, y and z, each with possible values 1 and 2, and three classifications $c1$, $c2$ and $c3$. The node in the bottom row labelled 'mixed' represents a clash set, i.e. there are instances with more than one of the three possible classifications, but no more attributes to split on.

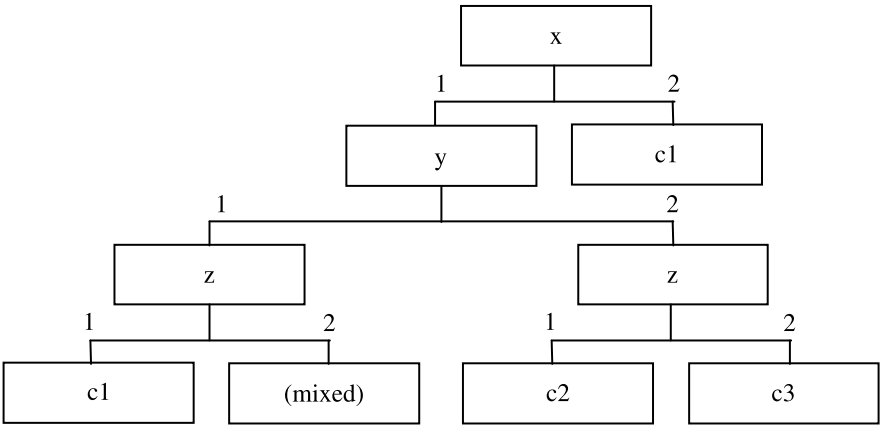

Figure 9.1 Incomplete Decision Tree (With Clash Set)

There are many possible ways of dealing with clashes but the two principal ones are:

(a) The 'delete branch' strategy: discard the branch to the node from the node above. This is similar to removing the instances in the clash set from the training set (but not necessarily equivalent to it, as the order in which the attributes were selected might then have been different).

Applying this strategy to Figure 9.1 gives Figure 9.2. Note that this tree will be unable to classify unseen instances for which $x = 1$, $y = 1$ and $z = 2$, as previously discussed in Section 6.7.

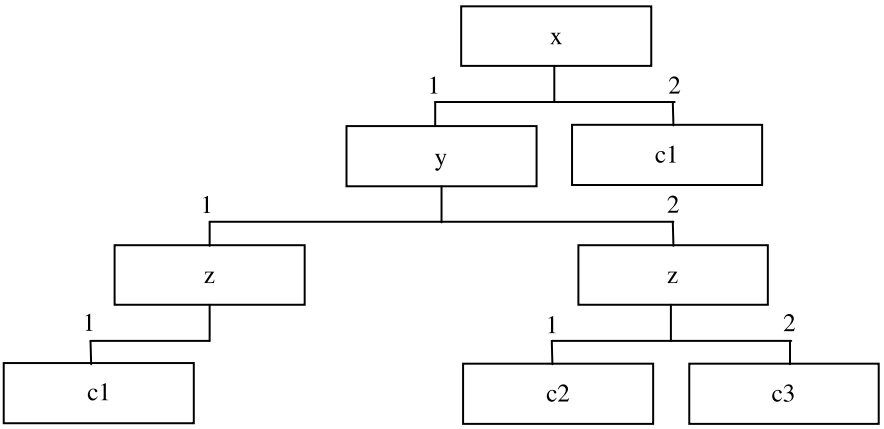

Figure 9.2 Decision Tree Generated from Figure 9.1 by 'Delete Branch' Strategy

(b) The 'majority voting' strategy: label the node with the most common classification of the instances in the clash set. This is similar to changing the classification of some of the instances in the training set (but again not necessarily equivalent, as the order in which the attributes were selected might then have been different).

Applying this strategy to Figure 9.1 gives Figure 9.3, assuming that the most common classification of the instances in the clash set is $c3$.

The decision on which of these strategies to use varies from one situation to another. If there were, say, 99 instances classified as *yes* and one instance classified as *no* in the training set, we would probably assume that the *no* was a misclassification and use method (b). If the distribution in a weather forecasting application were 4 *rain*, 5 *snow* and 3 *fog*, we might prefer to discard the instances in the clash set altogether and accept that we are unable to make a prediction for that combination of attribute values.

