22
Classifying Streaming Data II: Time-Dependent Data

22.1 Stationary versus Time-dependent Data

In all the discussion of classification techniques in this book up to now, there has been an implicit assumption, which it will now be helpful to make explicit.

We are assuming throughout that there is some underlying process which leads to an instance (or record) with a given set of attributes having a particular classification rather than another one. The classification is not just random – there is a reason (unknown to us) why a share price goes up or down, why a customer does or does not buy breakfast cereal, why a US congressman votes one way or another, why an earthquake does or does not occur on a given day and so on. We can refer to the process that determines the classification in a given domain as the *underlying causal model*.

The world is very complex and in many domains it may well be that we will never fully know the underlying model. To express it fully may involve a very complex mathematical formalism – possibly one that has not yet been invented. If we knew it in full detail we would probably realise that we had not recorded all the attributes needed to make an accurate calculation of the correct classification in each case and also that we had recorded some attributes that were entirely irrelevant to the result.

This all sounds very complicated and it is. The task of the data miner is

M. Bramer, *Principles of Data Mining*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-7307-6_22

to produce a model which approximates to the true underlying one using a formalism of our choice: a decision tree, a set of rules or perhaps some other formalism not discussed in this book such as a Neural Net or a Support Vector Machine. The accuracy of our approximation will probably depend crucially on which of the various available formalisms we use but in most cases we have no way of knowing which one to choose, although experience may act as a guide.

We now come to the reason for making all this explicit. Up to now we have assumed that however complex the underlying model may be it is *fixed*. Whatever the process is that produced the original classification it will be the same tomorrow, next week and next year as it is today. If data is collected and then analysed and re-analysed, perhaps over a period of years, then the underlying model we try to approximate is inevitably fixed. We generally assume that if we collect more data over time it will simply be more samples generated by the same underlying model. We will call such data *stationary data*.

The situation with classifying streaming data where we are classifying records over a long (potentially infinite) period of time is significantly different. Although it is perfectly possible that the underlying model is fixed (and this is the assumption made in Chapter 21) it is also possible that the model may change from time to time. For example if we are predicting which customers in a supermarket will buy a particular food product the underlying model may vary considerably from mid-summer to mid-winter, possibly changing gradually as time goes by but also possibly changing rapidly if there is an effective advertising campaign that boosts sales or a contaminated food alarm that causes them serious damage. We will call data resulting from a model that changes across time *time-dependent data*. The underlying model is often referred to as the concept we need to model and the phenomenon of a changing model is known as *concept drift*.

Although the H-Tree algorithm described in Chapter 21 works well with streaming data, if the model is fixed, it has a major weakness when faced with streaming data that is time-dependent: once a tree has been created it can only be changed by further splitting on nodes. There is no way of 'unsplitting', i.e. converting an internal node back to a leaf node or changing a split on attribute X at a given node to a split on attribute Y.

The H-Tree algorithm produces trees that are generally stable: once a reasonable number of records have been processed a large number of additional records can be processed with little or no further change to the tree. With time-dependent data such stability is highly undesirable – a tree that predicts with a high level of accuracy today may predict steadily less accurately as the underlying concept drifts. We need some way of revisiting decisions previously made when building our tree. This may involve replacing

the subtree hanging from a node by another subtree that is more appropriate for the changed concept.

Four crucial features that distinguish algorithms for classifying streaming data from the TDIDT algorithm described in Chapters 4–6, which we will call a *batch model*, are:

- We cannot collect all the data before generating a classification tree as the volume is potentially infinite.
- We cannot store all the data and revisit it repeatedly as we can do with TDIDT, once again as the volume is potentially infinite.
- We cannot wait until we have a fixed classification tree before we use the tree to predict the classification for previously unseen data. We must be able to use it for prediction with a high level of accuracy at any time, except for possibly a fairly short start-up phase.
- The algorithm must be able to operate in real-time and thus the amount of processing needed as each record comes in must be quite small. This is particularly important if we want to allow for data that arrives in high volume day-by-day such as data recording supermarket transactions or withdrawals from bank ATMs.

The H-Tree algorithm (based on Domingos and Hulten's VFDT algorithm [1]) meets these four criteria. In this chapter we will develop a revised version of the algorithm that meets the same criteria and also deals with data that is time-dependent. It is based closely on an algorithm introduced by Hulten, Spencer and Domingos [2] called CVFDT, standing for Concept-adapting Very Fast Decision Tree learner. As always with influential algorithms there are many published variants that purport to be improvements. Our own version, which will be described in this chapter, is based closely on CVFDT but incorporates some detailed changes and simplifications. To avoid confusion we will call it the CDH-Tree algorithm, standing for 'Concept Drift Hoeffding Tree'.

We will start by reviewing the H-Tree algorithm and then change it piece by piece to become the final version of CDH-Tree. The next section summarises the key points of the H-Tree algorithm without explanation. It is assumed that there is a constant stream of records arriving and that each one is processed as it arrives and is then thrown away. **If you have not yet read Chapter 21 you are strongly encouraged to do so before going on.**

22.2 Summary of the H-Tree Algorithm

This section gives a brief summary of the H-Tree algorithm developed in Chapter 21 as a reminder of the main points.

As 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.

As the 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 G records have been sorted to a leaf node and some other conditions are met that a change to the tree by splitting at that node is considered (G stands for 'Grace Period').

When a leaf node is *split* on an attribute, a subtree is created below it with one branch for each possible value of the selected attribute. Once a node has been split on an attribute it cannot later be 'unsplit' or 'resplit' on a different attribute.

Figure 22.1 shows a partly developed H-Tree.

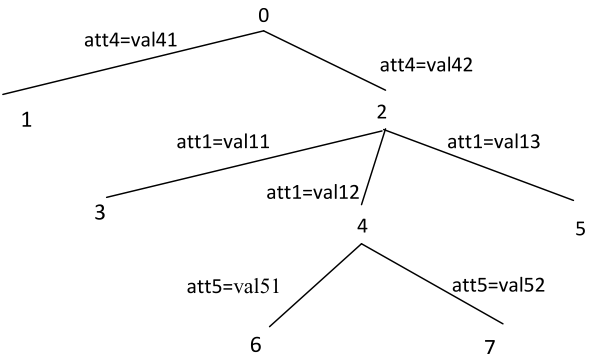

Figure 22.1 A Partly Developed H-Tree

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 nodes 1 and 2), *att1* at node 2 (creating nodes 3, 4 and 5) and *att5* at node 4 (creating nodes 6 and 7).

There are six arrays maintained at every node. They are listed below.

22.2.1 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.

If our data records have the values of seven attributes *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*}.

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 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*}.

22.2.2 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* and *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'.

22.2.3 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. As each new record is sorted to a leaf node L, the value of *hitcount*[L] is increased by 1. 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.

22.2.4 Array *classtotals*

This is another two-dimensional array. If there are three possible classifications named *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. When a new node is created its *classtotals* value is set to zero for all classes.

22.2.5 Array *acvCounts*

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 known as a frequency table, which records the number of occurrences of each possible combination of class value and the value of attribute A. 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.

22.2.6 Array *branch*

When leaf node N is split on attribute A a branch leading to a new node is created for each value of that attribute. For each value V of attribute A $branch[N][A][V]$ is set to *nextnode* and *nextnode* is increased by 1.

22.2.7 Pseudocode for the H-Tree Algorithm

The algorithm for processing records developed in Chapter 21 can be summarised by the following pseudocode fragments. These are not intended as a replacement for the explanation in Chapter 21 but are provided for the benefit of readers who may be interested in developing their own implementations of the algorithm. Other readers can safely ignore them.

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)

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'

We will consider all the pseudocode and other information in this section to be the 'initial draft' (version 1) of a specification for CDH-Tree and will progressively refine it in what follows.

22.3 From H-Tree to CDH-Tree: Overview

The two key ideas incorporated in the CDH-Tree algorithm that make it suitable for use with time-dependent data are:

- The values of arrays *hitcount* and *acvCounts* at each node are based on the most recent records to be processed, not all records ever received.
- There is a facility for changing the evolving classification tree by 'resplitting' at a node using a different attribute.

Before turning to these we begin by changing the counts recorded in arrays *hitcount* and *acvCounts* at internal nodes.

22.4 From H-Tree to CDH-Tree: Incrementing Counts

The first step in the transition from H-Tree to CDH-Tree is to increment the *acvCounts* and *hitcount* arrays not only for a leaf node L but also for each of the internal nodes (including the root) through which each record passes on its path from the root to node L.

Assuming there are no other changes, incrementing those values at the internal nodes will not alter the structure of the evolving tree. Only the values at leaf nodes will do that. Although the change will make no difference at this stage, it will turn out to be very useful as we go on adjusting the algorithm until it eventually becomes the final version of CDH-Tree. The key point is that at each internal node the *hitcount* and *acvCounts* arrays will now hold the values that they would have had if they had remained unsplit, thus opening up the possibility of being split on a different attribute at a later stage.

