21
Classifying Streaming Data

21.1 Introduction

One of the most significant developments in Data Mining in recent years has been the huge increase in availability of *streaming data*, i.e. data which arrives (generally in large quantities) from some automatic process over a period of days, months, years or potentially forever.

Some examples of this are:

- Sales transactions in supermarkets
- Data from GPS systems
- Records of changes in share prices
- Logs of telephone calls
- Logs of accesses to webpages
- Records of credit card purchases
- Records of postings to social media
- Data from networks of sensors

For some applications the volume of data received can be as high as tens of millions of records per day, which we can regard as effectively infinite.

As elsewhere in this book we will restrict ourselves to data that is symbolic in nature, as opposed to say images sent from CCTV. We will concentrate on data records that are labelled with a classification and assume that the task is to learn an underlying model in the form of a decision tree. We will further impose the restriction that all attributes are categorical. Continuous

M. Bramer, *Principles of Data Mining*, Undergraduate Topics
in Computer Science, DOI 10.1007/978-1-4471-7307-6_21

(numerical) attributes can be handled by some method similar to that discussed in the context of the TDIDT tree generation algorithm in Chapter 8.

As a concrete example, we can think of some supermarket checkout system that has information about a customer (from a loyalty card, say) and records of his/her recent purchases, where the aim is to predict which customers will buy a particular brand of product when they visit the supermarket and which will not. This information will be used in a future sales campaign perhaps to reinforce the behaviour of those who would normally buy the product or to change the behaviour of as many as possible of those who normally would not. In general there is no need to restrict ourselves to just two classifications and in the examples used in this chapter there will be three possibilities available.

We can think of a process which reads a potentially endless stream of labelled data records (instances) as they arrive and uses them to generate a classification tree piece-by-piece possibly over a long period of time. There are some crucial differences between this and generating classification trees as described earlier in this book:

- Because of the large (potentially infinite) volume of data involved it is fundamental that each record is examined, used to update the information recorded in the evolving tree and then discarded – hence it may only be examined once. The original data records are not stored.
- The processing must take place in real-time and must be rapid enough to avoid creating a large backlog of incoming data waiting to be processed, so time-efficient methods are even more important than usual.
- We cannot wait until training is completed before we start using the classification tree for prediction of the classification of previously unseen data. It is important that at any point it is possible to use the incomplete tree to predict the classification of an unseen instance. It must also be possible to evaluate the quality of the evolving tree's classification performance as the tree-building is going on.

The technique to be described in this chapter is adapted from the VFDT method developed by Domingos and Hulten [1]. VFDT stands for 'Very Fast Decision Trees'. As always with popular methods a large number of variants have been developed. The version described here is the present author's own. It is based closely on [1] but uses a substantially different notation and has some simplifications for ease of explanation. It is certainly not claimed to be the fastest (or best) version of the method but should suffice as a basis for further study or development.

The trees constructed by this and similar methods are often known as *Hoeffding Trees* for a reason that will be explained later. To acknowledge this and to avoid any confusion with VFDT, we will call our variant the *H-Tree*

algorithm.

21.1.1 Stationary v Time-dependent Data

A crucial assumption made in this chapter is that the underlying process we aim to model is fixed, so that having constructed our decision tree model we can use it repeatedly day-after-day, month-after-month etc. without change. We call data arising from such a fixed process *stationary data*.

Although this is a perfectly correct assumption for some types of data, for other types the underlying model varies from time to time, perhaps seasonally. We call such data *time-dependent data*. Dealing with time-dependent data is the subject of Chapter 22.

One consequence of assuming that the data is stationary is that it seems reasonable to expect it to be possible to generate a classification tree from only a (relatively) small number of records that predicts exactly (or almost) as well as one built using many millions of records. The H-Tree algorithm makes this implicit assumption in the way it generates its trees.

21.2 Building an H-Tree: Updating Arrays

As more and more data records are read a classification tree is developed branch by branch, starting with just one node, numbered zero, which will act as the root of the eventual classification tree. Initially it is treated as a leaf node[1]. When certain conditions are met a leaf node can be *split* on an attribute, meaning that a subtree is created below it with one branch for each possible value of the selected attribute.

It will make it easier to explain the H-Tree algorithm and to illustrate the information that needs to be associated with each node in the classification tree if we start by jumping ahead to a future point where there is a partly developed tree, which is shown below as Figure 21.1.

The nodes in the tree are numbered in the order in which they were created. The system variable *nextnode* holds the number of the next node in sequence for possible future use. In this case *nextnode* is 8.

The tree has been created by splitting on attribute *att4* at node 0 (creating

[1]We distinguish between nodes which have or have not previously been split on an attribute. The former are called *internal nodes*; the latter are called *leaf nodes*. We will consider the root node not as a third type of node but as an internal node after it has been split on an attribute and a leaf node before that.

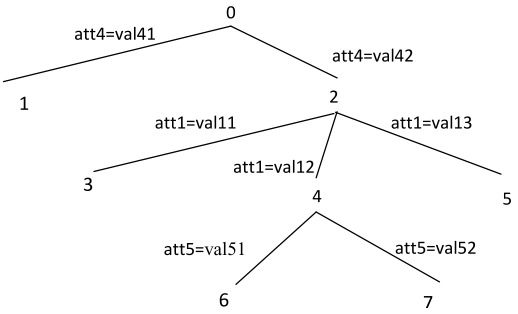

Figure 21.1 A Partly Developed H-Tree

nodes 1 and 2), *att1* at node 2 (creating nodes 3, 4 and 5) and *att5* at node 4 (creating nodes 6 and 7). Each attribute can potentially have any number of values but to simplify the figures we will generally assume they have either two or three. We will adopt the convention that if, say, attribute *att5* has two values we will call them *val51* and *val52*.

Although we are not concerned with the fine detail of implementation in this book, it will make the description of the H-Tree algorithm much easier to follow if we think in terms of maintaining six arrays at each node. These will be discussed in the following sections.

21.2.1 Array *currentAtts*

This is a two-dimensional array. The element *currentAtts*[N] is an array containing the names of all the attributes available for splitting on at node N, listed in a standard order. This is called the *current attributes array* for that node. In the interest of clarity we will generally use phrases such as 'the current attributes array of node N' in preference to 'the array which is the value of *currentAtts*[N]'.

If our data records have the values of seven attributes (an unrealistically small number in most cases) which in standard order are named *att1*, *att2*, *att3*, *att4*, *att5*, *att6* and *att7* then at the root node *currentAtts*[0] is initialised to the array {*att1*, *att2*, *att3*, *att4*, *att5*, *att6*, *att7*}[2].

When a leaf node is 'expanded' by being split on at an attribute, its immediate descendant nodes inherit the current attributes array of the parent

[2]A note on notation. In this chapter array elements are generally shown enclosed in square brackets, e.g. *currentAtts*[2]. However an array containing a number of constant values will generally be denoted by those values separated by commas and enclosed in braces. So *currentAtts*[2] is {*att1*, *att2*, *att3*, *att5*, *att6*, *att7*}.

with the splitting attribute removed. Thus

- *currentAtts*[1] and *currentAtts*[2] are both the array {*att1*, *att2*, *att3*, *att5*, *att6*, *att7*}
- *currentAtts*[3], *currentAtts*[4] and *currentAtts*[5] are all the array {*att2*, *att3*, *att5*, *att6*, *att7*}
- *currentAtts*[6] and *currentAtts*[7] are both the array {*att2*, *att3*, *att6*, *att7*}.

