JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Batch and Incremental Learning of Decision Trees

DIPLOMARBEIT

zur Erlangung des akademischen Grades

DIPLOMINGENIEUR

im Diplomstudium Mathematik

Angefertigt am *Institut für Wissensbasierte Mathematische Systeme*

Eingereicht von:

Zheng He

Betreuung:

Univ.-Prof. Dr. Erich Peter Klement
Dr. Edwin Lughofer

Linz, Oktober 2008

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Zheng He

To My Grandpa

Abstract

Nowadays, classification plays a very important role in data mining problems, and has been studied comprehensively in some research communities. In the dozens of classification models developed by scientists, decision trees are one of the most popular techniques. It is a decision support tool that uses a graph or model of decisions and their possible consequences, including chance event outcomes, resource costs and utility. A decision tree is used to identify the strategy most likely to reach a goal. In the tree structures, leaves represent classifications and branches represent conjunctions of features that lead to those classifications. The machine learning technique for inducing a decision tree from data is called decision tree learning, or decision tree for short.

In Chapter 2 of this paper, we will give a detailed introduction and comparison of three famous algorithms: ID3, C4.5 and CART in both theoretical and experimental aspects. In Chapter 3, we give a rough introduction of some typically incremental version of ID3 and CART, like ID4, ID5R, ITI and an extension algorithm of CART, to see however would they treat with incremental cases when instances come as a data stream. Then, in Chapter 4 we make an empirical comparison (and eventually a comparative analysis) between the different batch methods and the incremental methods. We will take a close look at the performance of ID3, C4.5 and CART with different parameters setting in the learning procedure (e.g. different prune levels, different selection resp. and stopping criteria and so on) . And also, we will see that the incremental approach can come close to the batch version of it with respect to classification accuracy. Finally, in the last Chapter, a conclusion and outlook will be presented for each approach of decision tree learning.

Acknowledgements

First of all, I want to thank my supervisor Dr. Edwin Lughofer, for assigning me such an interesting topic, giving me plenty of suggestion and support, and being very patient on correcting my paper.

I also want to thank Prof. Erich Peter Klement for providing me such a precious chance to do my thesis in FLLL, and Dr. Hennie ter Morsche for being my thesis judge representative from Eindhoven, and Dr. Roland Richter for coordinating everything at the beginning.

Then, I should thank Qingsong Liu, who gives me a strong help on algorithms programming, and also my brother Xiangming Cheng.

My special thanks goes to my girlfriend and my parents. Without their support, I cannot go through this tough period of life, not to mention this thesis.

Finally, I would like to thank all the teachers in Johannes Kepler University Linz and Technology University Eindhoven, the technical knowledge and skills inherited from them are the basis for this thesis.

Contents

1 Introduction **1**

1.1 Decision Trees 1

1.2 Incremental Learning of Decision Trees 2

2 Batch Learning Approaches for Decision Trees **4**

2.1 ID3 6

2.1.1 ID3 algorithm 6

2.1.2 Continuous-valued attributes 9

2.1.3 Selection criterion 11

2.1.4 Noise 12

2.1.5 Unknown attributes values 15

2.2 C4.5 18

2.2.1 Pruning decision trees 18

2.2.2 Early stopping by separate test error 21

2.2.3 From trees to rules 22

2.2.4 Windowing 24

2.3 CART 26

2.3.1 Splitting rules 26

2.3.2 Stopping criterion 29

2.3.3 Pruning 31

2.3.4 Outline 35

3 Incremental Decision Trees **36**

3.1 Incremental ID3 36

3.1.1 The ID4 algorithm 36

3.1.2 The ID5R algorithm 37

3.1.3 The ITI algorithm 47

3.2 Incremental CART 48

4 Evaluation **49**

4.1 Evaluation on "Image Segmentation" 50

4.2 Evaluation on "Soybean Classification" 55

4.3 Evaluation on "CD Imprint Inspection" 59

5 Conclusion and Outlook **63**

List of Figures

1.1 Examples of Decision Tree 2

2.1 Decision Tree for $Play\,Tennis$ 9
2.2 Decision Tree with continuous-valued attribute. 11
2.3 Decision tree generated with noise. 13
2.4 Modified decision tree with unknown attribute. 17
2.5 Decision tree before pruning. 19
2.6 Decision tree after pruning. 19
2.7 Simple decision tree for F=G=1 or J=K=1. 22
2.8 Decision tree for hypothyroid conditions. 23
2.9 Examples of different splits 27
2.10 Decision tree for fisher's iris. 33
2.11 Smallest tree within 1 std. error of minimum cost tree. 34
2.12 Decision tree for fisher's iris after pruning. 34