The first revision to the H-Tree algorithm in its transition to CDH-Tree is to replace pseudocode fragment 3 by:

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

1. Set N to the number of the root node
2. **Set Continue to 'yes'**
3. **While (Continue = 'yes')**
   a) **Update arrays at node N**
      - **Increase $hitcount[N]$ by 1**
      - **For each attribute Att in the current attributes array for node N**
        - **set Val to the value that the attribute has in record R**
        - **increase $acvCounts[N][Att][C][Val]$ by 1**

   b) **If $splitAtt[N]$ = 'none' set Continue to 'no'**
      else
      - 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. **Increase $classtotals[L][C]$ by 1**
6. 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)

Here and subsequently the parts altered are shown in **bold**. Note that array *classtotals* is only updated at leaf nodes. It is not used when making decisions about splitting or unsplitting but has an important role when using the classification tree for prediction, so its value needs to be increased only at the leaf node to which each record is sorted.

Pseudocode fragments 1, 2 and 4 remain unaltered.

22.5 The Sliding Window Method

Up to now we have built the tree using all the records that have so far arrived. From now on we will use only the most recent W records, say the most recent 10,000. We can think of this as looking at the records going past through a 'window' of fixed size W. We call this the *sliding window* method and call W the *window size*.

This change means that we now need to store the most recent W records[1]. When initially the first W records are processed they are stored in a table of W rows (or some equivalent form), working from row 1 down to row W. When the next record is read its values go into row 1 and the record that previously occupied that row (i.e. the oldest record) is discarded. When the next record is read, it goes into row 2 (which is now the oldest record), the previous occupant of that row is discarded and so on. After $2 * W$ records have been processed none of the original W records will remain in the window.

When a new record is added to the window the *acvCounts* and *hitcount* arrays are incremented for all the nodes it passes through on its path from the root node to the appropriate leaf (including the root node and leaf node) and the *classtotals* array is incremented for the leaf node itself. When an old record is removed from the window it is not merely discarded, it is *forgotten*. Forgetting a record means decreasing by one the values in the *acvCounts* and *hitcount* arrays for all the nodes it passed through on its path from the root node to the appropriate leaf node when it was first added to the window and decreasing by one the values in the *classtotals* array for the leaf node itself.

Since at present we are keeping to the principle that once we have split on an attribute at a given node that decision is never changed, it is still only the values of the *acvCounts* and *hitcount* arrays at the leaf nodes that affect the evolving tree. In [2] it is argued that if the data is stationary, adding records to or removing records from the sliding window should have little effect on the evolving tree.

Our aim in this chapter is to create an algorithm that will deal with data that is time-dependent and the value of introducing a sliding window will become apparent soon.

The revised version of the pseudocode for the main CDH-Tree algorithm now looks like this.

[1] We will assume that each record comprises a set of attribute values together with a classification.

CDH-Tree Algorithm
(version 2)

1. Set values of G, *Prob* **and** $\boldsymbol{W}$
2. Initialise root node (see pseudocode 1)
3. For each record R with classification C that arrives to be processed
   a) **If the number of records in the window** $<$ $\boldsymbol{W}$ **add** $\boldsymbol{R}$ **to the window**
      else
      i. **take a copy of the oldest record in the window** $\boldsymbol{R}_{old}$ **with classification** $\boldsymbol{C}_{old}$
      ii. **replace** $\boldsymbol{R}_{old}$ **by** $\boldsymbol{R}$
      iii. **'forget' record** $\boldsymbol{R}_{old}$ **with classification** $\boldsymbol{C}_{old}$ **(see pseudocode 5)**
   b) Process record R with classification C (see pseudocode 3)

Steps (a) and (b) of the algorithm are often described as the grow/forget stage.

Pseudocode fragment 5 is based on fragment 3 but of course it does not need to include any possibility of splitting on the leaf node. *Warning – this version contains an error.*

Pseudocode 5: Forget Record $\boldsymbol{R}_{old}$**with Classification** $\boldsymbol{C}_{old}$

1. Set N to the number of the root node
2. Set Continue to 'yes'
3. While (Continue = 'yes')
   a) Update arrays at node N
      i. decrease $hitcount[N]$ by 1
      ii. for each attribute Att in the current attributes array for node N
         - set Val to the value that the attribute has in record R
         - decrease $acvCounts[N][Att][C_{old}][Val]$ by 1
   b) If $splitAtt[N]$ = 'none' set Continue to 'no'
      else
      i. Set A to $splitAtt[N]$
      ii. Set V to value of attribute A in record R_{old}
      iii. Set N to $branch[N][A][V]$
4. Decrease $classtotals[N][C_{old}]$ by 1

This form of the algorithm is not quite correct. It ignores an important complication that can occur when we forget a record.

Suppose when we added a record to the window the tree looked like Figure 22.2 and the record was sorted to leaf node 4 following the path $0 \rightarrow 2 \rightarrow 4$.

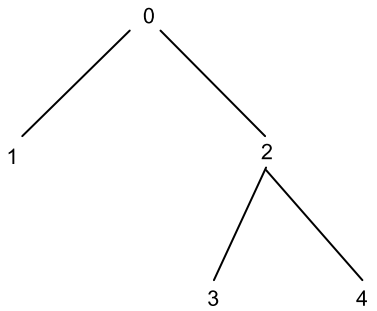

Figure 22.2 A Small Decision Tree

When we come to forget the record the tree may have evolved to look like this (Figure 22.3).

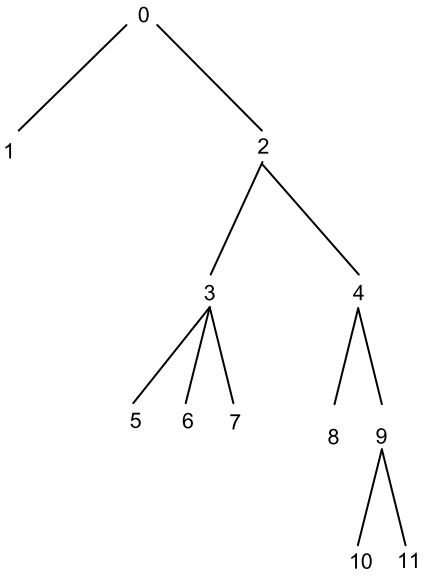

Figure 22.3 Decision Tree at a Later Stage

During the forgetting process (as so far described) the record might now be sorted to leaf node 10 following the path $0 \rightarrow 2 \rightarrow 4 \rightarrow 9 \rightarrow 10$.

It would clearly be wrong to decrement the various counts at nodes 9 and 10 as the record never reached there at the 'grow' stage and so the counts at those nodes were never increased.

To deal with this problem we note that node numbers are allocated sequentially as new nodes are created. The record that we are currently 'forgetting' was originally sorted to leaf node 4. Any further nodes added to any part of the tree after that happened must inevitably have a number that is greater than 4.

So we adopt the strategy that whenever a record is added to the window we store the number of the leaf node to which it is sorted in the window with it. In this case the number is 4. In the general case we will call the value *idMax*.

This revised version of pseudocode fragment 3 shows the change needed.

Pseudocode 3: Process Record R with Classification C (version 3)
As version 2 but with a new Step 6
6. Store the value of L in the window as the *idMax* value of the newest record R
and the previous Step 6 renumbered as Step 7

Now when we forget a record we first retrieve its corresponding *idMax* value and follow the record's path to its (possibly new) corresponding leaf node (decrementing the counts at each node as we do so) only as long as the number of each node is less than or equal to *idMax*.

It will prove convenient to place the retrieve part of this in the 'main' algorithm which becomes

Outline of CDH-Tree Algorithm (version 3)
As version 2 but with the following added as the second stage of the 'else' part of Step 3(a)

- **set $idMax_{old}$ to the *idMax* value of record R_{old}**

Pseudocode fragment 5 now looks as follows.

Pseudocode 5: Forget Record R_{old} with Classification C_{old} (version 2)

1. Set N to the number of the root node
2. **If $N \leq idMax_{old}$**
   a) Set Continue to 'yes'
   b) While (Continue = 'yes')
      i. Update arrays at node N
         - decrease $hitcount[N]$ by 1
         - for each attribute Att in the current attributes array for node N
           - set Val to the value that the attribute has in record R
           - decrease $acvCounts[N][Att][C_{old}][Val]$ by 1
      ii. **Set N_{last} to N**
      iii. If $splitAtt[N]$ = 'none' set Continue to 'no'
      else
         - Set A to $splitAtt[N]$
         - Set V to value of attribute A in record R_{old}
         - Set N to $branch[N][A][V]$
         - **If $N > idMax_{old}$ set Continue to 'no'**
   c) **Decrease $classtotals[N_{last}][C_{old}]$ by 1**

22.6 Resplitting at Nodes

We now come to the most important change in the transition from the H-Tree algorithm to CDH-Tree, which the previous sections have been building up to.