A middle approach between the 'delete branch' and the 'majority voting' strategies is to use a *clash threshold*. The clash threshold is a percentage from 0 to 100 inclusive.

The 'clash threshold' strategy is to assign all the instances in a clash set to the most commonly occurring class for those instances provided that the proportion of instances in the clash set with that classification is at least equal to the clash threshold. If it is not, the instances in the clash set (and the corresponding branch) are discarded altogether.

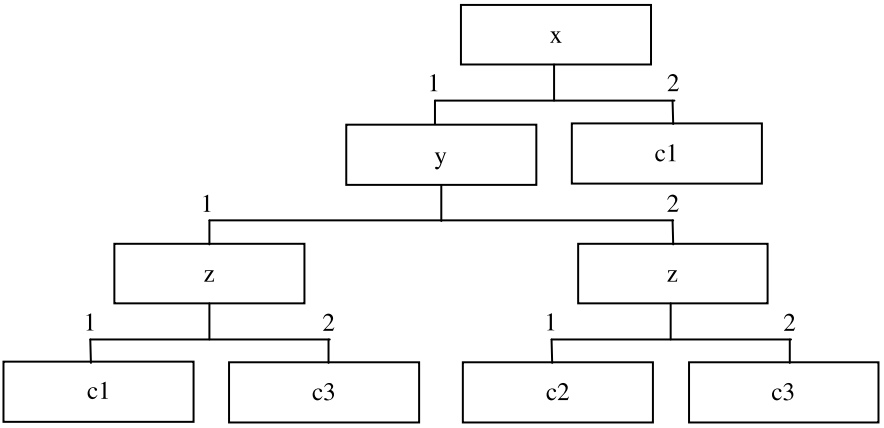

Figure 9.3 Decision Tree Generated from Figure 9.1 by 'Majority Voting' Strategy

Setting the clash threshold to zero gives the effect of *always* assigning to the most common class, i.e. the 'majority voting' strategy. Setting the threshold to 100 gives the effect of *never* assigning to the most common class, i.e. the 'delete branch' strategy.

Clash threshold values between 0 and 100 give a middle position between these extremes. Reasonable percentage values to use might be 60, 70, 80 or 90.

Figure 9.4 shows the result of using different clash thresholds for the same dataset. The dataset used is the *crx* 'credit checking' dataset modified by deleting all the continuous attributes to ensure that clashes will occur. The modified training set does not satisfy the adequacy condition.

The results were all generated using TDIDT with attributes selected using information gain in 'train and test' mode.

Clash threshold	Training set			Test set		
	Correct	Incorr.	Unclas	Correct	Incorr.	Unclas
0% Maj. Voting	651	39	0	184	16	0
60%	638	26	26	182	10	8
70%	613	13	64	177	3	20
80%	607	11	72	176	2	22
90%	552	0	138	162	0	38
100% Del. Branch	552	0	138	162	0	38

Figure 9.4 Results for *crx* (Modified) With Varying Clash Thresholds

From the results given it is clear that when there are clashes in the training data it is no longer possible to obtain a decision tree that gives 100% predictive accuracy on the training set from which it was generated.

The 'delete branch' option (threshold = 100%) avoids making any errors but leaves many of the instances unclassified. The 'majority voting' strategy (threshold = 0%) avoids leaving instances unclassified but gives many classification errors. The results for threshold values 60%, 70%, 80% and 90% lie between these two extremes. However, the predictive accuracy for the training data is of no importance — we already know the classifications! It is the accuracy for the test data that matters.

In this case the results for the test data are very much in line with those for the training data: reducing the threshold value increases the number of correctly classified instances but it also increases the number of incorrectly classified instances and the number of unclassified instances falls accordingly.

If we use the 'default classification strategy' and automatically allocate each unclassified instance to the largest class in the original training set, the picture changes considerably.