4.1 Smallest tree within 1 std. error of minimum cost tree for "Image Segmentation". 52
4.2 Classification performance of different noise level. 53
4.3 Smallest tree within 1 std. error of minimum cost tree for "Soybean". 56
4.4 Classification performance of different noise level. 57
4.5 Smallest tree within 1 std. error of minimum cost tree for "CD Imprint". 60
4.6 Classification performance of different noise level. 61

List of Tables

2.1 A small training set. 8
2.2 A training set with a continuous-valued attribute. 10
2.3 Gain for each threshold. 10
2.4 The training set with a threshold $\frac{0.72+0.87}{2} = 0.795$. 11
2.5 Attribute and classification. 13
2.6 The expected attribute value and classification. 13
2.7 Chi-squares of attributes. 14
2.8 Chi-squares distribution table. 14
2.9 The training set except one case with attribute of an unknown value. 15
2.10 The subset with unknown value for attribute 'Outlook'. 16
2.11 Results with congressional voting data. 21
2.12 Possible conditions to delete. 24
2.13 Windowing method to generate trees using hypothyroid data. 25

4.1 Comparison of trees generated by different classification approaches. 50
4.2 Comparison of different selection methods. 50
4.3 Comparison of different stopping criteria. 51
4.4 Comparison of different improved approaches. 51
4.5 Tree stability of different noise levels. 52
4.6 Comparison of trees generated by batch and incremental approaches. 53
4.7 Comparison of trees generated by different classification approaches. 55
4.8 Comparison of different selection methods. 55
4.9 Comparison of different stopping criteria. 55
4.10 Comparison of different improved approaches. 56
4.11 Tree stability of different noise levels. 57
4.12 Comparison of trees generated by batch and incremental approaches. 58
4.13 Comparison of trees generated by different classification approaches. 59
4.14 Comparison of different selection methods. 59
4.15 Comparison of different stopping criteria. 59

4.16 Comparison of different improved approaches. 60
4.17 Tree stability of different noise levels. 61
4.18 Comparison of trees generated by batch and incremental approaches. 62

Chapter 1

Introduction

1.1 Decision Trees

Nowadays, classification plays a very important role in data mining problems, and has been studied comprehensively in some research communities. During the recent decades, classification has also been successfully applied to various fields such as marketing management[1], credit approval[2], customer segment[3], equipment or medical diagnosis[4], performance prediction[5], surface inspection [6], novelty detection [7] and fault detection for quality assurance in industrial systems [8] [9]. The scientists applied several classification models like pattern classification [10], neural networks [11], statistical models such as linear or quadratic discriminant analysis [12], fuzzy classifiers [13] and decision trees.

Among the models mentioned above, decision trees are one of the most popular techniques of data mining.

1. are simple to understand and interpret. A non-expert is able to understand decision tree models after a brief explanation. This kind of function can be easily illustrated in Figure 1.1.
2. have value even with little complete data. Important insights can be generated based on experts describing a situation (its alternatives, probabilities, and costs) and their preferences for outcomes.
3. include an embedded feature selective process, omitting superfluous features.
4. have transparent description of the data result.
5. can be combined with other soft computing techniques, e.g. fuzzy decision trees, NN.