The fundamental problem with the method described when the data is time-dependent is that once a split is made at a node it can never be reversed or changed. We now need to address this issue.

The general idea is that periodically all the internal nodes of the tree are checked to determine for each one whether the attribute on which it was split previously would still be chosen if the decision were being made now, using the counts currently recorded, which relate only to the records currently in the window. If it would not still be chosen and a condition involving the Hoeffding Bound is met by a different attribute, this is taken as a possible indication of concept drift and the node (which will necessarily have a subtree hanging from it) is treated as 'suspect'. We will defer until later sections the

issue of what to do with suspect nodes.

22.7 Identifying Suspect Nodes

We will set another parameter D, representing the number of records between checks for concept drift. After each D records have been processed we conduct a review of each of the internal nodes in the tree in turn, to check whether the attribute that was selected to split on there would still be selected if the decision were being made now, based on the records in the current sliding window. Any node which fails this test and at which a different attribute passes a test involving the Hoeffding Bound is treated as 'suspect'.

This requires some minor changes to the main algorithm. We will use variable *recordnum* to store the number of each record to be processed, counting from zero.

Outline of CDH-Tree Algorithm (version 4)

1. Set values of G, $Prob$ and W **and** D
   Set *recordnum* to zero
2. Initialise root node (see pseudocode 1)
3. For each record R with classification C that arrives to be processed
   a) if the number of records in the window $< W$ add R to the window else
      i. take a copy of the oldest record in the window: R_{old} with classification C_{old}
      ii. set $idMax_{old}$ to the $idMax$ value of record R_{old}
      iii. replace R_{old} by R
      iv. 'forget' record R_{old} with classification C_{old} (see pseudocode 5)
   b) process record R with classification C (see pseudocode 3)
   c) **increase *recordnum* by 1**
   d) **if *recordnum* is a multiple of D review node 'root' and all its descendants that are internal nodes (see pseudocode 6)**

Pseudocode fragment 6 will be developed later in this section.

It is straightforward to identify all the internal nodes in the tree systematically by starting at the root node and using the contents of the

splitAtt and *branch* arrays. Figure 22.4 illustrates the method[2].

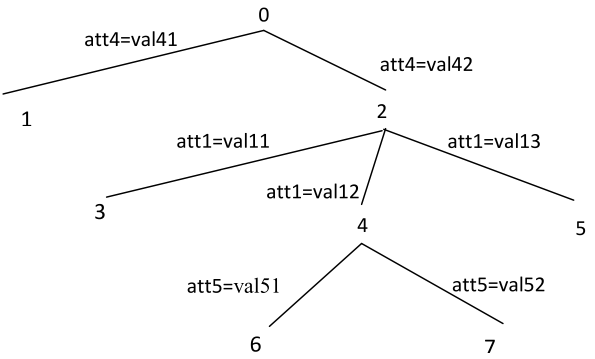

Figure 22.4 A Partly Developed CDH-Tree

The splitting attribute at node zero (the root) is *att4*, which has two values: *val41* and *val42*. These lead to nodes 1 and 2 using the *branch* array: *branch*[0][*att4*][*val41*] is 1 and *branch*[0][*att4*][*val42*] is 2. We look at each of these nodes in turn. Node 1 is a leaf node (we know this from examining *splitAtt*[1]), so we take no action at that node. Node 2 is an internal node. It is split on attribute *att1*, which has three values: *val11*, *val12* and *val13*. The first and third of these lead to leaf nodes 3 and 5 respectively, but the second branch leads to node 4 which is an internal node, and so the process continues, until every path eventually leads to a leaf node and ends.

At each internal node we calculate the Information Gain corresponding to each attribute available for splitting. If the one that was chosen for splitting, attribute A, does not have the largest value, we find the two attributes with the largest Information Gain values and calculate the Hoeffding Bound. If the difference between the values of Information Gain for the best attribute X and the second best attribute Y is greater than the value of the Hoeffding Bound, the node is considered suspect and attribute X is considered to be the alternative splitting attribute for that node.

Whether the node is deemed 'suspect' or not it remains split on attribute A at present. We go on to examine its direct descendants using the contents of array *branch*.

All this gives two new fragments of pseudocode: 6 and 7.

[2]Figure 22.4 is identical to Figure 22.1. It is repeated for ease of reference.

Pseudocode 6: Review Node *N* and all its Descendants that are Internal Nodes

1. If N is a leaf node stop
   else
   a) Examine internal node N (see pseudocode 7)
   b) If node N is suspect with alternative attribute X *To be discussed in Section 22.8*
   c) Set A to $splitAtt[N]$
   d) For each value V of attribute A
      i. set *N1* to $branch[N][A][V]$
      ii. review node *N1* and all its descendants that are internal nodes

Pseudocode 7: Examine Internal Node *N*

1. Let A be the attribute currently split on at node N, i.e. $splitAtt[N]$
2. Find the value of Information Gain for each attribute in node N's current attribute array
3. If A has the largest value of Information Gain, return 'not suspect'
   else
   a) Find the two attributes with the largest values of Information Gain. Call them X and Y respectively
   b) Calculate the value of the Hoeffding Bound ϵ for node N
   c) If $IG(X) - IG(Y) > \epsilon$ return X as alternative attribute for splitting at node N

22.8 Creating Alternate Nodes

When a node N is identified as 'suspect', one possible action would be to convert it back to a leaf, throwing away its descendant subtree, and rely on the normal process of growing/forgetting as new records arrive to evolve a new subtree structure as time passes. The problem with doing this is that while it is going on, there is likely to be a substantial drop in the predictive accuracy of the tree for unseen records, especially if the node concerned is close to the root.

Instead we create an 'alternate node' for any suspect node N with initially a one-level subtree hanging from it formed by splitting on the 'alternative splitting attribute' X. We will assume that this alternate node is numbered *N1*. Node *N1* is given the same *currentAtts*, *hitcount*, *classtotals* and *acvCounts* arrays as node N but its splitting attribute (the value of array element *splitAtt*[*N1*]) is different. It has the value X.

Next we create a branch from node *N1* to a new node for each value of the new splitting attribute.

Node *N1* and its substructure are not part of our classification tree but we can consider nodes N and *N1* being linked by a dotted line as in Figure 22.5, where N and *N1* are 2 and 10 respectively. Node 10 is an *alternate node* for node 2 in the main tree.

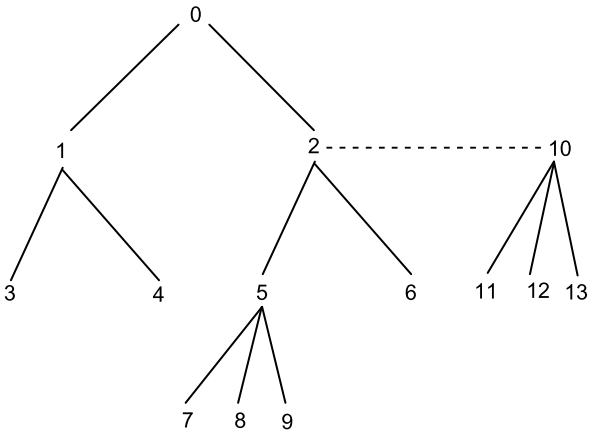

Figure 22.5 A CDH-Tree with an Alternate Node

As time goes by more than one alternate node and its substructure can be associated with a suspect node. As the grow/forget process continues the substructure below each alternate node may also evolve.

At some later time (see Section 22.10) a decision will be made whether or not to replace each of the suspect internal nodes by one of its alternate nodes, and if there is more than one of them which one. At that time Figure 22.5 may have evolved to look like this (Figure 22.6).

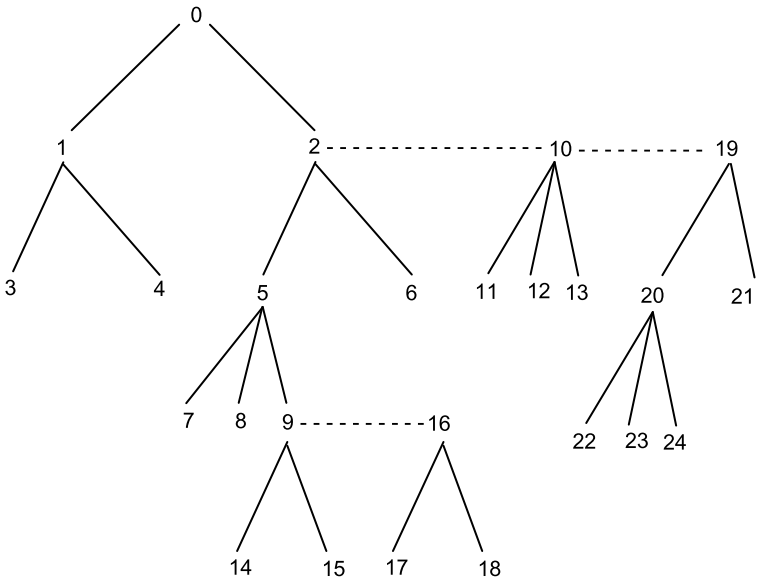

Figure 22.6 CDH-Tree at a Later Stage

To record the link between nodes in the main tree and their alternate nodes in array form we use a seventh array: *altTreeList*. This is a two-dimensional array. For Figure 22.6 *altTreeList*[2] is the array $\{10, 19\}$. The *altTreeList* array is only applicable to nodes in the main tree, not alternate nodes or any other nodes in the substructure hanging from them[3].