Clash threshold	Training set			Test set	
	Correct	Incorr.	Unclas	Correct	Incorr.
0% maj. voting	651	39	0	184	16
60%	638	26	26	188	12
70%	613	13	64	189	11
80%	607	11	72	189	11
90%	552	0	138	180	20
100% del. branch	552	0	138	180	20

Figure 9.5 Results for *crx* (Modified) With Varying Clash Thresholds (Using Default to Largest Class)

Figure 9.5 shows the results given in Figure 9.4 modified so that for the test data any unclassified instances are automatically assigned to the largest class. The highest predictive accuracy is given for clash thresholds 70% and 80% in this case.

Having established the basic method of dealing with clashes in a training set, we now turn back to the main subject of this chapter: the problem of avoiding the overfitting of decision trees to data.

9.2 More About Overfitting Rules to Data

Let us consider a typical rule such as

IF $a = 1$ and $b =$ yes and $z =$ red THEN class = OK

Adding an additional term to this rule will *specialise* it, for example the augmented rule

IF $a = 1$ and $b =$ yes and $z =$ red and $k =$ green THEN class = OK

will normally refer to fewer instances than the original form of the rule (possibly the same number, but certainly no more).

In contrast, removing a term from the original rule will *generalise* it, for example the depleted rule

IF $a = 1$ and $b =$ yes THEN class = OK

will normally refer to more instances than the original form of the rule (possibly the same number, but certainly no fewer).

The principal problem with TDIDT and other algorithms for generating classification rules is that of *overfitting*. Every time the algorithm splits on an attribute an additional term is added to each resulting rule, i.e. tree generation is a repeated process of specialisation.

If a decision tree is generated from data containing noise or irrelevant attributes it is likely to capture erroneous classification information, which will tend to make it perform badly when classifying unseen instances.

Even when that is not the case, beyond a certain point, specialising a rule by adding further terms can become counter-productive. The generated rules give a perfect fit for the instances from which they were generated but in some cases are too specific (i.e. specialised) to have a high level of predictive accuracy for other instances. To put this point another way, if the tree is over-specialised, its ability to generalise, which is vital when classifying unseen instances, will be reduced.

Another consequence of excessive specificity is that there is often an unnecessarily large number of rules. A smaller number of more general rules may have greater predictive accuracy on unseen data.

The standard approach to reducing overfitting is to sacrifice classification accuracy on the training set for accuracy in classifying (unseen) test data. This can be achieved by pruning the decision tree. There are two ways to do this:

- **Pre-pruning** (or *forward pruning*)
  Prevent the generation of non-significant branches
- **Post-pruning** (or *backward pruning*)
  Generate the decision tree and then remove non-significant branches.

Pre- and post-pruning are both methods to increase the generality of decision trees.

9.3 Pre-pruning Decision Trees

Pre-pruning a decision tree involves using a 'termination condition' to decide when it is desirable to terminate some of the branches prematurely as the tree is generated.

Each branch of the evolving tree corresponds to an incomplete rule such as

IF $x = 1$ AND $z =$ yes AND $q > 63.5 \ldots$ THEN $\ldots$

and also to a subset of instances currently 'under investigation'.

If all the instances have the same classification, say c1, the end node of the branch is treated by the TDIDT algorithm as a leaf node labelled by c1. Each such completed branch corresponds to a (completed) rule, such as

IF $x = 1$ AND $z =$ yes AND $q > 63.5$ THEN class = c1

If not all the instances have the same classification the node would normally be expanded to a subtree by splitting on an attribute, as described previously. When following a pre-pruning strategy the node (i.e. the subset) is first tested to determine whether or not a termination condition applies. If it does not, the node is expanded as usual. If it does, the subset is treated as a clash set in the way described in Section 9.1, using a 'delete branch', a 'majority voting' or some other similar strategy. The most common strategy is probably the 'majority voting' one, in which case the node is treated as a leaf node labelled with the most frequently occurring classification for the instances in the subset (the 'majority class').

The set of pre-pruned rules will wrongly classify some of the instances in the training set. However, the classification accuracy for the test set may be greater than for the unpruned set of rules.