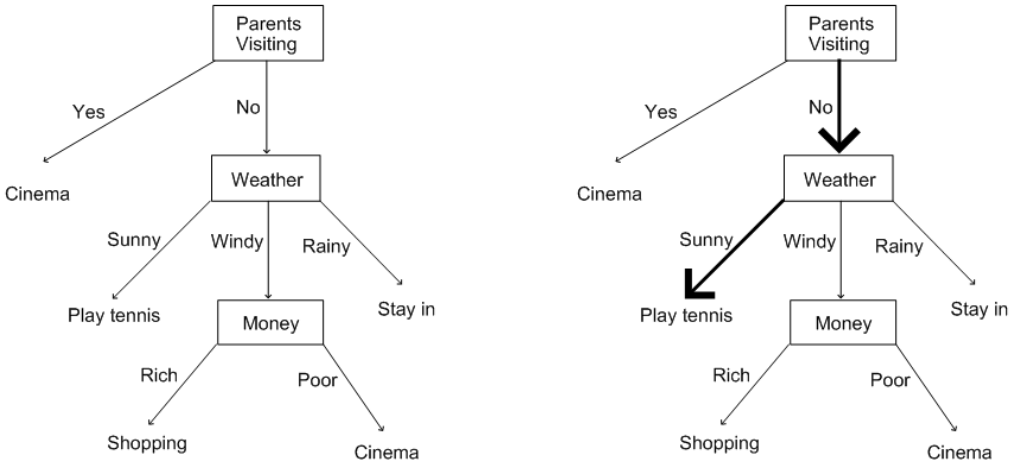
It is a decision support tool that uses a graph or model of decisions and their possible consequences, including chance event outcomes, resource costs and utility. A decision tree is used to identify the strategy most likely to reach a goal.

In the tree structures, leaves represent classifications and branches represent conjunctions of features that lead to those classifications. The machine learning technique for inducing a decision tree from data is called decision tree learning, or decision tree for short.

Decision tree training and classification The input to building a classifier is a *training data set* of records, each of which is tagged with a class label. A set of attribute values defines each

record. Attributes with discrete domains are referred to as *categorical*, whereas those with ordered domains are referred to as *numerical*. Training is the process of generating a concise description or model for each class of the training data set in terms of the predictor attributes defined in the data set. The model may be subsequently tested with a *test data set* for verification purpose and if the data set is large enough even on a separate (independent) validation data. This is usually attained within a cross-validation procedure [14], which partitions a sample of data into several subsets such that the analysis is initially performed on some of subsets, while validating can be employed on the initial analysis by using the other subsets. A final model is trained with optimal parameters and then used to classify future records whose classes are unknown.

Figure 1.1 shows an example of decision tree classifier (categorical) for some given training data set. It is used to find the most suitable arrangement for the weekend. The suitable arrangement will appear in one of the leaves of the tree.



(a) Decision tree from the training set.

(b) An instance tested through the tree.

Figure 1.1: Examples of decision tree.

Figure 1.1(a) is a decision tree constructed from some training set, and Figure 1.1(b) is the testing procedure of an instance from the root to the leaf. As the figures above show, if they do not want to visit their parents and the weather is sunny, they will finally choose to play tennis.

In Chapter 2, we will introduce three classic decision tree training methods, like ID3[15], C4.5[16] and CART[17].

1.2 Incremental Learning of Decision Trees

The data sets for data mining applications are usually large and may involve several millions of records. Each record typically consists of ten to hundreds attributes, some of them with large number of distinct values. In general, using large data sets results in the improvement in the accuracy of the classifier, but the capacity and complexity of the data involved in these applications make the task of training computationally very expensive. Since data sets are large, they cannot reside completely in memory, which makes I/O a significant bottleneck. Performing training such large data sets requires development of new techniques that limit accesses to the secondary storage in order to minimize the overall execution time.

Another problem that makes training more difficult is that most data sets are growing over time (e.g. in on-line applications), continuously making the previously constructed classifier obsolete. Addition of new records (or deletion of old records) may potentially invalidate the existing classifier that has been built on the old data set, which is still valid. However, constructing a classifier for large growing data set from scratch would be extremely wasteful and computationally prohibitive.

Many machine learning techniques and statistical methods have been developed for relative small data sets. These techniques are generally iterative in nature and require several passes over the data set, which make them inappropriate for data mining applications. As an alternative, we now introduce a prevailing technique called *Incremental Learning* as following:

Incremental Learning A decision tree *T* has been built for a data set *D* of size *N* by a decision tree classification algorithm. After a certain period of time, a new incremental data set *d* of size *n* has been collected. We are now to build a decision tree T' for the combined data set *D*+*d*. This problem is nontrivial because addition of new records may change some splitting points and thus consequently the new decision tree T' may be quite different from *T*. The goal is to build a decision tree classifier T' with minimal computational cost such that T' is comparably as accurate as the decision tree that would be built for *D*+*d* from scratch.