We can now replace the words in italics in pseudocode fragment 6:

[3]This is a restriction imposed by the CDH-Tree algorithm. It would be possible to allow nodes in an alternate tree to have their own alternate nodes but at the risk of creating and needing to maintain an increasingly unwieldy structure, most of which will never form part of the main tree. It is only the current main tree that is used for prediction.

Pseudocode 6: Review Node *N* and all its Descendants that are Internal Nodes
(version 2)

1. If N is a leaf node stop
   else
   a) Examine internal node N (see pseudocode 7)
   b) If node N is suspect with alternative attribute X
      i. **set *newnode* to *nextnode***
      ii. **increase *nextnode* by 1**
      iii. **add *newnode* to the array at *altTreeList*[*N*]**
      iv. **copy arrays *hitcount*, *classtotals*, *acvCounts* and *currentAtts* from node *N* to node *newnode***
      v. **split at node *newnode* using attribute *X* (see pseudocode 2)**
   c) Set A to $splitAtt[N]$
   d) For each value V of attribute A
      i. set *N1* to $branch[N][A][V]$
      ii. review node *N1* and all its descendants that are internal nodes

Replacing node 2 in Figure 22.6 by its (second) alternate node 19 would give this new tree structure (Figure 22.7).

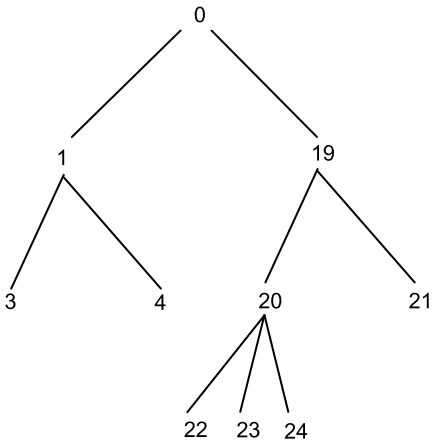

Figure 22.7 CDH-Tree after Node 2 Replaced by Alternate Node

Once a decision is made to replace, say node 2 by its alternate node 19, implementing the change is straightforward. Node 2 is a direct descendant of node 0. Let us say that the branch in question is for attribute *att7* with value *val72*. Then the element of the *branch* array that links nodes 0 and 2 is *branch*[0][*att7*][*val72*]. All we have to do is to change the value of this array element from 2 to 19.

The aim of this cautious approach to 'resplitting' a node on a different attribute from before is to ensure a smooth changeover to the new tree, which will enable it to maintain a high level of predictive accuracy throughout.

Using the 'alternate node' approach would be pointless for stationary data (as the data used with the algorithm developed in Chapter 21 is assumed to be), as it is unlikely that any would ever be created and, if they were, it is unlikely that they would ever replace the original nodes. In the case of time-dependent data it is hoped that using alternate nodes will give a smooth and appropriately cautious way of re-splitting the decision tree at one or more nodes as concept drift occurs.

22.9 Growing/Forgetting an Alternate Node and its Descendants

After any alternate nodes have been created the usual process of growing and forgetting as new records are read continues with the crucial difference that the increases/decreases in the counts are also applied to the alternate nodes and the nodes in the subtrees hanging from them. (In doing this each alternate node is treated as if it were the root of a separate tree.) Leaf nodes in these alternate trees can also be split whenever the Hoeffding Bound requirement is met. To do all this requires a further change to pseudocode fragment 3.

Pseudocode 3: Process Record R with Classification C (version 4)

1. Set N to the number of the root node
2. Set Continue to 'yes'
3. While (Continue = 'yes')
   a) Update arrays at node N
      i. Increase $hitcount[N]$ by 1
      ii. For each attribute Att in the current attributes array for node N
         - set Val to the value that the attribute has in record R
         - increase $acvCounts[N][Att][C][Val]$ by 1
   b) **If $altTreeList[N]$ is not empty, for each node number $nextalt$ in the array, treating node $nextalt$ as the root**
      - **process record R with classification C (*)**
   c) If $splitAtt[N]$ = 'none' set Continue to 'no'
      else
      i. Set A to $splitAtt[N]$
      ii. Set V to value of attribute A in record R
      iii. Set N to $branch[N][A][V]$
4. Set $L = N$
5. Increase $classtotals[L][C]$ by 1
6. **If $L >$ the $idMax$ value of the newest record R in the window, replace it by L**
7. 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
   a) determine whether to split on an attribute at node L (see pseudocode 4)
   b) if answer is 'yes', with splitting attribute A, split at node L, using attribute A (see pseudocode 2)

The notation (*) indicates a recursive call to the same pseudocode fragment.

Pseudocode 5 also needs to be changed to enable forgetting to take place in alternate trees (if applicable) not only the main tree.

Pseudocode 5: Forget Record R_{old} with Classification C_{old} (version 3)

1. Set N to the number of the root node
2. If $N \leq idMax_{old}$
   a) Set Continue to 'yes'
   b) While (Continue = 'yes')
      i. Update arrays at node N
         - decrease $hitcount[N]$ by 1
         - for each attribute Att in the current attributes array for node N
           - set Val to the value that the attribute has in record R
           - decrease $acvCounts[N][Att][C_{old}][Val]$ by 1
      ii. **If $altTreeList[N]$ is not empty, for each node number *nextalt* in the array, treating node *nextalt* as the root, forget record R_{old} with classification C_{old} (*)**
      iii. Set N_{last} to N
      iv. If $splitAtt[N]$ = 'none' set Continue to 'no'
      else
         - Set A to $splitAtt[N]$
         - Set V to value of attribute A in record R_{old}
         - Set N to $branch[N][A][V]$
         - If $N > idMax_{old}$ set Continue to 'no'
   c) Decrease $classtotals[N_{last}][C_{old}]$ by 1

22.10 Replacing an Internal Node by One of its Alternate Nodes

All that remains to be added to the algorithm is a mechanism for determining whether and when to replace an internal node by one of its alternate nodes and, if it has more than one alternate node, how to decide which one to choose.

In order to do this, after each T records (say 10,000) are processed the system enters into a testing phase. The next M records (say 500) are used for testing the predictive accuracy of the tree. They do not affect the contents of the window or any of the *acvCounts*, *classtotals* and *hitcount* arrays and do not cause any splitting at leaf nodes to occur.

Instead each record processed is sorted to the corresponding leaf node in the main tree and also in any alternate tree linked to any of the internal nodes through which it passes.

To illustrate this Figure 22.8 shows a possible state of the tree at the start of a testing phase. Node 4 has alternate nodes 8 and 14; node 19 has alternate node 23 and node 7 has alternate node 26.

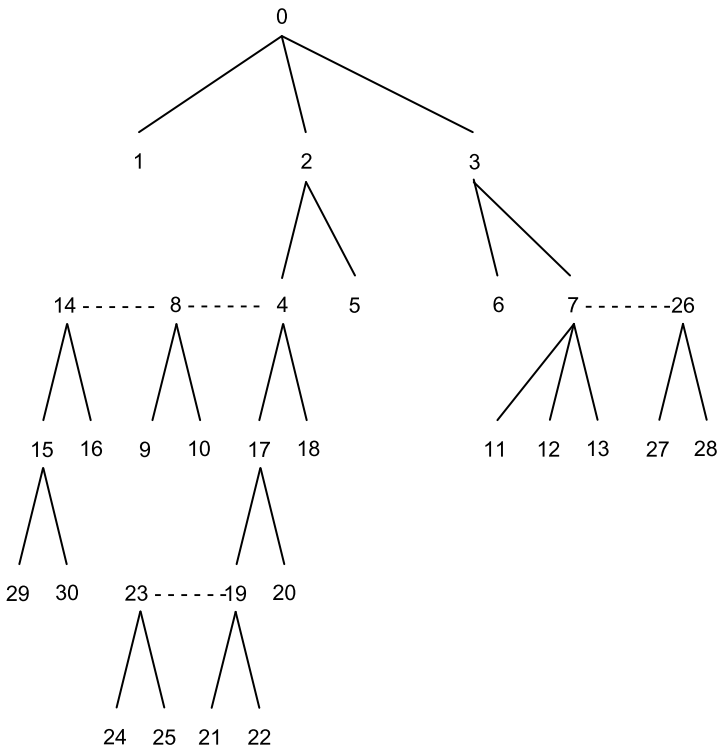

Figure 22.8 Tree Structure at Start of Testing Phase (Including Alternate Trees)

Suppose that a test record is sorted to leaf node 22 following the path 0 → 2 → 4 → 17 → 19 → 22. As the record 'passes through' internal node 4 (on its way to the leaf node) it is automatically copied and passed to the alternate trees hanging from the alternate nodes for node 4, i.e. 8 and 14. This may lead to it also being sorted to leaf nodes 9 and 29. As the record goes on and passes through node 19 it is also automatically copied and passed to that node's alternate, node 23, and from there may be sorted to leaf node 24. Thus a single record is sorted to four different leaf nodes.