There are several criteria that can be applied to a node to determine whether or not pre-pruning should take place. Two of these are:

- **Size Cutoff**
  Prune if the subset contains fewer than say 5 or 10 instances
- **Maximum Depth Cutoff**
  Prune if the length of the branch is say 3 or 4.

Figure 9.6 shows the results obtained for a variety of datasets using TDIDT with information gain for attribute selection. In each case 10-fold cross-validation is used, with a size cutoff of 5 instances, 10 instances or no cutoff

(i.e. unpruned). Figure 9.7 shows the results with a maximum depth cutoff of 3, 4 or unlimited instead. The 'majority voting' strategy is used throughout.

	No cutoff		5 Instances		10 Instances	
	Rules	% Acc.	Rules	% Acc.	Rules	% Acc.
breast-cancer	93.2	89.8	78.7	90.6	63.4	91.6
contact_lenses	16.0	92.5	10.6	92.5	8.0	90.7
diabetes	121.9	70.3	97.3	69.4	75.4	70.3
glass	38.3	69.6	30.7	71.0	23.8	71.0
hypo	14.2	99.5	11.6	99.4	11.5	99.4
monk1	37.8	83.9	26.0	75.8	16.8	72.6
monk3	26.5	86.9	19.5	89.3	16.2	90.1
sick-euthyroid	72.8	96.7	59.8	96.7	48.4	96.8
vote	29.2	91.7	19.4	91.0	14.9	92.3
wake_vortex	298.4	71.8	244.6	73.3	190.2	74.3
wake_vortex2	227.1	71.3	191.2	71.4	155.7	72.2

Figure 9.6 Pre-pruning With Varying Size Cutoffs

	No cutoff		Length 3		Length 4	
	Rules	% Acc.	Rules	% Acc.	Rules	% Acc.
breast-cancer	93.2	89.8	92.6	89.7	93.2	89.8
contact_lenses	16.0	92.5	8.1	90.7	12.7	94.4
diabetes	121.9	70.3	12.2	74.6	30.3	74.3
glass	38.3	69.6	8.8	66.8	17.7	68.7
hypo	14.2	99.5	6.7	99.2	9.3	99.2
monk1	37.8	83.9	22.1	77.4	31.0	82.2
monk3	26.5	86.9	19.1	87.7	25.6	86.9
sick-euthyroid	72.8	96.7	8.3	97.8	21.7	97.7
vote	29.2	91.7	15.0	91.0	19.1	90.3
wake_vortex	298.4	71.8	74.8	76.8	206.1	74.5
wake_vortex2	227.1	71.3	37.6	76.3	76.2	73.8

Figure 9.7 Pre-pruning With Varying Maximum Depth Cutoffs

The results obtained clearly show that the choice of pre-pruning method is important. However, it is essentially *ad hoc*. No choice of size or depth cutoff consistently produces good results across all the datasets.

This result reinforces the comment by Quinlan [1] that the problem with pre-pruning is that the 'stopping threshold' is "not easy to get right — too high a threshold can terminate division before the benefits of subsequent splits become evident, while too low a value results in little simplification". It would be highly desirable to find a more principled choice of cutoff criterion to use with pre-pruning than the size and maximum depth approaches used previously, and if possible one which can be applied completely automatically without the need for the user to select any cutoff threshold value. A number of possible ways of doing this have been proposed, but in practice the use of post-pruning, to which we now turn, has proved more popular.

9.4 Post-pruning Decision Trees

Post-pruning a decision tree implies that we begin by generating the (complete) tree and then adjust it with the aim of improving the classification accuracy on unseen instances.

There are two principal methods of doing this. One method that is widely used begins by converting the tree to an equivalent set of rules. This will be described in Chapter 11.

Another commonly used approach aims to retain the decision tree but to replace some of its subtrees by leaf nodes, thus converting a complete tree to a smaller pruned one which predicts the classification of unseen instances at least as accurately. This method has several variants, such as *Reduced Error Pruning*, *Pessimistic Error Pruning*, *Minimum Error Pruning* and *Error Based Pruning*. A comprehensive study and numerical comparison of the effectiveness of different variants is given in [2].