21.2.2 Array *splitAtt*

For an internal node N, i.e. one that has previously been split on an attribute, the array element *splitAtt*[N] is the name of the splitting attribute, so

- *splitAtt*[0] is *att4*
- *splitAtt*[2] is *att1*
- *splitAtt*[4] is *att5*.

All the other nodes are leaf nodes at which there are (by definition) no splitting attributes. The value of *splitAtt*[N] for a leaf node is 'none'.

21.2.3 Sorting a record to the appropriate leaf node

As the data records are received they are processed and discarded, but doing this does not immediately alter the structure (nodes and branches) of the evolving incomplete tree. As each new record comes in and is processed it is *sorted* through the tree to the appropriate leaf node. It is only when a number of conditions are met at a leaf node that a change to the tree by splitting at that node is considered.

To illustrate the sorting process let us assume we have the following record (Figure 21.2):

att1	*att2*	*att3*	*att4*	*att5*	*att6*	*att7*	class
val12	xxx	xxx	val42	val51	xxx	xxx	*c2*

Figure 21.2 Sample Record (Seven Attribute Values Plus a Classification)

Values that are not relevant to the example are denoted by xxx.)

The record passes (notionally) through the tree starting at the root (node 0). From there because the value of *att4* is val42 it passes on to node 2. Then

because *att1* has the value val12 it goes to node 4. Finally, because *att5* has value val51 it arrives at node 6, a leaf. The route taken from root to leaf is 0 to 2 to 4 to 6.

21.2.4 Array *hitcount*

For each leaf node N, *hitcount*[N] is the number of 'hits' on the node since it was created, i.e. the number of records sorted to that leaf node by the process illustrated above. As each new record is sorted to a leaf node L, the value of *hitcount*[L] is increased by 1. The internal nodes 'passed through' in the above example (0, 2 and 4) have their own *hitcount* values, obtained before they were split and became internal nodes. These values are not increased for internal nodes as they play no further part in the tree generation process.

When a new node is created its *hitcount* value is set to zero.

21.2.5 Array *classtotals*

This is another two-dimensional array. If there are three possible classifications named (in a standard order) *c1*, *c2* and *c3* and node N is a leaf node, then *classtotals*[N][*c1*], *classtotals*[N][*c2*] and *classtotals*[N][*c3*] record the number of 'hits' on node N for which the classification is *c1*, *c2* and *c3*, respectively. The sum of these three values is of course *hitcount*[N].

As with array *hitcount*, internal nodes have their own *classtotals* values obtained before they were split and became internal nodes. These values are not increased for internal nodes as they play no part in the tree generation process.

When a new node is created its *classtotals* value is set to zero for all classes.

21.2.6 Array *acvCounts*

This is the most complex of the six arrays. The name stands for 'attribute-class-value counts'. It has four dimensions.

If N is a leaf node then for each attribute A in its current attributes array, *acvCounts*[N][A] is a two-dimensional array which records the number of occurrences of each possible combination of class value and the value of attribute A.

In Figure 21.1, the attributes in the current attributes array for node 6 are *att2*, *att3*, *att6* and *att7*. Assume that *hitcount*[6] is 200 and the values

of *classtotals*[6][*c1*], *classtotals*[6][*c2*] and *classtotals*[6][*c3*] are 100, 80 and 20, respectively.

If attribute *att2* has three values, here are some possible values for the array elements at node 6 for attribute *att2*:

- *acvCounts*[6][*att2*][*c1*][*val21*]: 44
- *acvCounts*[6][*att2*][*c1*][*val22*]: 18
- *acvCounts*[6][*att2*][*c1*][*val23*]: 38
- *acvCounts*[6][*att2*][*c2*][*val21*]: 49
- *acvCounts*[6][*att2*][*c2*][*val22*]: 24
- *acvCounts*[6][*att2*][*c2*][*val23*]: 7
- *acvCounts*[6][*att2*][*c3*][*val21*]: 5
- *acvCounts*[6][*att2*][*c3*][*val22*]: 11
- *acvCounts*[6][*att2*][*c3*][*val23*]: 4

It is cumbersome to write the values in this way and it is much easier (and more natural) to depict *acvCounts*[6][*att2*] by a two-dimensional array known as a *frequency table*[3] such as Figure 21.3:

Class	*val21*	*val22*	*val23*
c1	44	18	38
c2	49	24	7
c3	5	11	4

Figure 21.3 Frequency Table for Attribute *att2*

The sum of the values in the *c1* row is the same as *classtotals*[6][*c1*] and similarly for the other classes. The overall total of the numbers in the whole table is the same as *hitcount*[6].

These two-dimensional arrays are precisely what are needed to calculate measures such as Information Gain (discussed in Chapter 5) which are often used to determine which attribute to split on at a node and, as we shall see in Section 21.4, this is how they will be used in developing the H-Tree. As each new record is sorted to a leaf node N, one of the entries in the frequency table corresponding to every attribute in the node's current attributes array is increased by 1.

As for arrays *hitcount* and *classtotals* internal nodes have their own *acvCounts* values, obtained before they were split and became internal nodes.

[3]The row and column headings are provided to assist the reader only. The table itself has 3 rows and 3 columns.

These values are not increased for internal nodes as they play no part in the H-Tree algorithm.

When a new leaf node is created its *acvCounts* value for each combination of class and attribute value for every attribute in its current attributes array is set to zero.

21.2.7 Array *branch*

This final array, together with the *splitAtt* array provides the 'glue' that keeps the tree structure together.

When leaf node N is split on attribute A, for each value of that attribute a branch leading to a new node is created.

For each value V of attribute A:

- $branch[N][A][V]$ is set to *nextnode*
- *nextnode* is increased by 1.

21.3 Building an H-Tree: a Detailed Example

We now return to the starting state with a tree comprising just the root node and will use a detailed example to illustrate the processing involved. As before we will assume that there are three classes, *c1*, *c2* and *c3*, plus seven attributes *att1*, *att2*, ..., *att7*. The number of possible values of an attribute can vary from one to another.

The steps involved in constructing the tree are now as follows.

21.3.1 Step (a): Initialise Root Node 0

We start with a tree with just a single node, numbered zero, and associate five arrays (all except *branch*) with it as described in Section 21.2.

The pseudocode for this is given below[4].

[4]Pseudocode fragments are provided for the benefit of readers who may be interested in developing their own implementations of the H-Tree algorithm. Other readers can safely ignore them.

Pseudocode 1: Initialise Root Node

1. Set *currentAtts*[0] to be an array comprising the names of all the attributes
2. Set *splitAtt*[0] to 'none'
3. Set *hitcount*[0] to zero
4. For each class C, set *classtotals*[0][C] to zero
5. For each attribute A in the node's current attributes array
   - For each combination of class C and value V of attribute A
     - set *acvCounts*[0][A][C][V] to zero
6. Set *nextnode* to 1

21.3.2 Step (b): Begin Reading Records

We now begin reading the incoming records one-by-one, in each case processing the record and then discarding it. (Each record is 'sorted' to node zero as there is only one at present.) We increase one of the numbers in the *classtotals*[0] array by one and the *hitcount* value by one for each record read. We also adjust the contents of the frequency table for each attribute by adding one to one of the values in the table, depending on the combination of the value of the attribute and the specified classification for that record.