We can now make a prediction of the classification at each of the four nodes and compare it with the true classification for the record. We do this using the contents of the *classtotals* array for each of the nodes on the path from the root to the leaf node (including the root and the leaf itself).

To predict the classification at node 22 in the main tree we use the contents of the *classtotals* arrays at nodes 0, 2, 4, 17, 19 and 22. If there are three possible classifications c1, c2 and c3, the array *classtotals*[2], say, will contain three values such as $\{50, 23, 42\}$. These correspond to the number of records with each classification that were sorted to that node when it was still a leaf node, i.e. before splitting on an attribute occurred there. We add together the contents of the *classtotals* arrays at nodes 0, 2, 4, 17, 19 and 22, element by element, into a combined array *testTotals* which might then contain values such as $\{312, 246, 385\}$. The class with the largest value in the *testTotals* array in our example is c3 and so this is taken to be the class predicted for leaf node 22.

We compare the prediction with the true classification, which in this case we will assume is c3, so in this case the predicted and the true classification are the same.

The same method is used for prediction if a record is sorted to a leaf node in an alternate tree, e.g. node 29. In this case we would combine the contents of the *classtotals* arrays at nodes 0, 2, 14, 15 and 29.

As more records are processed during the testing phase we accumulate a count for each leaf node (in both the main tree and the alternate trees) of the total number of records sorted to it and the number of those records for which the classification is correctly predicted. We store these values for each node in the two-dimensional array *testcounts*. At node 22, say, array elements *testcounts*[22][0] and *testcounts*[22][1] give the number of records so far sorted to that node during the testing stage and the number correctly classified respectively.

At the end of the testing phase we will have a *testcounts* array with contents such as the following for each of the 19 leaf nodes in our tree (Figure 22.9).

We can now fill in the table with *testcounts* array values for each of the internal nodes too, calculated by adding together the values of each of its immediate successors, working upwards from the leaf nodes.

Ignoring at present the possible existence of alternate nodes, the procedure is straightforward. Taking the part of the tree shown in Figure 22.10, the *testcounts* array for node 19 is the sum of those for nodes 21 and 22, i.e. $\{25, 18\}$ and $\{5, 3\}$ making a total of $\{30, 21\}$. Adding the *testcounts* values for leaf node 20 to this gives the array for internal node 17 as $\{40, 29\}$. Proceeding in this way the values at the leaf nodes are propagated up the tree right up to the root.

Leaf Node N	Number of Records $testcounts[N][0]$	Number of Correct Classifications $testcounts[N][1]$
1	25	16
5	6	3
6	5	3
9	20	14
10	24	16
11	10	7
12	5	3
13	5	3
16	14	12
18	4	3
20	10	8
21	25	18
22	5	3
24	15	13
25	15	6
27	10	6
28	10	4
29	20	13
30	10	10

Figure 22.9 Contents of *testcounts* Array at End of Testing Phase

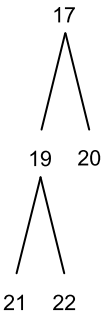

Figure 22.10 Extract from Figure 22.8 – Tree Structure at Start of Testing Phase

However there is an important complication that arises when a node such as 19 is reached which has an alternate node, in this case 23. We will reinstate this and the alternate tree hanging from it to give the extract shown in Figure 22.11.

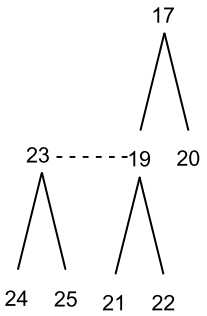

Figure 22.11 Figure 22.10 Augmented with Alternate Node 23 and its Subtree

Before passing the values from node 19 up to its parent node 17, we first consider the alternate tree rooted at node 23. The *testcounts* array at node 23 is the combination of those at its successor nodes 24 and 25 (both leaf nodes) giving $\{30, 19\}$.

We now compare the scores at nodes 19 and 23: $\{30, 21\}$ versus $\{30, 19\}$. The first elements are identical as they must always be, since the same records pass through each of them in the testing phase. However the descendants of node 19 correctly predicted the classifications of 21 records, compared with only 19 for the descendants of alternate node 23. Node 19 is not replaced by its alternate node and remains unchanged in the tree, as also does alternate node 23.

When we get to node 4 the performance at that node needs to be compared with that of the alternate trees rooted at its two alternate nodes, 8 and 14. The *testcounts* values are as follows.

Node 4: $\{44, 32\}$

Node 8: $\{44, 30\}$

Node 14: $\{44, 35\}$

This time the alternate tree rooted at node 14 has the best performance, so node 4 is replaced by alternate node 14.

The tree now looks like this (Figure 22.12)[4].

[4]Although nodes 14, 15, 16, 29 and 30 were previously parts of an alternate tree they are now in the main tree and so potentially can have alternate tree structures attached to them.

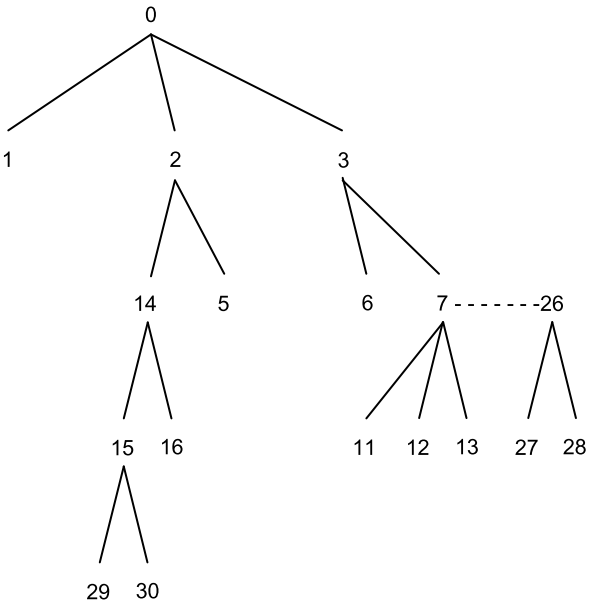

Figure 22.12 Tree Structure Following Replacement of Node 4 by Alternate Node 14

Note that the value of the *testcounts* array for node 2 is now the combination of those for nodes 14 and 5, i.e. $\{50, 38\}$.

Finally we come to the decision whether to replace node 7 by its alternate node 26. The *testcounts* arrays are $\{20, 13\}$ for node 7 and $\{20, 10\}$ for node 26. Node 7 has the better performance so both nodes remain unchanged in the tree.

This completes the investigation of the tree at the end of the testing phase. Part of the structure has changed but there is still one suspect node, 7, with an alternate, 26. The possibility of replacing the suspect node will be considered again at the end of the next testing phase, by which time new alternate nodes and their subtrees may have been created[5].

The above method requires a final change to the pseudocode for the main CDH-Tree algorithm plus two additional pseudocode fragments 8 and 9.

[5]Nodes 4 and 8 in Figure 22.8 and the subtrees hanging from them are not part of the revised structure and are no longer accessible. It may be possible for a practical implementation to reuse the memory they occupy but we will not pursue this here.

Outline of CDH-Tree Algorithm
(version 5)

1. Set values of G, *Prob* and W, D, ***T*** **and** ***M***
   Set *testmode* to 'no'
   Set *recordnum* to zero
2. Initialise root node (see pseudocode 1)
3. For each record R with classification C that arrives to be processed
   a) **if *testmode* = 'yes' sort record *R* and its classification *C* to leaf node(s) and predict classification(s) (see pseudocode 8) else**
      i. if the number of records in the window $< W$ add R to the window else
         - take a copy of the oldest record in the window: R_{old} with classification C_{old}
         - set $idMax_{old}$ to the *idMax* value of record R_{old}
         - replace R_{old} by R
         - 'forget' record R_{old} with classification C_{old} (see pseudocode 5)
      ii. process record R with classification C (see pseudocode 3)
   b) increase *recordnum* by 1
   c) **if *recordnum* is a multiple of *T* set *testmode* to 'yes' else if *testmode* = 'yes' and *recordnum* is a multiple of *T* plus *M***
      i. **set *testmode* to 'no'**
      ii. **check internal nodes for possible replacement, starting with root node, returning array *tcounts* (see pseudocode 9)**
   d) if *recordnum* is a multiple of D review node 'root' and all its descendants that are internal nodes (see pseudocode 6)

(The array *tcounts* returned by step 3(c)(ii) is not used here, but is important to the recursive definition of pseudocode 9 given below.)