The details of the methods used vary considerably, but the following example gives the general idea. Suppose we have a complete decision tree generated by the TDIDT algorithm, such as Figure 9.8 below.

Here the customary information about the attribute split on at each node, the attribute value corresponding to each branch and the classification at each leaf node are all omitted. Instead the nodes of the tree are labelled from A to M (A being the root) for ease of reference. The numbers at each node indicate how many of the 100 instances in the training set used to generate the tree correspond to each of the nodes. At each of the leaf nodes in the complete tree all the instances have the same classification. At each of the other nodes the corresponding instances have more than one classification.

The branch from the root node A to a leaf node such as J corresponds to a decision rule. We are interested in the proportion of unseen instances to which

that rule applies that are incorrectly classified. We call this the *error rate* at node J (a proportion from 0 to 1 inclusive).

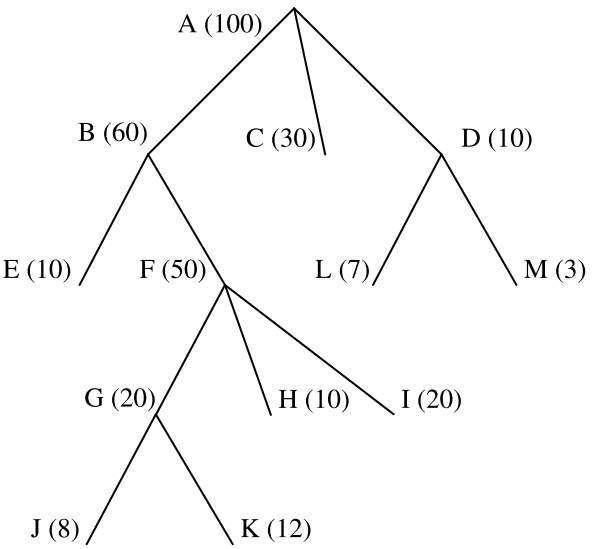

Figure 9.8 Initial Decision Tree

If we imagine the branch from the root node A to an internal node such as G were to terminate there, rather than being split two ways to form the two branches A to J and A to K, this branch would correspond to an incomplete rule of the kind discussed in Section 9.3 on pre-pruning. We will assume that the unseen instances to which a truncated rule of this kind applies are classified using the 'majority voting' strategy of Section 9.1.1, i.e. they are all allocated to the class to which the largest number of the instances in the training set corresponding to that node belong.

When post-pruning a decision tree such as Figure 9.8 we look for non-leaf nodes in the tree that have a descendant subtree of depth one (i.e. all the nodes one level down are leaf nodes). All such subtrees are candidates for post-pruning. If a pruning condition (which will be described below) is met the subtree hanging from the node can be replaced by the node itself. We work from the bottom of the tree upwards and prune one subtree at a time. The method continues until no more subtrees can be pruned.

For Figure 9.8 the only candidates for pruning are the subtrees hanging from nodes G and D.

Working from the bottom of the tree upwards we start by considering the replacement of the subtree 'hanging from' node G by G itself, as a leaf node in

a pruned tree. How does the error rate of the branch (truncated rule) ending at G compare with the error rate of the two branches (complete rules) ending at J and K? Is it beneficial or harmful to the predictive accuracy of the tree to split at node G? We might consider truncating the branch earlier, say at node F. Would that be beneficial or harmful?

To answer questions such as these we need some way of estimating the error rate at any node of a tree. One way to do this is to use the tree to classify the instances in some set of previously unseen data called a *pruning set* and count the errors. Note that it is imperative that the pruning set is *additional to* the 'unseen test set' used elsewhere in this book. The test set must not be used for pruning purposes. Using a pruning set is a reasonable approach but may be unrealistic when the amount of data available is small. An alternative that takes a lot less execution time is to use a formula to estimate the error rate. Such a formula is likely to be probability-based and to make use of factors such as the number of instances corresponding to the node that belong to each of the classes and the prior probability of each class.

Figure 9.9 shows the estimated error rates at each of the nodes in Figure 9.8 using a (fictitious) formula.