Let us assume that by the time the 100th record has been read the *classtotals*[0] array contains $\{63, 17, 20\}$. The sum of the three values in the array will of course be 100. At this stage the frequency table for attribute *att6* might contain the following (Figure 21.4). There will be frequency tables similar to this for each of the other attributes. In each case the right-most column (row sums) will be the same.

Class	*val61*	*val62*	*val63*	*val64*
c1	32	18	4	9
c2	0	5	5	7
c3	0	10	7	3

Figure 21.4 Frequency Table for Attribute *att6*

It will help to interpret this if we add an extra row containing the sum of the numbers in each of the existing columns and a further column containing the sum of the numbers in each row. Note that these additional values do not need to be stored. They are calculated from the values in the stored $3 * 4$ table

as and when needed.

The augmented frequency table for attribute *att6* at node 0 looks like this (Figure 21.5):

Class	*val61*	*val62*	*val63*	*val64*	Row sums
c1	32	18	4	9	63
c2	0	5	5	7	17
c3	0	10	7	3	20
Column sums	32	33	16	19	100

Figure 21.5 Augmented Frequency Table for Attribute *att6*

The number in the bottom right-hand corner is the sum of the numbers in the bottom (column sums) row, which is the same as the sum of the numbers in the right-most (row sums) column. This overall total is the same as the number of records sorted to node 0 – in this case 100.

The other values in the right-most column show how many instances have been classified with each of the three possible classes. They are the same as the values in the *classtotals* array, i.e. $\{63, 17, 20\}$.

The first four numbers in the bottom row are the column sums. They show that attribute *att6* had the value *val61* 32 times, *val62* 33 times, *val63* 16 times and *val64* 19 times.

21.3.3 Step (c): Consider Splitting at Node 0

To develop the tree we need to split on an attribute at the root node, but we clearly cannot do this after just one record has been read as the resulting tree would be essentially arbitrary and likely to have extremely low predictive power. Instead we wait until a specified number of records have been sorted to node zero[5] and then make a decision on whether or not to split on an attribute and if so which one.

The specified number is denoted by G and is sometimes referred to as the *grace period*. In this chapter the same value will be used at each leaf node as the tree evolves, but it would be possible to use a larger value at some points in the processing (e.g. at or near the root of the tree) than at others. To make the numbers reasonably small in our examples we will use a value of 100 for G.

[5] As initially there are no other nodes, all incoming records will be sorted there.

Once G, i.e. 100 records have been sorted to node 0, we next consider splitting at that node provided that the records sorted to it have more than one classification. If all the classifications were the same we would continue receiving and processing records until the next 100 were received at which time splitting would be considered again. In this example the classifications are not all identical so we go on to determine which attribute to split on, but with 'no split' as one of the options. At present we will assume that we will definitely split and will choose the attribute to split on using a method such as the maximising Information Gain method described in Chapter 5, or one of the other similar methods that use a frequency table for each attribute.

21.3.4 Step (d): Split on Root Node and Initialise New Leaf Nodes

We will say that it is decided to split at node 0 using attribute *att6*. This gives us four branches (one per value of *att6*) leading to four new nodes, numbered from 1 to 4, as shown in Figure 21.6[6]. To achieve this we start by setting array element *branch*[0][*att6*][*val61*] to *nextnode*, i.e. 1, and increasing *nextnode* by 1. We then create the other three branches in a similar way. The value of *nextnode* is now 5.

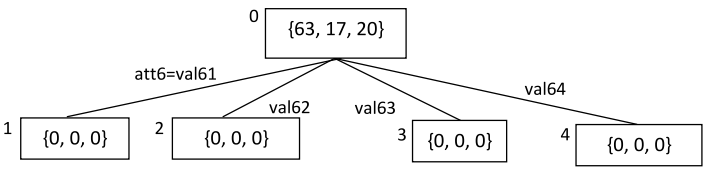

Figure 21.6 H-Tree After Splitting on Node 0 (with current attributes array for each node shown)

Pseudocode for the process of splitting on an attribute and initialising the resulting new nodes is given below.

[6]In Figures 21.6, 21.8 and 21.9 we depart from our usual notation for trees and show the values that are in the *classtotals* array for each node.

Pseudocode 2: Split at Node L Using Attribute A

1. Set $splitAtt[L]$ to A.
2. For each value V of attribute A
   - Set $branch[L][A][V]$ to $nextnode$ (to create a new leaf node)
   - Set $currentAtts[nextnode]$ to be the same array as $currentAtts[L]$ with attribute A removed
   - Initialise arrays *splitAtt*, *hitcount*, *classtotals* and *acvCounts* for node *nextnode* as for Pseudocode 1, steps 2 to 5
   - Increase *nextnode* by 1

For each of the new nodes 1 to 4 the *classtotals* array is initialised to $\{0, 0, 0\}$, the value of *hitcount* is set to zero and the value of *splitAtt* is set to 'none'.

We create a current attributes array for each of the new nodes, by taking the current attributes array from the parent node (node 0) and removing attribute *att6* from each. This gives all of them the array {*att1*, *att2*, *att3*, *att4*, *att5*, *att7*}. These are the attributes available for any future splitting at those nodes.

We also create a frequency table for each attribute at each of the three nodes.

For attribute *att2* which has two values, *val21* and *val22*, the values of frequency tables *acvCounts*[1][*att2*], *acvCounts*[2][*att2*], *acvCounts*[3][*att2*] and *acvCounts*[4][*att2*] are all initially the same, i.e. (Figure 21.7):

Class	*val21*	*val22*
c1	0	0
c2	0	0
c3	0	0

Figure 21.7 Frequency Table for Attribute *att2*

Creating the new frequency tables this way may seem innocuous but it is in fact a major departure from the Information Gain method and the other methods described in Chapter 5 for situations where all the data is available. Ideally we would like the new frequency tables to begin with counts of all the class / attribute value combinations for all the relevant records so far received. However there is no way of doing this. We would need to re-examine the original data, but it has all already been discarded. The best we can do is to start with a table with all zero values for each attribute at each of the nodes, but this will inevitably mean that the tree eventually generated will be different

– perhaps very significantly so – from that which would be generated by the methods given in Chapter 5 if we were somehow able to capture and store all the data.

Before going on it is important to understand that the role of leaf nodes in this method is entirely different from their role in algorithms such as TDIDT described earlier in this book. Here a leaf node has no descendants but not necessarily a single classification such as $\{84, 0, 0\}$. The mixture of classifications at a leaf node changes as more records are received and a leaf node can subsequently acquire descendants as the tree building continues.

All the leaf nodes that we will see in this chapter are *expandable* leaf nodes, i.e. ones for which there is a non-empty current attributes array signifying that there are still attributes available for splitting if required. It is possible for a leaf node to be at the end of a long path comprising the same number of branches as there are attributes, with one attribute/value combination pair per branch, although this is not very likely if the number of attributes is large. We call such nodes *non-expandable* leaf nodes. The current attributes array for a non-expandable leaf node will be empty. Naturally we do not consider splitting at a leaf node that is non-expandable.