Pseudocode 8: Sort Record *R* and its Classification *C* to Leaf Node(s) and Predict Classification(s)

1. Set N to the number of the root node
2. Set elements of *testTotals* array to zero (one for each class)
3. Set Continue to 'yes'
4. While (Continue = 'yes')
   a) Increase elements of *testTotals* array by corresponding elements of *classtotals*$[N]$
   b) If *splitAtt*$[N]$!= 'none'
      i. If *altTreeList*$[N]$ is not empty
      for each node number *N1* in the array, treating node *N1* as the root
      - sort record R and its classification C to leaf node(s) and predict classification(s) (*)

      ii. Set A to *splitAtt*$[N]$
      iii. Set V to value of attribute A in record R
      iv. Set N to *branch*$[N][A][V]$
5. Set $L = N$
6. Set predicted class to the class with the largest value in the *testTotals* array
7. Increase first element of array *testcounts*$[N]$ by 1
8. If (predicted class = C) Increase second element of array *testcounts*$[N]$ by 1

Pseudocode 9: Check Internal Nodes for Possible Replacement, Starting with Node *N*, Returning Array *tcounts*

1. If *splitAtt*[*N*] = 'none' Set array *tcounts* (two elements) to be the same as *testcounts*[*N*]
   else
   a) Set array *tcounts* to zero (two elements)
   b) For each immediate descendant node *N1* of node *N*
      i. Check internal nodes for possible replacement, starting with node *N1*, returning array *tcounts1* (*)
      ii. Increase elements of *tcounts* by the corresponding values of array *tcounts1*
   c) If array *altTreeList*[*N*] is not empty
      i. for each alternate node number *alt* in the array
         - check internal nodes for possible replacement, starting with node *alt*, returning array *tcountalt* (*)
      ii. Find alternate node alt_{best} with largest value of second element of *tcountalt*
      iii. If that value is larger than the second element of *tcounts*
         - set array *tcounts* to array *tcountalt*
         - replace node *N* in the main tree by alternate node alt_{best}
2. Return array *tcounts*

22.11 Experiment: Tracking Concept Drift

The CDH-Tree algorithm is a complex one, at least compared with other classification algorithms described elsewhere in this book. To illustrate how it works with time-dependent data, i.e. in the presence of concept drift, it is probably better to use synthetic data with concept drift introduced in a controlled way rather than real-world data, where it would be very hard to develop any feel for what the classification tree ought to be like at each stage.

As in Chapter 21 we will construct an experiment with an extremely small (but very useful) 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*. Attribute *age* has three values: 1, 2 and 3. The other three attributes have two values: 1 and 2. There are three classes: 1, 2 and 3.

If all 24 records are input to CDH-Tree (in the same order) a large number of times, 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 the tree-generation algorithm TDIDT described in Chapters 4–6. For any exact multiple of the original data records TDIDT will give the same result as if it had only been processed once. In Section 21.8.1 it was shown that at some point between processing 7200 records and processing 9600 records the H-Tree constructed from this data had already produced the same tree as TDIDT and hence the same rules.

After 9600 records (i.e. 400 repetitions of the *lens24* data records in the same order each time) had been processed the decision tree looked like this (Figure 22.13)[6].

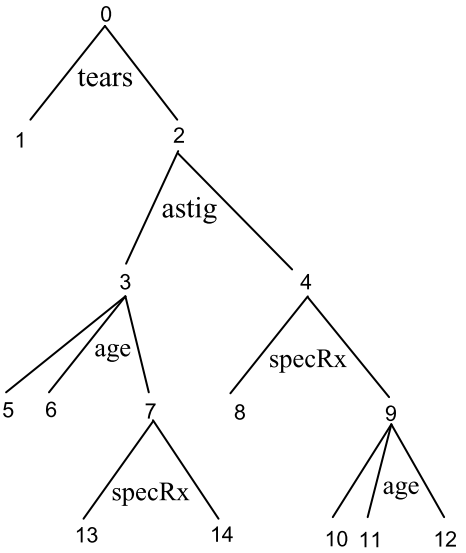

Figure 22.13 Tree Generated by TDIDT and H-Tree for 9,600 Records of *lens24* Data

We can extract rules from such a tree branch by branch, each rule corresponding to the path from the root node to a leaf node, working from left to right.

[6]We will adopt the convention that the branches at each internal node correspond to attribute values 1 and 2 (or 1, 2 and 3 in the case of *age*) in that order, working from left to right. So, for example, node 6 corresponds to a rule with left-hand side IF *tears* = 2 AND *astig* = 1 AND *age* = 2. (The corresponding classifications on the right-hand side are not shown.)

In this case there are nine rules generated by both TDIDT and H-Tree (Figure 22.14). All the leaf nodes have a single classification.

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 $= 3$

Figure 22.14 Rules Extracted from Decision Tree for *lens24* Data (9,600 Records)

Using the same data with the CDH-Tree algorithm, with W set to 9600, will produce exactly the same result[7]. We now need to find a way to introduce concept drift into the stream of data coming into CDH-Tree.

22.11.1 *lens24* Data: Alternative Mode

We start by introducing an alternative way of interpreting the *lens24* data records. Each record comprises four attribute values plus a classification. Normally the first, second, third and fourth attribute values are assigned to attributes *age*, *specRx*, *astig* and *tears* respectively. We will refer to this as the 'standard mode' of the data. If instead the four attribute values are assigned to attributes *age*, *tears*, *specRx* and *astig* in that order we will say the data is in 'alternative mode'[8].

If we run TDIDT, H-Tree or CDH-Tree (with W = 9600) on the *lens24* data in alternative mode for 9,600 records we will obtain exactly the same tree and corresponding nine rules as before except that every occurrence of *specRx*, *astig* and *tears* will be replaced by *tears*, *specRx* and *astig*, respectively.

[7] Up to the point where the sliding window is full, and provided D is greater than W, CDH-Tree is effectively the same algorithm as H-Tree.

[8] We have left attribute *age* unchanged to avoid irrelevant complications. It has three attribute values whereas the other attributes all have only two.

The decision tree will now look like this (Figure 22.15).

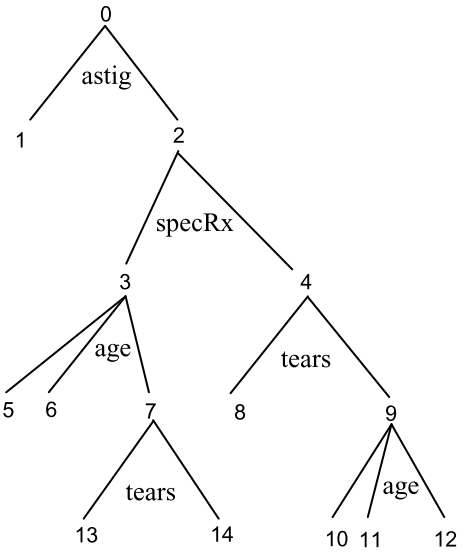

Figure 22.15 Tree Generated by TDIDT and H-Tree for 9,600 Records of *lens24* Alternative Mode

The rules extracted from the tree are now these (Figure 22.16).

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

Figure 22.16 Rules Extracted from Decision Tree for *lens24* Data – Alternative Mode

22.11.2 Introducing Concept Drift

We are now in a position to introduce concept drift into our stream of data. We will interpret the data as standard mode for the first 19,200 records, then switch to alternative mode for the next 19,200 records and so on indefinitely, alternating between the two modes. Figure 22.17 shows the mode applicable to the part of the infinite stream of records that we will use for our experiment.

Records	*lens24* Mode
0–19,199	Standard
19,200–38,399	Alternative
38,400–57,599	Standard
57,600–76,799	Alternative

Figure 22.17 Modes for Concept Drift Experiment

22.11.3 An Experiment with Alternating *lens24* Data

The following is a description of the behaviour of our implementation of the CDH-Tree algorithm when applied to the alternating *lens24* data, with the following variable settings:

Prob = 0.999
G = 500
W = 9600
D = 14000
T = 18000
M = 1200

Every D (i.e. 14,000) records the system checks for possible concept drift by checking that each internal node of the tree is split in the way that it would be split if the decision were made with the current contents of the sliding window. If not, it considers creating alternate nodes for some of the nodes in the main tree. Any internal node for which an alternate node is created can be considered 'suspect'. Initially each such alternate node has a one-level subtree hanging from it, split on the now preferred attribute.