Node	Estimated error rate
A	0.3
B	0.15
C	0.25
D	0.19
E	0.1
F	0.129
G	0.12
H	0.05
I	0.2
J	0.2
K	0.1
L	0.2
M	0.1

Figure 9.9 Estimated Error Rates at Nodes in Figure 9.8

Using Figure 9.9 we see that the estimated error rates at nodes J and K are 0.2 and 0.1, respectively. These two nodes correspond to 8 and 12 instances, respectively (of the 20 at node G).

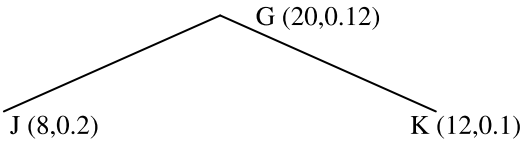

Figure 9.10 Subtree Descending From Node G[1]

To estimate the error rate of the subtree hanging from node G (Figure 9.10) we take the weighted average of the estimated error rates at J and K. This value is $(8/20) \times 0.2 + (12/20) \times 0.1 = 0.14$. We will call this the *backed-up estimate* of the error rate at node G because it is computed from the estimated error rates of the nodes below it.

We now need to compare this value with the value obtained from Figure 9.9, i.e. 0.12, which we will call the *static estimate* of the error rate at that node.[2]

In the case of node G the static value is less than the backed-up value. This means that splitting at node G increases the error rate at that node, which is obviously counter-productive. We prune the subtree descending from node G to give Figure 9.11.

The candidates for pruning are now the subtrees descending from nodes F and D. (Node G is now a leaf node of the partly pruned tree.)

We can now consider whether or not it is beneficial to split at node F (Figure 9.12). The static error rates at nodes G, H and I are 0.12, 0.05 and 0.2. Hence the backed-up error rate at node F is $(20/50) \times 0.12 + (10/50) \times 0.05 + (20/50) \times 0.2 = 0.138$.

The static error rate at node F is 0.129, which is smaller than the backed-up value, so we again prune the tree, giving Figure 9.13.

The candidates for pruning are now the subtrees hanging from nodes B and D. We will consider whether to prune at node B (Figure 9.14).

The static error rates at nodes E and F are 0.1 and 0.129, respectively, so the backed-up error rate at node B is $(10/60) \times 0.1 + (50/60) \times 0.129 = 0.124$. This is less than the static error rate at node B, which is 0.15. Splitting at node B reduces the error rate, so we do not prune the subtree.

We next need to consider pruning at node D (Figure 9.15). The static error rates at nodes L and M are 0.2 and 0.1, respectively, so the backed-up error

[1] In Figure 9.10 and similar figures, the two figures in parentheses at each node give the number of instances in the training set corresponding to that node (as in Figure 9.8) and the estimated error rate at the node, as given in Figure 9.9.

[2] From now on, for simplicity we will generally refer to the 'backed-up' error rate and the 'static error rate' at a node, without using the word 'estimated' every time. However it is important to bear in mind that they are only estimates not the accurate values, which we have no way of knowing.