21.3.5 Step (e): Process the Next Set of Records

We now go on to read and process the next set of records. As each one is read, it is *sorted* to the correct leaf node. We can think of the instance starting at node 0, and then falling down to node 1, 2, 3 or 4 depending on the value of attribute *att6*. In a larger tree it might fall further down to a lower level, but in all cases the instance will be sorted to one of the leaf nodes.

As each record is sorted to a leaf node the values in the *hitcount* and *classtotals* arrays and the frequency table for each attribute in the current attributes array are updated at that node.

Pseudocode for processing a record is given below.

Pseudocode 3: Process Record R with Classification C

1. Set N to the number of the root node
2. While $splitAtt[N]$!= 'none'
   - Set A to $splitAtt[N]$
   - Set V to value of attribute A in record R
   - Set N to $branch[N][A][V]$
3. Set $L = N$
4. Update arrays at leaf node L:
   - Increase $hitcount[L]$ by 1
   - Increase $classtotals[L][C]$ by 1
   - For each attribute A in the current attributes array for node L
     - set V to the value that the attribute has in record R
     - increase $acvCounts[L][A][C][V]$ by 1
5. If suitable conditions are met consider splitting at node L

Step 5 will be developed further as this section progresses.

Going back to our example, our 'grace period' G is 100 and by the time the 100th record is sorted to node 2 we will assume that the *classtotals* arrays for each of the five nodes are the following:

Node 0: $\{63, 17, 20\}$
Node 1: $\{12, 8, 0\}$*
Node 2: $\{87, 10, 3\}$*
Node 3: $\{0, 0, 0\}$*
Node 4: $\{40, 10, 20\}$*

Leaf nodes are indicated by an asterisk. The *classtotals* array for node 0 has not changed as it is no longer a leaf node. When the records were sorted to leaf nodes 1, 3 and 4 there were fewer than 100 records sorted to each of those nodes so no consideration was given to splitting on an attribute there. Now G (i.e. 100) records have been sorted to node 2 a decision needs to be made whether to split there or not.

21.3.6 Step (f): Consider Splitting at Node 2

We now consider splitting at node 2. The records that have been sorted to that node have more than one classification, so we go on to calculate the Information Gain (or other measure) for each attribute in the node's current attributes array.

This time we will say that attribute *att2*, which has two values, is chosen

for splitting, giving the new but still incomplete tree structure shown in Figure 21.8. The new nodes (5 and 6) all have *classtotals* arrays with the value $\{0, 0, 0\}$, *hitcount* values of zero and frequency tables containing all zero values. The *classtotals* and *hitcount* arrays and frequency tables at the other nodes are left unchanged.

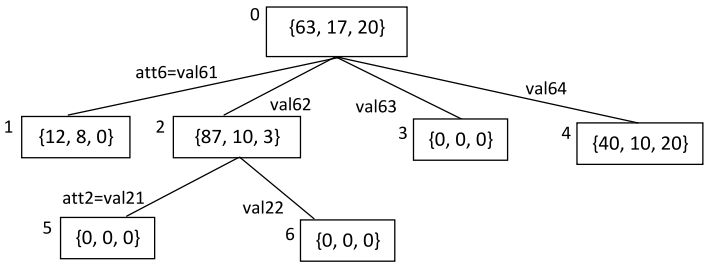

Figure 21.8 H-Tree After Splitting on Node 2

The new nodes 5 and 6 will all have a *currentAtts* array containing {*att1*, *att3*, *att4*, *att5*, *att7*}. The value of *splitAtt* at each node will be 'none'.

21.3.7 Step (g): Process the Next Set of Records

We now continue to read records, sorting each one to the appropriate leaf node, adjusting the values of *classtotals* and the contents of the frequency tables for each attribute each time.

We will assume that at some stage the total number of records sorted to node 4 in Figure 21.8 increases to 100, the value of G, and that at that stage the *classtotals* arrays for the records sorted to the seven nodes are the following:

Node 0: $\{63, 17, 20\}$
Node 1: $\{22, 9, 1\}$*
Node 2: $\{87, 10, 3\}$
Node 3: $\{8, 7, 15\}$*
Node 4: $\{45, 20, 35\}$*
Node 5: $\{25, 12, 31\}$*
Node 6: $\{0, 0, 0\}$*

Leaf nodes are again indicated by an asterisk. The *classtotals* arrays for nodes 0 and 2 have not changed as they are no longer leaf nodes.

We next consider splitting at node 4. If the decision is no, we simply go on to read further records. However we will assume that this time attribute *att2* is chosen for splitting (as it was at node 2), giving the new tree structure shown

in Figure 21.9.

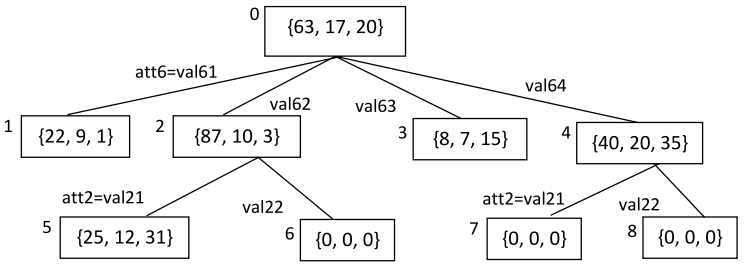

Figure 21.9 H-Tree After Splitting on Node 4

We carry on in this way, expanding at most one leaf node at each stage. Depending on the number of current attributes we have and the number of values each of them has, we may eventually reach a tree where every leaf node is non-expandable. If that happens the tree is now fixed and cannot later be altered. If there are quite a large number of attributes this is highly unlikely to happen. It is far more likely that the tree will initially develop fairly rapidly (or as rapidly as the size of the grace period allows) and then stabilise, i.e. stop evolving or change only slightly as further records are processed.

21.3.8 Outline of the H-Tree Algorithm

The algorithm described above can be summarised by a very simple 'main' algorithm and a revised version of pseudocode fragment 3. Here and subsequently the parts altered are shown in **bold**.

Outline of H-Tree Algorithm

1. Initialise root node (see pseudocode 1)
2. For each record R with classification C that arrives to be processed
   - Process record R with classification C (see pseudocode 3)

Pseudocode 3: Process Record *R* with Classification *C* (version 2)

1. Set N to the number of the root node
2. While *splitAtt*[N] != 'none'
   - Set A to *splitAtt*[N]
   - Set V to value of attribute A in record R
   - Set N to *branch*[N][A][V]
3. Set $L = N$
4. Update arrays at leaf node L:
   - Increase *hitcount*[L] by 1
   - Increase *classtotals*[L][C] by 1
   - For each attribute A in the current attributes array for node L
     - set V to the value that the attribute has in record R
     - increase *acvCounts*[L][A][C][V] by 1
5. **If conditions are met at node *L***
   - **Determine whether to split on an attribute at node *L***
   - **If answer is 'yes', with splitting attribute *A*, split at node *L*, using attribute *A* (see pseudocode 2)**

This leaves two important issues:

(a) What conditions need to be met at node L before checking for a possible split at that node?
(b) How to determine which attribute (if any) to split on at node L?

In answer to (a) there are three conditions that must be met.

(i) The value of *hitcount*[L] is a multiple of G.

It would be possible to consider splitting at a node every time a new record was sorted to it but that would be computationally expensive and might lead to very poor splits being made. Once a split is made at a node the node cannot be 'unsplit' or 'resplit' on some different attribute. This implies that we need to be cautious and only consider splitting at a leaf node after a significant number of records have been sorted to it.