Formally, let *M* be a method (either a framework or an algorithm) for incremental decision tree classifier. Then, the decision tree *T* induced for *D* needs to be transformed by *M* as follows:

$$M(T, d) = T'$$

We want to keep the computational cost of inducing T' from T for incremental data set d much lower than that of inducing T' from $D + d$ from scratch, while generating T' of equal or at least slightly worse quality.

In Chapter 3, we will discuss possible approaches for incremental learning of decision trees, including ID4, ID5R[22], ITI[23] and incremental CART[25].

Chapter 2

Batch Learning Approaches for Decision Trees

Batch learning for decision trees get a whole training matrix as input, e.g.:

$$data\_matrix = \begin{bmatrix} & x_1 & x_2 & ... & x_n & label\ entry \\ row_1 & ... & ... & ... & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ row_N & ... & ... & ... & ... & 1 \end{bmatrix}$$

where $row_1, ..., row_N$ the N data points and $x_1, x_2, ..., x_n$ the n input variables, in the last column the label entry (as integer number). The goal of the batch learning algorithms is to process all training instances at once rather than incremental learning that updates a hypothesis after each example.

The most well-known and widely used batch training procedures for decision trees are:

- ID3 [15]:

  is a decision tree building algorithm which determines the classification of objects by testing the values of their properties. It builds the tree in a top down fashion, starting from a set of objects and a specification of properties. At each node of the tree, a property is tested and the results used to partition the object set. This process is recursively done till the set in a given subtree is homogeneous with respect to the classification criteria - in other words it contains objects belonging to the same category. This becomes a leaf node. At each node, the property to test is chosen based on information theoretic criteria that seek to maximize information gain and minimize entropy. In simpler terms, that property is tested which divides the candidate set in the most homogeneous subsets.

- C4.5 [16]:

  generates a classification-decision tree for the given data-set by recursive partitioning of data. The decision is grown using Depth-first strategy. The algorithm considers all the possible tests that can split the data set and selects a test that gives the best information gain. For each discrete attribute, one test with outcomes as many as the number of distinct values of the attribute is considered. For each continuous attribute, binary tests involving every

distinct values of the attribute are considered. In order to gather the entropy gain of all these binary tests efficiently, the training data set belonging to the node in consideration is sorted for the values of the continuous attribute and the entropy gains of the binary cut based on each distinct values are calculated in one scan of the sorted data. This process is repeated for each continuous attribute.

- CART [17]:

  is based on Classification and Regression Trees by Breiman et al [17] and has become a common basic method for building statistical models from simple feature data. The basic algorithm is based on the following: given a set of samples (a feature vector) find the question about some feature which splits the data minimizing the mean "impurity" of the two partitions, and recursively apply this splitting on each partition until some stop criteria is reached (e.g. a minimum number of samples in the partition). CART is powerful because it can deal with incomplete data, multiple types of features (floats, categorial sets) both in input features and predicted features, and the trees it produces often contain rules which are readable by humans.

In the subsequent schemes, we will describe these approaches in detail. There we will point out the basic training algorithms and some important extension towards missing values in the above matrices, unknown values, dealing with noise in the data and some other important issues.

2.1 ID3

2.1.1 ID3 algorithm

ID3 is actually a member of *Top-Down Induction of Decision Trees* (palindromically called *TDIDT* for short) family of learning systems. The main conceptional loop of TDIDT can be described four steps as below:

1. Choose the "best" decision attribute for each node.
2. For each value of the attribute, create a new branch and a new descendant.
3. Divide the instances according to the decision attribute into subsets and store the subsets to leaf nodes.
4. IF training examples are perfectly classified, then STOP, ELSE iterate over all new leaf nodes.

Technically, we implement this method by two existing well-known algorithms: node-wise growing and level-wise growing. For the former choice, one starts at the root and generates branches as far as possible at each node until all requirements are satisfied. Details about this algorithm are listed as below:

1. Input: training instances
2. Start: current_node = root_node, call ID3(current_node)
3. Function ID3(current_node)
   (a) Choose the "best" decision attribute for the current node.
   (b) For each value of the attribute, create a new branch and a new descendant.
   (c) Divide the instances according to the decision attribute into subsets and store the subsets to leaf nodes.
   (d) For each subset do
      i. If it contains one unique class label for all instances,
      assign this label to the leaf node and the leaf node becomes a terminal node
      ii. Else
      call ID3(current_leafnode)
   (e) End for
4. End function

For the level-wise version, one has to simply go through all new leaf nodes expanded from one node (i.e. all nodes from the new depth-level), storing all nodes and corresponding subsets into a list and do the whole procedure from the beginning for all entries in the list.