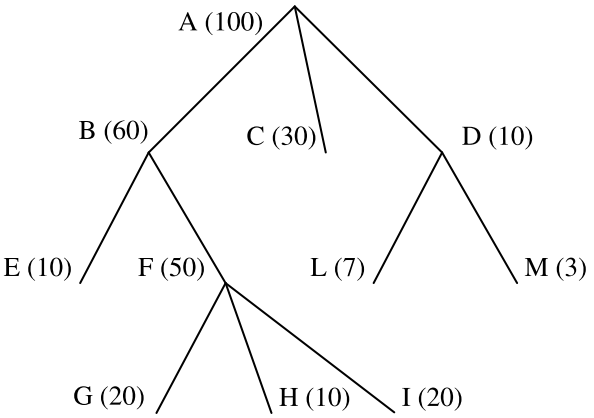

Figure 9.11 Decision Tree With One Subtree Pruned

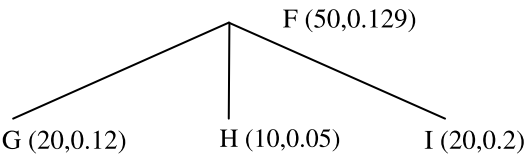

Figure 9.12 Subtree Descending From node F

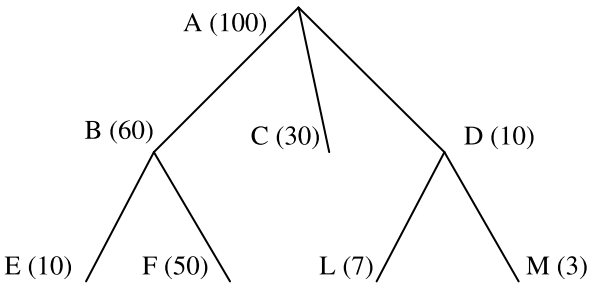

Figure 9.13 Decision Tree With Two Subtrees Pruned

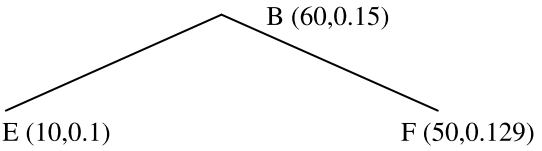

Figure 9.14 Subtree Descending From Node B

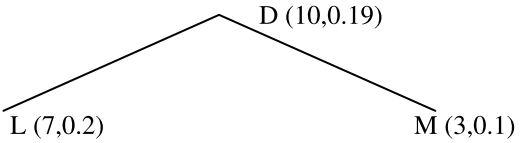

Figure 9.15 Subtree Descending From Node D

rate at node D is $(7/10) \times 0.2 + (3/10) \times 0.1 = 0.17$. This is less than the static error rate at node D, which is 0.19, so we do not prune the subtree. There are no further subtrees to consider. The final post-pruned tree is Figure 9.13.

In an extreme case this method could lead to a decision tree being post-pruned right up to its root node, indicating that using the tree is likely to lead to a higher error rate, i.e. more incorrect classifications, than simply assigning every unseen instance to the largest class in the training data. Luckily such poor decision trees are likely to be very rare.

Post-pruning decision trees would appear to be a more widely used and accepted approach than pre-pruning them. No doubt the ready availability and popularity of the C4.5 classification system [1] has had a large influence on this. However, an important practical objection to post-pruning is that there is a large computational overhead involved in generating a complete tree only then to discard some or possibly most of it. This may not matter with small experimental datasets, but 'real-world' datasets may contain many millions of instances and issues of computational feasibility and scaling up of methods will inevitably become important.

The decision tree representation of classification rules is widely used and it is therefore desirable to find methods of pruning that work well with it. However, the tree representation is itself a source of overfitting, as will be demonstrated in Chapter 11.

9.5 Chapter Summary

This chapter begins by examining techniques for dealing with clashes (i.e. inconsistent instances) in a training set. This leads to a discussion of methods for avoiding or reducing *overfitting* of a decision tree to training data. Overfitting arises when a decision tree is excessively dependent on irrelevant features of the training data with the result that its predictive power for unseen instances is reduced.

Two approaches to avoiding overfitting are distinguished: *pre-pruning* (generating a tree with fewer branches than would otherwise be the case) and *post-*

pruning (generating a tree in full and then removing parts of it). Results are given for pre-pruning using either a size or a maximum depth cutoff. A method of post-pruning a decision tree based on comparing the static and backed-up estimated error rates at each node is also described.

9.6 Self-assessment Exercise for Chapter 9

What post-pruning of the decision tree shown in Figure 9.8 would result from using the table of estimated error rates given below rather than the values given in Figure 9.9?

Node	Estimated error rate
A	0.2
B	0.35
C	0.1
D	0.2
E	0.01
F	0.25
G	0.05
H	0.1
I	0.2
J	0.15
K	0.2
L	0.1
M	0.1

References

[1] Quinlan, J. R. (1993). *C4.5: programs for machine learning*. San Mateo: Morgan Kaufmann.

[2] Esposito, F., Malerba, D., & Semeraro, G. (1997). A comparative analysis of methods for pruning decision trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *19*(5), 476–491.