Note that the test is 'a multiple of G', rather than 'equal to G'. If we consider splitting when the *hitcount* is G and decide not to do so, we do not consider it again at that node until a further G hits have been received there.

(ii) The classifications of the records that have been sorted to the node are not all the same.

If all the classifications are identical, say *c1*, the entropy at the leaf node is zero and the entropy resulting from splitting on any attribute will also inevitably be zero. For example if the *classtotals* array is $\{100, 0, 0\}$ the frequency table for attribute *att6*, which has four values, might look like this (Figure 21.10):

Class	*val61*	*val62*	*val63*	*val64*	Row sums
c1	28	0	30	42	100
c2	0	0	0	0	0
c3	0	0	0	0	0
Column sums	28	0	30	42	100

Figure 21.10 Frequency Table for Attribute *att6*

This table has an entropy value of zero. The contribution from each of the non-zero values in the main body of the table will be cancelled out by the contribution from the corresponding column sum. This is a general feature of any frequency table with either zero or one positive entries in each column of the table. The result is that the entropy will be identical (i.e. zero) for each attribute so there will be no basis for making a split on one attribute rather than another and no benefit at all from splitting.

(iii) The node must be expandable, i.e. its current attributes array must not be empty.

This leads to another revised version of pseudocode fragment 3.

Pseudocode 3: Process Record *R* with Classification *C* (version 3)

1. Set N to the number of the root node
2. While $splitAtt[N]$!= 'none'
   - Set A to $splitAtt[N]$
   - Set V to value of attribute A in record R
   - Set N to $branch[N][A][V]$
3. Set $L = N$
4. Update arrays at leaf node L:
   - Increase $hitcount[L]$ by 1
   - Increase $classtotals[L][C]$ by 1
   - For each attribute A in the current attributes array for node L
     - set V to the value that the attribute has in record R
     - increase $acvCounts[L][A][C][V]$ by 1
5. **If $hitcount[L]$ is a multiple of G and the classifications of the records sorted to node L are not all the same and $currentAtts[L]$ is not empty**
   - Determine whether to split on an attribute at node L **(see pseudocode 4)**
   - If answer is 'yes', with splitting attribute A, split at node L, using attribute A (see pseudocode 2)

Issue (b), i.e. how to determine which attribute to split on (if any) at node L forms the topic of the next two sections. (At step 5 there is a forward reference to pseudocode fragment 4, which will be given in Section 21.5.)

21.4 Splitting on an Attribute: Using Information Gain

We will start by assuming that we definitely want to split on an attribute at leaf node L and illustrate the method of doing so using the Information Gain criterion, as described in Chapters 5 and 6. Other splitting criteria such as those also mentioned in Chapter 6 (Gini Index, Gain Ratio etc.) may also be used but for definiteness we will assume in this chapter that the criterion used is always Information Gain.

We will not repeat the detailed explanation in previous chapters but will summarise the method for splitting at leaf node L briefly here:

- Calculate the 'initial entropy' E_{start} at node L.
- For each attribute A
- Calculate the (weighted average) entropy E_{new} at the new nodes that would result from splitting on attribute A
- Calculate the value of $E_{start} - E_{new}$. This value is the Information Gain.
- Split on the attribute with the highest value of Information Gain.

As an example, suppose the *classtotals* array at node L, i.e. *classtotals*$[L]$, contains the values $\{100, 150, 250\}$. We will illustrate the method without justification. Full details are given in Chapters 5 and 6.

The total class count is 500, so we calculate E_{start} at node L as

$$-(100/500) * \log_2(100/500) - (150/500) * \log_2(150/500)$$
$$-(250/500) * \log_2(250/500)$$

which is $0.464 + 0.521 + 0.5 = 1.4855$ to 4 decimal places.

It can be proved that Information Gain must always be positive or zero.

To calculate the entropy resulting from splitting on an attribute, say attribute *att3* which has three values, we use a frequency table, which in our notation would be the two-dimensional array *acvCounts*$[L][att3]$. Augmented with column sums and headings it might look like this (Figure 21.11):

	att3 = val31	*att3 = val32*	*att3 = val33*
Class *c1*	64	4	32
Class *c2*	50	50	50
Class *c3*	200	25	25
Column sum	314	79	107

Figure 21.11 Augmented Frequency Table for Attribute *att3*

Note that the row sums for classes *c1*, *c2* and *c3* are 100, 150, 250, respectively, and the overall total of all the values in the main body of the table (i.e. not including those in the column sum row) is 500.

We now form a sum as follows:

(a) For every non-zero value V in the main body of the table subtract $V * \log_2 V$.
(b) For each non-zero value S in the column sum row add $S * \log_2 S$.
(c) Finally divide the total sum by the overall total of all the values in the main body of the table.

This gives us the value of E_{new}:

$$-64 * \log_2 64 - 4 * \log_2 4 - 32 * \log_2 32$$
$$-50 * \log_2 50 - 50 * \log_2 50 - 50 * \log_2 50$$
$$-200 * \log_2 200 - 25 * \log_2 25 - 25 * \log_2 25$$
$$+314 * \log_2 314 + 79 * \log_2 79 + 107 * \log_2 107$$

all divided by 500.

This gives $Enew = 1.3286$ to 4 decimal places. The value of Information Gain for splitting on attribute *att3* at node L is $E_{start} - E_{new} = 0.1569$ to 4 decimal places.

We calculate the Information Gain for all the attributes in the current attributes array for node L, i.e. $currentAtts[L]$, and select the attribute which gives the highest value.

21.5 Splitting on An Attribute: Using a Hoeffding Bound

We now come to the issue of whether or not to split at a leaf node.

A built-in problem with the evolving classification tree approach described in this chapter is that once a leaf node is split on an attribute it cannot be 'unsplit' back to a leaf node or 'resplit' using some other attribute. This makes the outcome particularly sensitive to the choices of splitting attribute near the top of the tree, especially the very first split on the original root/leaf node, node 0.

One way of avoiding poor splits at an early stage is to start with a value of G much larger than its ultimately intended value and only reduce it to its 'correct' value once the first few splits have been made. Another is to start the tree generation process not with just one node but with part or all of a classification tree generated using a method such as TDIDT (described in Chapter 4) using say 10,000 initial records. All leaf nodes in such a 'startup tree' would be subject to splitting in the usual way as more records arrive.

As the evolving classification tree is sensitive to the order of the initial records processed it is important to ensure as far as possible that there is nothing special or unusual about the initial records, e.g. the data stream does not happen to start with 5,000 records with identical classifications. If the data is taken from an artificial source, rather than a live stream of input it may be worthwhile to randomise the order of the records before use.

However these startup issues are handled, the general problem remains and can apply at any point in an evolving tree: there is a risk of making an inappropriate split at a leaf node when it would give more accurate

predictions to leave it as a leaf and possibly split at that node on a different attribute at a later stage when more records have been sorted to it. We need a method of deciding when we should and should not split at a node that is more cautious than the method adopted so far.

In Section 21.3.8 we gave three conditions that must be met before we consider splitting on an attribute at node L. Here we will add a further condition: splitting only takes place if the best value of the measure that can be obtained, assumed to be by splitting on attribute X, is significantly better than the second best value, assumed to be obtained by splitting on attribute Y. We will assume for the remainder of this chapter that the measure used is Information Gain and will write these two values as $IG(X)$ and $IG(Y)$ respectively.