As mentioned in the first chapter, each instance is described in terms of a collection of attributes and each attribute has either some mutually exclusive discrete values or continuous values. In ID3 program, each training data set is described as a list of attribute-value pairs, which constitutes a conjunctive description of that data set. The data set is labeled with the name of the class to which it belongs. To make the situation simple, we assume that the training data set contains two classes of instances, P and N. Before introducing ID3, we firstly look at some notation and definition:

- A, the set of attributes.
- a_i, the individual attribute, with i=1, ... ,|A|.
- V_i, the set of possible values for a_i.
- v_{ij}, the individual attribute value, with j=1, ... ,$|V_i|$.
- E-score, the expected information function E of an attribute at a node.
- p, number of instances in P.
- n, number of instances in N.
- p_{ij}, number of instances in P with value v_{ij} in a_i.
- n_{ij}, number of instances in N with value v_{ij} in a_i.

Then, the expected information function E-score is formulated as below:

$$E(a_i) = \sum_{j=1}^{|V_i|} \frac{p_{ij} + n_{ij}}{p + n} I(p_{ij}, n_{ij}) \tag{2.1}$$

with

$$I(x, y) = \begin{cases} 0 & if\ x = 0 \\ 0 & if\ y = 0 \\ -\frac{x}{x+y} \log \frac{x}{x+y} - \frac{y}{x+y} \log \frac{y}{x+y} & otherwise \end{cases} .$$

Thus, the information gain expresses as

$$gain(a_i) = I(p, n) - E(a_i). \tag{2.2}$$

The complete ID3 algorithm includes a iterative method for selecting a set of training instances from which the decision tree will be built. This method is called the basic ID3 tree-construction algorithm, which will be embedded within the complete ID3 algorithm.

1. Input: training instances
2. Start: current_node = root_node, call ID3(current_node)
3. If all the instances are from exactly one class, then the decision tree is an terminal node containing that class label.
4. Function ID3(current_node)
   (a) Calculate the E-score as in (2.1) for all features.
   (b) Define a_{best} to be an attribute with the lowest E-score for the current node.
   (c) For each value $v_{best,j}$ of the a_{best}, create a new branch and a new descendant.
   (d) Divide the instances according to the decision attribute into subsets and store the subsets to leaf nodes.
   (e) For each subset do
      i. If it contains one unique class label for all instances, assign this label to the leaf node and the leaf node becomes a terminal node

ii. Else Call ID3(current_node)

(f) End for

5. End function

Let us look at a small example to learn how ID3 works. Table 2.1 shows a small training set, the instance are concerned about the weather status on Saturday mornings, and the task is to decide whether to play tennis or not. Each instance's value of each attribute is shown, together with the class of the instance (P or N). Using ID3, we can train these instances to form a decision tree.

No.	Attributes				Class
	Outlook	Temperature	Humidity	Windy	
1	sunny	hot	high	false	N
2	sunny	hot	high	true	N
3	overcast	hot	high	false	P
4	rain	mild	high	false	P
5	rain	cool	normal	false	P
6	rain	cool	normal	true	N
7	overcast	cool	normal	true	P
8	sunny	mild	high	false	N
9	sunny	cool	normal	false	P
10	rain	mild	normal	false	P
11	sunny	mild	normal	true	P
12	overcast	mild	high	true	P
13	overcast	hot	normal	false	P
14	rain	mild	high	true	N

Table 2.1: A small training set.