Every T (i.e. 18,000) records the system goes into a testing phase. It uses the next M (i.e. 1,200) records not to develop the tree but to test the performance of each of the suspect internal nodes against the performance of each of its alternate nodes. The performance of a node is measured by the number of the M records that are correctly classified by the subtree hanging from it. At the end of the testing phase any internal node that is outperformed by one or more of its alternates is replaced by the best one.

The results shown below are a series of 'snapshots' of the state of the tree as more and more records are processed.

9600 Records

The data is in standard mode.

The sliding window is now complete for the first time (W = 9600). The tree and corresponding rules are those shown in Figures 22.13 and 22.14, respectively.

14,000 Records

The first check for concept drift is made (D = 14000). No nodes are found to be suspect. This is unsurprising as the standard data mode is still being used.

18,000 Records

The first testing phase begins (T = 18000). There are no alternate nodes, so this is certain to have no effect.

19,200 Records

The first testing phase ends (T = 18000, M = 1200). With no alternate nodes, no changes to the tree are possible. The tree predicts perfectly the classification of all 1,200 records.

Next the alternative data mode begins.

28,000 Records

The second check for concept drift is made (D = 14000). The split on attribute *tears* at node 0 is found no longer to be the best choice. An alternate node 15 is created with a one-level subtree split on attribute *astig*. No other suspect nodes are found.

The tree now looks like this (Figure 22.18). It is in transition from Figure 22.13 (standard mode of *lens24* data) towards Figure 22.15 (alternative mode of the data).

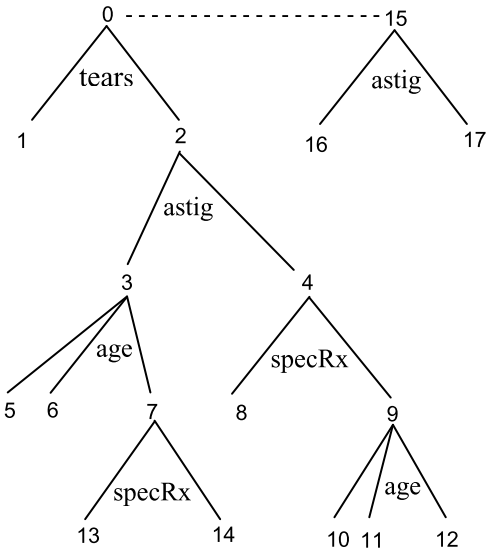

Figure 22.18 Tree After 28,000 Records Showing Alternate Node

The corresponding rules are now (Figure 22.19):

1. IF $tears = 14$ THEN Class $= \{1100, 734, 2966\}$
2. IF $tears = 2$ AND $astig = 1$ AND $age = 1$ THEN Class $= \{0, 66, 734\}$
3. IF $tears = 2$ AND $astig = 1$ AND $age = 2$ THEN Class $= \{0, 66, 734\}$
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 $= \{0, 34, 366\}$
6. IF $tears = 2$ AND $astig = 2$ AND $specRx = 1$ THEN Class $= \{100, 1100, 0\}$
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 $= 3$

Figure 22.19 Rules After 28,000 Records

The arrays shown for several of the rules give the classcounts for the three classes (1, 2 and 3) in order.

The concept drift between one mode of the *lens24* data and another is clearly having an effect on the predictions that would be made about the classification of records sorted to the different leaf nodes.

36,000 Records

The tree now looks like this (Figure 22.20):

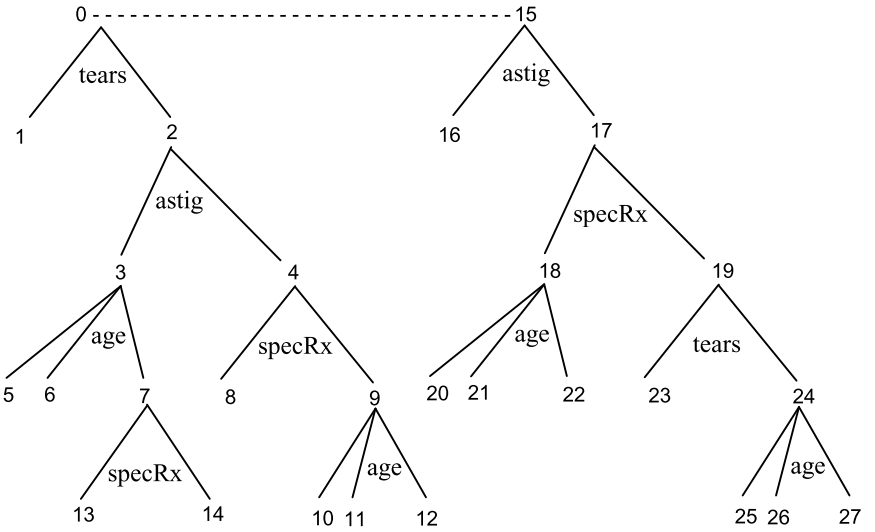

Figure 22.20 Tree after 36,000 Records

The alternate tree hanging from alternate node 15 is the same as Figure 22.15 (with the node numbers from 15 onwards rather than 0 onwards but in the same order) except that node 22 has not yet been split on attribute *tears*.

The corresponding rules are shown in Figure 22.21.

1. IF $tears = 1$ THEN Class $= \{1200, 800, 2800\}$
2. IF $tears = 2$ AND $astig = 1$ AND $age = 1$ THEN Class $= 3$
3. IF $tears = 2$ AND $astig = 1$ AND $age = 2$ THEN Class $= 3$
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 $= 3$
6. IF $tears = 2$ AND $astig = 2$ AND $specRx = 1$ THEN $Class = 2$
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 $= 3$

Figure 22.21 Rules Corresponding to Figure 22.20

The second testing phase now begins (T = 18000).

37,200 Records

The second testing phase ends (T = 18000, M = 1200)

The results show that alternative node 15 has out-performed node 0. Out of the 1200 examples it made the correct prediction for 1,050 of them whereas node 0 correctly predicted only 950 of them. Thus node 15 replaces node 0, in this case as the root node of the tree.

The new tree now looks like this (Figure 22.22).

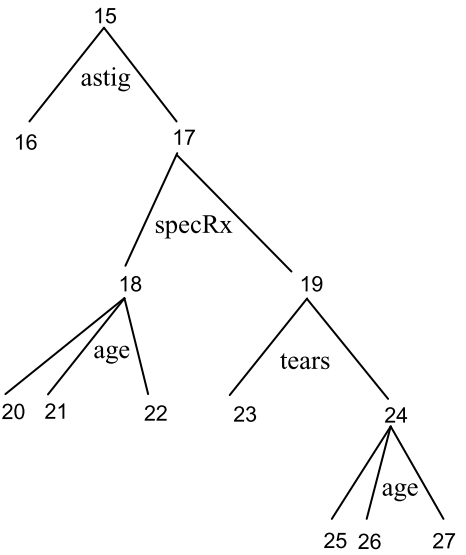

Figure 22.22 Tree After Substitution of Alternate Node 15 for Root Node

38,182 Records

Attribute 22 is split on attribute *tears*, giving new nodes 28 and 29. The tree is now identical to Figure 22.15.

38,400 Records

The data returns to standard mode.

At this point the rules corresponding to the nodes in the tree are identical to those shown in Figure 22.16.

40,198 Records

Attribute 23 is split on attribute *age*, giving new nodes 30, 31 and 32. The tree is starting its journey back towards Figure 22.13, i.e. the shape it had after 9,600 records.

42,000 Records

The third check for concept drift is made (D = 42000). Now nodes 18 and 19 are found to be 'suspect'. Node 18 is given an alternate node 33, from which hangs a one-level tree split on attribute *tears* (nodes 34 and 35). Node 19 is given an alternate node 36 from which hangs a one-level subtree split on attribute *age* (nodes 37, 38 and 39).

The tree now looks like Figure 22.23.

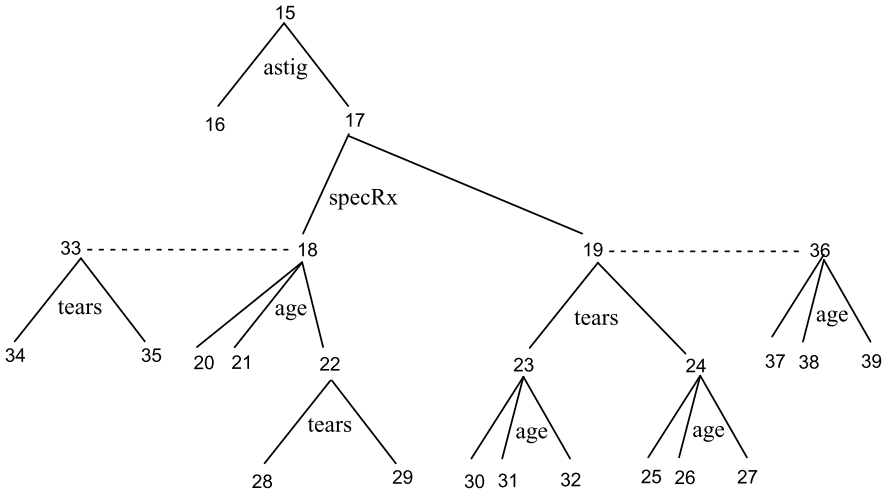

Figure 22.23 Tree After 42,000 Records

47,984 Records

Node 37 is split on attribute *tears* (giving new nodes 40 and 41).