By 'significantly better' we mean that the difference between $IG(X)$ and $IG(Y)$ is greater than some value that we will call a *bound*. If the difference is smaller than the bound we will consider the difference between splitting on X and splitting on Y not to be significant and will leave the leaf node unexpanded. (It may be expanded at a later stage.)

The bound used for the difference between $IG(X)$ and $IG(Y)$ is not a fixed number, but depends on a number of factors. The one we will use is called the *Hoeffding Bound*, which finally explains the term Hoeffding Tree used at the start of the chapter. The Hoeffding Bound was developed by the Finnish statistician Wassily Hoeffding [2] in a different context and was adapted for use with classification trees by Domingos and Hulten [1]. In its revised form (and in our notation) *Hoeffding's inequality* states that with probability *Prob* attribute X is the correct choice of attribute to split on at a leaf node, based on *nrec* records sorted to that node, provided that $IG(X) - IG(Y)$ is greater than a value that depends on *Prob* and *nrec*. That value is called the *Hoeffding Bound*.

Before going on to define the Hoeffding Bound we will first set out three criteria that any such bound needs to meet:

(a) The larger the range of values that the measure we use can take, the larger the bound needs to be.
(b) The higher the value we choose for probability *Prob*, the larger the bound needs to be.
(c) The larger the number of records on which the split is based, the more reliable the choice of splitting attribute is and so the smaller the bound needs to be.

The formula for the Hoeffding Bound meets all these criteria. The bound is denoted by ϵ (the Greek letter 'epsilon') and is defined by the formula

$$\epsilon = R * \sqrt{\frac{ln(1/\delta)}{2 * nrec}}$$

In this formula *nrec* is the number of records sorted to the given node, i.e. *hitcount*[*L*] in our array notation. This is usually the same as *G*, the 'grace period' but may be a multiple of *G*. The Greek letter δ (pronounced 'delta') is used to represent the value of 1-*Prob*.

Figure 21.12 shows the value of $\ln(1/\delta)$ for various common values of the probability *Prob*. The ln function is called the *natural logarithm function* and is often written as $\log_e$.

Probability *Prob*	δ = 1-*Prob*	$\ln(1/\delta)$
0.9	0.1	2.3026
0.95	0.05	2.9957
0.99	0.01	4.6052
0.999	0.001	6.9078

Figure 21.12 Values of $\ln(1/\delta)$ for Various Probability Values

The value *R* corresponds to the range of values of the measure we are using to decide which attribute to split on, which we will assume is Information Gain. The smallest value of Information Gain that can be obtained by splitting at a node is zero and the largest value is the 'initial entropy' E_{start} at the node. We will use the value of E_{start} for *R*.

Number of classes *c*	Maximum value of *R* = $\log_2 c$
2	1
3	1.5850
4	2
5	2.3219
6	2.5850
7	2.8074
8	3

Figure 21.13 Maximum Values of *R* for Various Numbers of Classes

The largest value that R can ever take occurs when all the classes are equally frequent at a node, in which case, assuming there are c classes, the value of the entropy and hence the value of R is $\log_2 c$. Even with a very large number of streaming records the number of classifications is likely to be a small number. The maximum values of R corresponding to some small values of c are given in Figure 21.13.

Putting all this together, if we have three classes distributed evenly (so $R = 1.5850$) and want to be 95% certain that attribute X is the best choice, Figure 21.14 shows the value of the bound ϵ for each of several possible values of *nrec*.

Number of records *nrec*	100	200	1,000	2,000	10,000	20,000
Bound ϵ	0.1940	0.1372	0.0613	0.0434	0.0194	0.0137

Figure 21.14 Values of Hoeffding Bound for $R = 1.5850$ and $Prob = 0.95$

For each value of *nrec*, only if the difference between the information gains of the best attribute X and the second best attribute Y is greater than ϵ will a split on X be made. As the number of records, *nrec*, becomes larger the Hoeffding Bound requirement becomes progressively easier to meet.

If we want to adopt a more cautious approach, i.e. require a higher probability of certainty before splitting, the value of the bound will be correspondingly larger (making it more difficult to achieve). Figure 21.15 shows the values of the Hoeffding Bound for $R = 1.5850$ and $Prob = 0.999$ for different values of *nrec*.

Number of records *nrec*	100	200	1,000	2,000	10,000	20,000
Bound ϵ	0.2946	0.2083	0.0931	0.0659	0.0295	0.0208

Figure 21.15 Values of Hoeffding Bound for $R = 1.5850$ and $Prob = 0.999$

We can consider the value of ϵ as a multiple of the value of the range R. If we denote ϵ/R by *mult* then

$$mult = \sqrt{\frac{ln(1/\delta)}{2 * nrec}}$$

Rearranging this we have

$$nrec = \frac{ln(1/\delta)}{2 * mult^2}$$

A value of $mult = 0.1$ (indicating that ϵ is 10% of R) is given when $nrec = 115$ for $Prob = 0.9$, $nrec = 150$ for $Prob = 0.95$, $nrec = 230$ for $Prob = 0.99$ and $nrec = 345$ for $Prob = 0.999$.

The choice of values for probability $Prob$ and grace period G determines the shape and size of the evolving tree. The most appropriate settings are likely to vary from one application to another.

There are two further adjustments that we can choose to make to the splitting process:

- We might decide only to split if $IG(X)$, the measure associated with the best attribute X, is greater than some specified multiple of the value of R.
- If we decide to make a split we might eliminate any attributes with low values of the measure from the *currentAtts* arrays of the descendant nodes. If there are many attributes, this may speed up subsequent processing of that part of the tree considerably.

This concludes the description of the process of deciding whether or not to spilt on an attribute at node L and if so which attribute to choose. It can be summarised by the following pseudocode.

Pseudocode 4: Choose Attribute to Split on at Node *L*

1. Calculate the initial entropy at node L
2. For every attribute *att* in node L's current attributes array calculate the Information Gain $IG(att)$
3. Denote the attributes with the largest and second largest IG values by X and Y respectively
4. Calculate the Hoeffding Bound ϵ for node L:
   - Set R to the initial entropy at node L
   - Set δ to the value of 1-$Prob$
   - Set $nrec$ to $hitcount[L]$
   - Calculate $\epsilon = R * \sqrt{\frac{ln(1/\delta)}{2 * nrec}}$
5. If $IG(X) - IG(Y) > \epsilon$ return X, otherwise return 'none'

If there is only one attribute in node L's current attributes list, attribute Y will be taken to be the 'null attribute', equivalent to not splitting at all, with an Information Gain value of zero.

21.6 H-Tree Algorithm: Final Version

The final form of the outline 'main' algorithm given in Section 21.3.8 is now as follows. It has an additional initial step to set the values of G and *Prob*.

H-Tree Algorithm: Final Version

1. **Set values of *G* and *Prob***
2. Initialise root node (see pseudocode 1)
3. For each record R with classification C that arrives to be processed
   - Process record R with classification C (see pseudocode 3)

The main algorithm uses pseudocode fragments 1 and 3, the latter of which uses numbers 2 and 4.

The final versions of all four pseudocode fragments are repeated here for ease of reference.