To illustrate the idea of ID3 algorithm, we first calculate the information required for classification. Since in Table 2.1, 9 instances are of class P and 5 instance are of class N, thus,

$$I(p,n) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{9}{14} = 0.940.$$

Then we consider the outlook attribute with values sunny, overcast, rain. 5 instance in Table 2.1 have the first value sunny, 2 of them from class P and three from class N. So

$$p_1 = 2,\ n_1 = 3,\ I(p_1,n_1) = 0.971.$$

Similarly,

$$p_2 = 4,\ n_2 = 0,\ I(p_2,n_2) = 0,$$

and

$$p_3 = 3,\ n_3 = 2,\ I(p_3,n_3) = 0.971.$$

Therefore, the E-score of outlook after testing all its values is

$$E(outlook) = \frac{5}{14}I(p_1,n_1) + \frac{4}{14}I(p_2,n_2) + \frac{5}{14}I(p_3,n_3) = 0.694.$$

The gain of attribute 'outlook' is then

$$gain(outlook) = 0.940 - E(outlook) = 0.246.$$

Same procedure provides

$$gain(temperature) = 0.029$$
$$gain(humidity) = 0.152$$
$$gain(windy) = 0.048.$$

So the tree-constructing method used in ID3 would choose 'outlook' as the attribute for the root of the decision tree. Based on this root, the instance would then be divided into subsets according to their values of 'outlook' and subtrees will be built with the same fashion of each subset. Finally, Figure 2.1 shows the actual decision tree created by ID3 from the training set of Table 2.1.

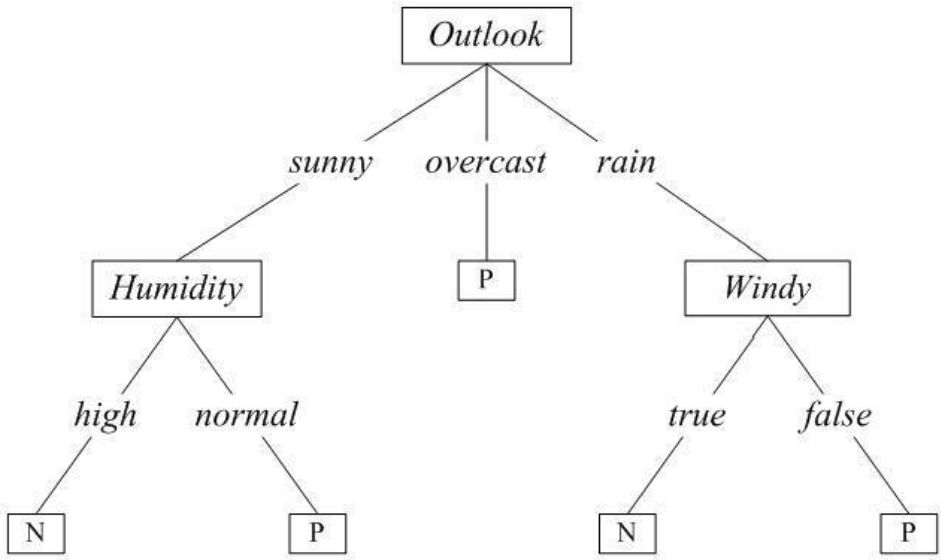


Figure 2.1: Decision Tree for $Play\ Tennis$

2.1.2 Continuous-valued attributes

We have generated a decision tree from some discrete-valued attributes as above. However, in real world scenarios, we are more likely to come up against continuous-valued attributes (see also Chapter 4). What we could do is to transfer numerical attributes back to categorial ones in some sense. To illustrate this problem, we introduce a new example in Table 2.2 which only made slight changes of the before one.

Now, we resort the distinct values of the attribute as V_1, V_2, ... , V_k. Then, each pair of values V_i, V_{i+1} suggests a possible threshold $\frac{V_i+V_{i+1}}{2}$ that divides the instances of the training set into two subsets. In this sense, the tree is split into two branches according to the rules: assign instances $< \frac{V_i+V_{i+1}}{2}$ to the leftward of $\frac{V_i+V_{i+1}}{2}$ and instances $> \frac{V_i+V_{i+1}}{2}$ to the rightward. We would like to pick out the threshold which produces the greatest information gain. To achieve this purpose, we need to calculate the gain function one by one for each threshold.

No.	Attributes				Class
	Outlook	Temperature	Humidity	Windy	
1	sunny	hot	0.90	false	N
2	sunny	hot	0.87	true	N
3	overcast	hot	0.93	false	P
4	rain	mild	0.89	false	P
5	rain	cool	0.80	false	P
6	rain	cool	0.59	true	N
7	overcast	cool	0.77	true	P
8	sunny	mild	0.91	false	N
9	sunny	cool	0.68	false	P
10	rain	mild	0.84	false	P
11	sunny	mild	0.72	true	P
12	overcast	mild	0.49	true	P
13	overcast	hot	0.74	false	P
14	rain	mild	0.86	true	N

Table 2.2: A training set with a continuous-valued attribute.