48,000 Records

The rules corresponding to the tree are now as shown in Figure 22.24:

54,000 Records

The third testing phase begins (T = 18000)

55,200 Records

The third testing phase ends (T = 18000, M = 1200). Node 18 is replaced by alternative node 33.

1. IF $astig = 1$ THEN Class $= \{0, 2000, 2800\}$
2. IF $astig = 2$ AND $specRx = 1$ AND $age = 1$ THEN Class $= \{400, 0, 400\}$
3. IF $astig = 2$ AND $specRx = 1$ AND $age = 2$ THEN Class $= \{400, 0, 400\}$
4. IF $astig = 2$ AND $specRx = 1$ AND $age = 3$ AND $tears = 1$ THEN Class $= 3$
5. IF $astig = 2$ AND $specRx = 1$ AND $age = 3$ AND $tears = 2$ THEN Class $= 1$
6. IF $astig = 2$ AND $specRx = 2$ AND $tears = 1$ AND $age = 1$ THEN Class $= 3$
7. IF $astig = 2$ AND $specRx = 2$ AND $tears = 1$ AND $age = 2$ THEN Class $= 3$
8. IF $astig = 2$ AND $specRx = 2$ AND $tears = 1$ AND $age = 3$ THEN Class $= 3$
9. IF $astig = 2$ AND $specRx = 2$ AND $tears = 2$ AND $age = 1$ THEN Class $= 1$
10. IF $astig = 2$ AND $specRx = 2$ AND $tears = 2$ AND $age = 2$ THEN Class $= 3$
11. IF $astig = 2$ AND $specRx = 2$ AND $tears = 2$ AND $age = 3$ THEN Class $= 3$

Figure 22.24 Rules After 48,000 Records

56,000 Records

The fourth check for concept drift is made (D = 14000). Node 17 is considered suspect and is replaced by node 42 which has a one-level descendant subtree split on attribute *tears* (new nodes 43 and 44). The state of the tree is now as shown in Figure 22.25.

It seems to be much harder for the tree to get back to the shape it had in Figure 22.13 than it was to get from there to Figure 22.15. The prevalence of splits on attribute *tears* appears to be a partial substitute for splitting on that attribute at the root node.

57,600 Records

The rules corresponding to the classification tree are now as shown in Figure 22.26.

The alternative data mode begins.

57,996 Records

Node 43 is split on attribute *specRx* (giving new nodes 45 and 46).

58,000 Records

Node 44 is split on attribute *specRx* (giving new nodes 47 and 48).

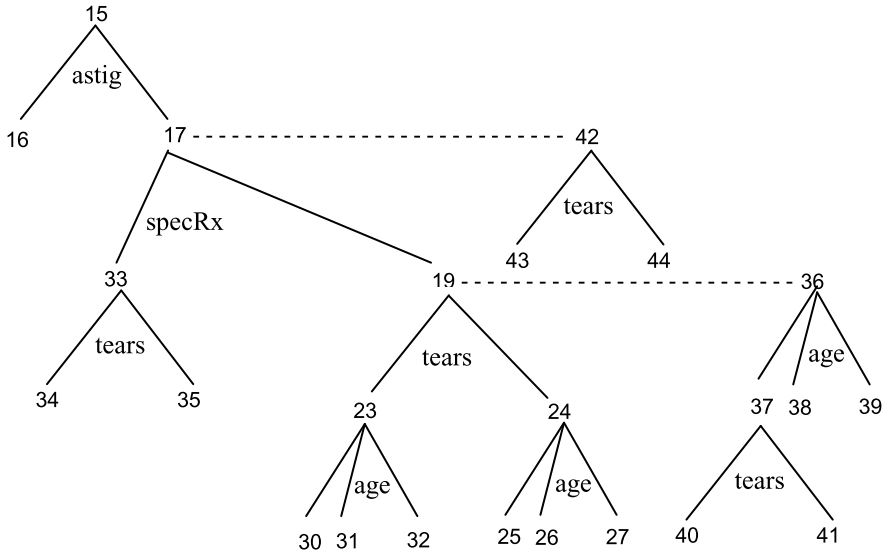

Figure 22.25 Tree After 56,000 Records

1. IF $astig = 1$ THEN Class $= \{0, 2000, 2800\}$
2. IF $astig = 2$ AND $specRx = 1$ AND $tears = 1$ THEN Class $= 3$
3. IF $astig = 2$ AND $specRx = 1$ AND $tears = 2$ THEN Class $= 1$
4. IF $astig = 2$ AND $specRx = 2$ AND $tears = 1$ AND $age = 1$ THEN Class $= 3$
5. IF $astig = 2$ AND $specRx = 2$ AND $tears = 1$ AND $age = 2$ THEN Class $= 3$
6. IF $astig = 2$ AND $specRx = 2$ AND $tears = 1$ AND $age = 3$ THEN Class $= 3$
7. IF $astig = 2$ AND $specRx = 2$ AND $tears = 2$ AND $age = 1$ THEN Class $= 1$
8. IF $astig = 2$ AND $specRx = 2$ AND $tears = 2$ AND $age = 2$ THEN Class $= 3$
9. IF $astig = 2$ AND $specRx = 2$ AND $tears = 2$ AND $age = 3$ THEN Class $= 3$

Figure 22.26 Rules After 57,600 Records

60,000 Records

We will leave the continuing story here. The tree now has nine corresponding rules, shown in Figure 22.27.

1. IF $astig = 1$ THEN Class $= \{0, 1500, 3300\}$
2. IF $astig = 2$ AND $specRx = 1$ AND $tears = 1$ THEN Class $= \{0, 200, 1000\}$
3. IF $astig = 2$ AND $specRx = 1$ AND $tears = 2$ THEN Class $= \{900, 300, 0\}$
4. IF $astig = 2$ AND $specRx = 2$ AND $tears = 1$ AND $age = 1$ THEN Class $= \{100, 0, 300\}$
5. IF $astig = 2$ AND $specRx = 2$ AND $tears = 1$ AND $age = 2$ THEN Class $= \{100, 0, 300\}$
6. IF $astig = 2$ AND $specRx = 2$ AND $tears = 1$ AND $age = 3$ THEN Class $= \{100, 0, 300\}$
7. IF $astig = 2$ AND $specRx = 2$ AND $tears = 2$ AND $age = 1$ THEN Class $= 1$
8. IF $astig = 2$ AND $specRx = 2$ AND $tears = 2$ AND $age = 2$ THEN Class $= 3$
9. IF $astig = 2$ AND $specRx = 2$ AND $tears = 2$ AND $age = 3$ THEN Class $= 3$

Figure 22.27 Rules After 60,000 Records

Following all the changes caused by the concept drift introduced into the stream of input records, the tree now misclassifies five of each batch of 24 records.

22.11.4 Comments on Experiment

Starting from Figure 22.13 with the data in standard mode after 9,600 records, the tree evolved very satisfactorily into Figure 22.15[9] with the data then in alternative mode. However this depended crucially on a suitable choice of variables W, D, T and M.

[9]Strictly, the nodes were numbered differently from Figure 22.15, but in the same order.

After a total of 60,000 records the tree still showed little sign of evolving back to Figure 22.13. The third check for concept drift, after 42,000 records, led to alternate nodes being associated with nodes 18 and 19 (Figure 22.23) but unfortunately not with the new root node 15. This in turn led to a drop in the predictive accuracy of the tree.

The algorithm appears to be very sensitive to the choice of variables, especially D and T. Their sizes relative to each other and to W may well be critical to the success of the algorithm on real-world data

22.12 Chapter Summary

This chapter builds on the description in Chapter 21 of the H-Tree algorithm for classifying *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. Chapter 21 was concerned with stationary data generated from a fixed causal model; Chapter 22 is concerned with data that is time-dependent, where the underlying model can change from time to time, perhaps seasonally. This phenomenon is known as *concept drift*.

The algorithm given here, *CDH-Tree*, is a variant of the popular CVFDT 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. A detailed example using synthetic data is given to illustrate the way in which the classification tree evolves as more and more records are processed in the presence of concept drift.

22.13 Self-assessment Exercises for Chapter 22

1. Under what circumstances would entering a testing phase not be appropriate? How can it be avoided?
2. Why would it not be appropriate to use the *hitcount* or *acvCounts* arrays when predicting the classification of a test record or a record with an unknown classification?

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] Hulten, G., Spencer, L., & Domingos, P. (2001). Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 97–106). New York: ACM.