Pseudocode 1: Initialise Root Node

1. Set *currentAtts*[0] to be an array comprising the names of all the attributes
2. Set *splitAtt*[0] to 'none'
3. Set *hitcount*[0] to zero
4. For each class C, set *classtotals*[0][C] to zero
5. For each attribute A in the node's current attributes array
   - For each combination of class C and value V of attribute A
     - set *acvCounts*[0][A][C][V] to zero
6. Set *nextnode* to 1

Pseudocode 2: Split at Node *L* Using Attribute *A*

1. Set *splitAtt*[L] to A.
2. For each value V of attribute A
   - Set *branch*[L][A][V] to *nextnode* (to create a new leaf node)
   - Set *currentAtts*[*nextnode*] to be the same array as *currentAtts*[L] with attribute A removed
   - Initialise arrays *splitAtt*, *hitcount*, *classtotals* and *acvCounts* for node *nextnode* as for Pseudocode 1, steps 2 to 5
   - Increase *nextnode* by 1

Pseudocode 3: Process Record *R* with Classification *C*

1. Set N to the number of the root node
2. While $splitAtt[N]$!= 'none'
   - Set A to $splitAtt[N]$
   - Set V to value of attribute A in record R
   - Set N to $branch[N][A][V]$
3. Set $L = N$
4. Update arrays at leaf node L:
   - Increase $hitcount[L]$ by 1
   - Increase $classtotals[L][C]$ by 1
   - For each attribute A in the current attributes array for node L
     - set V to the value that the attribute has in record R
     - increase $acvCounts[L][A][C][V]$ by 1
5. If $hitcount[L]$ is a multiple of G and the classifications of the records sorted to node L are not all the same and $currentAtts[L]$ is not empty
   - Determine whether to split on an attribute at node L (see pseudocode 4)
   - If answer is 'yes', with splitting attribute A, split at node L, using attribute A (see pseudocode 2)

Pseudocode 4: Choose Attribute to Split on at Node *L*

1. Calculate the initial entropy at node L
2. For every attribute att in node L's current attributes array calculate the Information Gain $IG(att)$
3. Denote the attributes with the largest and second largest IG values by X and Y respectively
4. Calculate the Hoeffding Bound ϵ for node L:
   - Set R to the initial entropy at node L
   - Set δ to the value of 1-$Prob$
   - Set $nrec$ to $hitcount[L]$
   - Calculate
     $$\epsilon = R * \sqrt{\frac{ln(1/\delta)}{2 * nrec}}$$
5. If $IG(X) - IG(Y) > \epsilon$ return X, otherwise return 'none'

21.7 Using an Evolving H-Tree to Make Predictions

In this section we will look at the issue of predicting the classification of an unseen instance from an evolving classification tree.

One of the unavoidable requirements forced on us by streaming data is to be able to use the classification tree while it is still incomplete. This runs the risk that any prediction we obtain from it might have been different if we had made it at a later stage, but realistically we have to accept this.

Suppose that we have the incomplete tree shown in Figure 21.9 in Section 21.3 and we want to classify an unseen instance that is sorted to node 5, i.e. has attribute values *att6* = *val62* and *att2* = *val21*. How should we classify it? We could use the *classtotals* array for node 5 and take the largest class, but that approach would not be appropriate in other cases such as for an unseen instance sorted to node 7, which has *a classtotals* array comprising all zeroes.

A simple but effective approach is to look at all the nodes on the path from the root node (node 0) down to node 5 and combine their *classtotals* arrays. The three arrays are $\{63, 17, 20\}$ at node 0, $\{87, 10, 3\}$ at node 2 and $\{25, 12, 31\}$ at node 5, giving combined counts of $\{175, 39, 54\}$. We can accumulate these values in an array *totalClassCounts* with three elements, one for each class.

For our example, the largest count in the *totalClassCounts* array is for the first class, i.e. *c1*, so that is the value predicted for the unseen instance. As more (labelled) records arrive at node 5, the values in array *classtotals* will change and some other class may become the majority one.

Pseudocode for this process is given below.

H-Tree Prediction Algorithm for Record R

1. Set array *totalClassCounts* to zero for each class
2. Set N to the number of the root node
3. While *splitAtt*[N] != 'none'
   - Add values in array *classtotals*[N] to those in *totalClassCounts* for each class in turn
   - Set A to *splitAtt*[N]
   - Set V to value of attribute A in record R
   - Set N to *branch*[N][A][V]
4. Set $L = N$
5. Add values in array *classtotals*[L] to those in *totalClassCounts* for each class in turn
6. Predict the class which has the largest value in *totalClassCounts*

21.7.1 Evaluating the Performance of an H-Tree

The above method can be adapted to give us a way of evaluating the performance of an evolving H-Tree. One possibility is to keep back a file of records that are not used for building the tree and then use them as a test set, i.e. treat them as if we did not know the classifications and record the predicted and actual classes in a confusion matrix[7]. (In this case we do not change the values of the arrays at the nodes to which they are sorted.)

After the last test record has been examined we might have a confusion matrix such as the one shown in Figure 21.16 (assuming that the test file has 1,000 records).

	Predicted Class		
Actual Class	*c1*	*c2*	*c3*
c1	263	2	21
c2	2	187	8
c3	4	9	504

Figure 21.16 Confusion Matrix

From this we can calculate predictive accuracy or other measures of

[7]Confusion matrices were described in Chapter 7.

accuracy and track how the values vary if we repeat the test every hour, every day etc.

A second possibility is to evaluate the performance of the tree each time a node is expanded, using the same records that were used to develop the tree, rather than a separate test set, but recording in the confusion matrix only the actual versus predicted classifications of those records that have arrived since the previous split. (With this approach, the values in arrays *hitcount*, *classtotals* and *acvCounts* are updated for each new record.)

We create a confusion matrix with all zero values immediately after each split on a node. Using the example in Figure 21.9 (Section 21.3), if the next record that arrives is sorted to node 5 with actual classification *c2* but predicted classification *c1*, we increase the count in row *c2*, column *c1* of the confusion matrix by one.

This method gives a straightforward way of tracking the performance of the classifier from one split to another in terms of the records that have arrived in that period, rather than on a fixed set of test data. This is probably preferable in view of the phenomenon of *concept drift*, which will be described in Chapter 22.

21.8 Experiments: H-Tree versus TDIDT

According to Domingos and Hulten [1]: 'A key property of the Hoeffding tree algorithm is that it is possible to guarantee ... that the trees it produces are asymptotically arbitrarily close to the ones produced by a batch learner (i.e. a learner that uses all the examples to choose a test at each node)'. Such a learning algorithm is TDIDT, which was described in Chapters 4–6.

It is difficult to test empirically whether this aim is achieved with very large datasets as there is no possibility of being able to process such datasets using TDIDT or any similar 'batch learner'. However we can make at least a partial comparison of the results achieved by TDIDT and H-Tree by a simple trick: using a small dataset repeatedly to give the effect of using a large one.

21.8.1 The lens24 Dataset

We will start with an extremely small dataset: *lens24*, an ophthalmological dataset which was described in Chapter 5. It has a mere 24 records, with four attributes: *age*, *specRx*, *astig* and *tears* and three classes: 1, 2 and 3.

If all 24 records are input to H-Tree many times, in batches of 24 and in

the same order each time, so the total number of records input is say 2,400, 24,000 or 24,000,000, we can examine the trees produced and compare them with those produced by TDIDT. For any exact number of replications of the original data records TDIDT will give the same result as if they had only been processed once.