We reorder the instances by the magnitude of 'Humidity' as Table 2.3. Then, we calculate Gain function for each threshold, which is set as above explained. The largest one of Gain value goes to the splitting with threshold 0.85, which is less than the $Gain(Outlook)$ all the same. Thus, the continuous-valued 'Humidity' makes no change of the first level of the tree.

No.	Humidity	Class	Threshold	Gain
1	0.49	P	0.48	0
2	0.59	N	0.54	0.047
3	0.68	P	0.635	0.010
4	0.72	P	0.70	0
5	0.74	P	0.73	0.015
6	0.77	P	0.755	0.045
7	0.80	P	0.785	0.090
8	0.84	P	0.82	0.152
9	0.86	N	0.85	0.236
10	0.87	N	0.865	0.102
11	0.89	P	0.88	0.025
12	0.90	N	0.895	0.079
13	0.91	N	0.905	0.010
14	0.93	P	0.92	0.047
15			0.94	0

Table 2.3: Gain for each threshold.

However, at the branch "Outlook=sunny", we have found that 'Humidity' with the threshold 0.795 has the greatest gain and thus is the best choice for splitting. Figure 2.2 depicts the new tree with continuous-valued 'Humidity'.

Humidity	0.68	0.72	0.87	0.90	0.91
Classification	P	P	N	N	N

Table 2.4: The training set with a threshold $\frac{0.72+0.87}{2} = 0.795$.

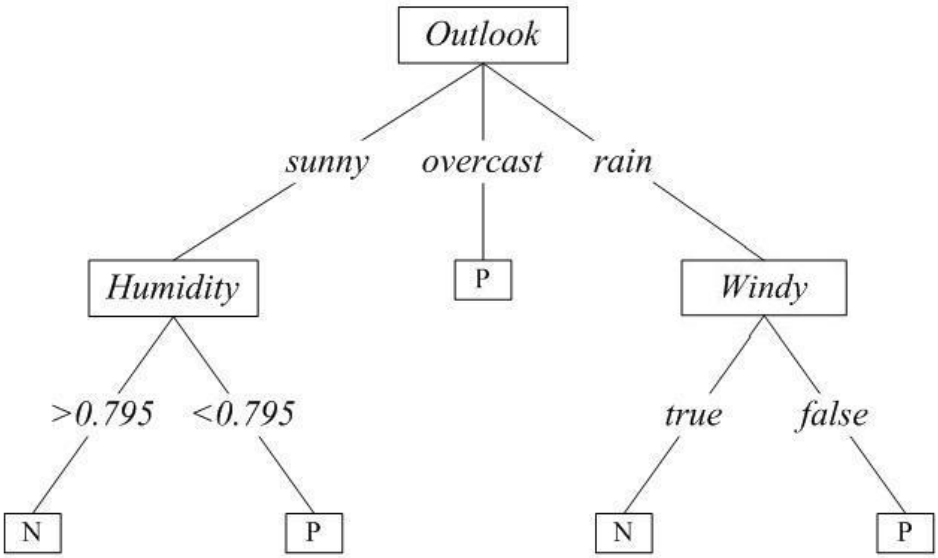


Figure 2.2: Decision Tree with continuous-valued attribute.

Consequently, if there are only continuous features in the above training set (which is quite usual in real world tasks), this procedure always works in a binary tree.

2.1.3 Selection criterion

The ID3 algorithm we use above, is based on the gain criterion. Although we can obtain quite good results from it, there is a serious deficiency of this criterion: it has a considerable bias in favor of tests with many outcomes. It means that sometimes, the partitioning will lead to a large number of subsets, each of which contains just one instance and hence causes a poor generalization behavior on unseen data. So the information gain of each subbranch is consequently maximal, and thus the division is quite useless. To adjust this bias, a normalization method called $gain\ ratio$ is applied. As before, let attribute a_i have values $v_{i1}, v_{i2}, ..., v_{ij}$ and let the numbers of classes of v_{ij} be p_{ij} and n_{ij} respectively. Then, the following expression