We will compare the algorithms by extracting rules from the trees generated, each rule corresponding to the path from the root node to a leaf node, working from left to right.

There are nine rules generated by the TDIDT algorithm:

1. IF $tears = 1$ THEN Class $= 3$
2. IF $tears = 2$ AND $astig = 1$ AND $age = 1$ THEN Class $= 2$
3. IF $tears = 2$ AND $astig = 1$ AND $age = 2$ THEN Class $= 2$
4. IF $tears = 2$ AND $astig = 1$ AND $age = 3$ AND $specRx = 1$ THEN Class $= 3$
5. IF $tears = 2$ AND $astig = 1$ AND $age = 3$ AND $specRx = 2$ THEN Class $= 2$
6. IF $tears = 2$ AND $astig = 2$ AND $specRx = 1$ THEN Class $= 1$
7. IF $tears = 2$ AND $astig = 2$ AND $specRx = 2$ AND $age = 1$ THEN Class $= 1$
8. IF $tears = 2$ AND $astig = 2$ AND $specRx = 2$ AND $age = 2$ THEN Class $= 3$
9. IF $tears = 2$ AND $astig = 2$ AND $specRx = 2$ AND $age = 3$ THEN Class $= 3$

The results below show the rules generated by H-Tree with G and $Prob$ set to 500 and 0.999 respectively for varying numbers of records (all multiples of 24).

2400 records

At this stage there are just three rules, listed below.

1. IF $tears = 1$ THEN Class $= 3$
2. IF $tears = 2$ AND $astig = 1$ THEN Class $= \{0, 187, 38\}$
3. IF $tears = 2$ AND $astig = 2$ THEN Class $= \{149, 0, 76\}$

The arrays shown for rules 2 and 3 give the class totals for the three classes (1, 2 and 3) in order.

Rules 2 and 3 look as if they may be 'compressed' versions of TDIDT rules 2–5 and 6–9 respectively. How will they develop as more records are processed?

4800 records

At this stage H-Tree has generated six rules as follow.

1. IF $tears = 1$ THEN Class $= 3$
2. IF $tears = 2$ AND $astig = 1$ AND $age = 1$ THEN Class $= 2$
3. IF $tears = 2$ AND $astig = 1$ AND $age = 2$ THEN Class $= 2$
4. IF $tears = 2$ AND $astig = 1$ AND $age = 3$ THEN Class $= \{0, 55, 54\}$
5. IF $tears = 2$ AND $astig = 2$ AND $specRx = 1$ THEN Class $= 1$
6. IF $tears = 2$ AND $astig = 2$ AND $specRx = 2$ THEN Class $= \{54, 0, 109\}$

Now rules 4 and 6 look as if they may be compressed versions of TDIDT rules 4–5 and 7–9 respectively.

7200 records

At this stage there are the same six rules as for 4800 records, except that the arrays for rules 4 and 6 are now {0, 155, 154} and {154, 0, 309}, respectively.

9600 records

Now H-Tree has generated nine rules, reproduced below. They are exactly the same as the rules produced by TDIDT.

1. IF *tears* = 1 THEN Class = 3
2. IF *tears* = 2 AND *astig* = 1 AND *age* = 1 THEN Class = 2
3. IF *tears* = 2 AND *astig* = 1 AND *age* = 2 THEN Class = 2
4. IF *tears* = 2 AND *astig* = 1 AND *age* = 3 AND *specRx* = 1 THEN Class = 3
5. IF *tears* = 2 AND *astig* = 1 AND *age* = 3 AND *specRx* = 2 THEN Class = 2
6. IF *tears* = 2 AND *astig* = 2 AND *specRx* = 1 THEN Class = 1
7. IF *tears* = 2 AND *astig* = 2 AND *specRx* = 2 AND *age* = 1 THEN Class = 1
8. IF *tears* = 2 AND *astig* = 2 AND *specRx* = 2 AND *age* = 2 THEN Class = 3
9. IF *tears* = 2 AND *astig* = 2 AND *specRx* = 2 AND *age* = 3 THEN Class = 3

Running H-Tree with large numbers of additional repetitions of the original data appears to produce no further changes to the tree.

21.8.2 The vote Dataset

The *vote* 'US congressional voting' dataset has 300 records, with 16 attributes and 2 classes. TDIDT generates 34 rules from this dataset.

Records	Rules
12,000	9
24,000	14
36,000	17
72,000	24
120,000	27
360,000	28
480,000	28
720,000	28

Figure 21.17 Number of Records Generated for *vote* Dataset

Figure 21.17 shows the number of rules generated by the H-Tree algorithm for different numbers of records, all multiple repetitions of the original 300 records.

In this case inspection shows that the H-Tree seems to be converging towards the 34 rules generated by TDIDT. Out of the H-Tree's 28 rules, 24 are the same as those generated by TDIDT, but even after 720,000 records have been processed (2,400 repetitions of the original 300), there are still four of H-Tree's rules with mixed classifications that have not (yet?) been expanded into rules generated by TDIDT. In each case there would seem to be an obvious way in which the rule could be expanded and it is entirely possible that the four H-Tree rules with mixed classifications might evolve into ten rules with single classifications as they are for TDIDT given yet more repetitions of the original 300 records.

It would seem that the tree produced by H-Tree is converging towards the one generated by TDIDT, albeit very slowly[8].

21.9 Chapter Summary

This chapter is concerned with the classification of *streaming data*, i.e. data which arrives (generally in large quantities) from some automatic process over a period of days, months, years or potentially forever. Generating a classification tree for streaming data requires a different approach from the TDIDT algorithm described earlier in this book. The algorithm given here, *H-Tree*, is a variant of the popular VFDT algorithm which generates a type of decision tree called a *Hoeffding Tree*. The algorithm is described and explained in detail with accompanying pseudocode for the benefit of readers who may be interested in developing their own implementations. An example is given to illustrate a way of comparing the rules generated by H-Tree with those from TDIDT.

21.10 Self-assessment Exercises for Chapter 21

1. Why can the TDIDT algorithm not be used directly with streaming data?
2. What benefit is gained by using a Hoeffding Bound when generating an H-Tree?
3. How does the availability of a potentially infinite amount of streaming data compensate for being unable to store all the data?

[8] For some practical applications, to have a tree with a smaller number of leaf nodes which predicts the same or almost the same classifications as the complete TDIDT decision tree might be considered preferable, but we will not pursue that issue here.

4. Assume that we are considering splitting on an attribute at node Z. The values of *classtotals*$[Z]$ and *currentAtts*$[Z]$ are the arrays $\{20, 30, 50\}$ and $\{att1, att3, att4, att7\}$ respectively. Also assume that the values of G and *Prob* are 100 and 0.999 respectively and that the Information Gain associated with splitting on each of the four attributes is as follows:

   IG(*att1*): 0.1614
   IG(*att3*): 1.3286
   IG(*att4*): 1.0213
   IG(*att7*): 0.8783

   Calculate the value of the Hoeffding Bound and determine whether or not we should split on an attribute at node Z and, if so, which one.

References

[1] Domingos, P., & Hulten, G. (2000). Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 71–80). New York: ACM.

[2] Hoeffding, W. (1963). Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, *58* (301), 13–30.