$$IV(a_i) = -\sum_{j=1}^{|V_i|} \frac{p_{ij}+n_{ij}}{p+n} \log_2 \frac{p_{ij}+n_{ij}}{p+n}$$

measures the information value of attribute a_i which is analogy with the definition of information function. Then,

$$gain\ ratio(a_i) = gain(a_i)/IV(a_i)$$

represents the proportion of the information generated by the split at nodes which is useful for classification. If the split is nearly trivial, $IV(a_i)$ will be small and the gain ratio will be unstable. To avoid this, the gain ratio criterion selects a test to maximize the above function, subject to the constraint that the information gain must be large, i.e. at least as large as the average gain over all tests examined. Recall the results before, the information gain of the four attributes is given as

$$\begin{gathered} gain(outlook) = 0.246 \\ gain(temperature) = 0.029 \\ gain(humidity) = 0.151 \\ gain(windy) = 0.048. \end{gathered}$$

We can easily observe that only 'Outlook' and 'Humidity' have above-average gain. The information obtained by determining the value of the 'Outlook' is therefore

$$IV(outlook) = -\frac{5}{14}\log_2\frac{4}{14} - \frac{4}{14}\log_2\frac{5}{14} - \frac{5}{14}\log_2\frac{5}{14} = 1.578.$$

Similarly,

$$IV(humidity) = -\frac{7}{14}\log_2\frac{7}{14} - \frac{7}{14}\log_2\frac{7}{14} = 1.$$

Thus,

$$\begin{gathered} gain\ ratio(outlook) = 0.246/1.578 = 0.156 \\ gain\ ratio(humidity) = 0.151/1.000 = 0.151. \end{gathered}$$

Though, the gain ratio criterion would still choose 'Outlook' as the root node, its superiority over 'Humidity' is now much reduced.

2.1.4 Noise

Until now, we assumed the data in the training set to be completely accurate. Unfortunately, data of industrial tasks in real-world usually do not ensure this assumption. Measurements or subjective judgements may give rise to errors in the values of attributes, which are usually called $noise$.

With the effecting of noise, some of instances in the training set may even be misclassified. To illustrate the idea, let us turn to the training set in Table 2.1 again. Suppose that the outlook of instance 6 is incorrectly recorded as overcast. Then, instances 6 and 7 have the same descriptions but are assigned to different classes. Moreover, if the class of instance 3 was misarranged to N, after training this misemployed set, the decision tree was generated in a totally different shape which contains 15 nodes illustrated as Figure 2.3.

From these points of view, we know that, there are two problems brought from noise:

1. Some of the attributes turn to be inadequate with errors in the training set.
2. Decision tree built from this corrupt training set become illusively complex.

To make the tree generating algorithm stably and effectively apply to the noise-interfered training set, two requirements should be satisfied:

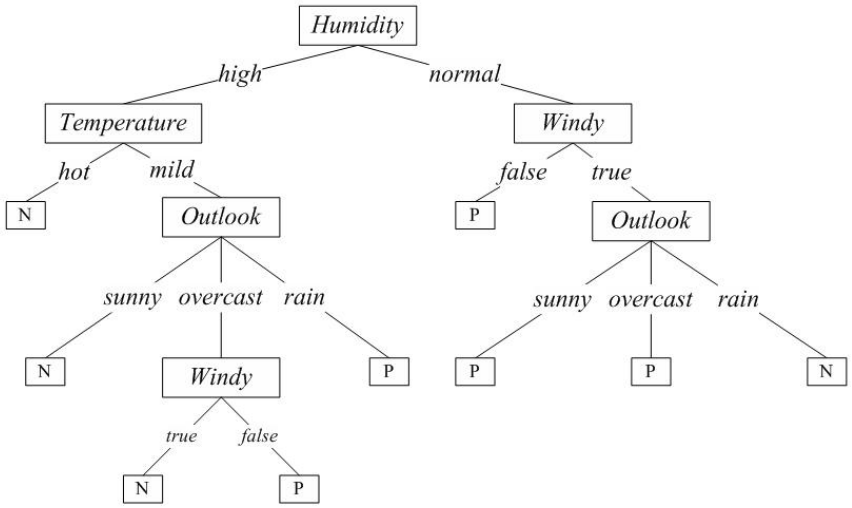


Figure 2.3: Decision tree generated with noise.

1. The algorithm should be able to deal with inadequate attributes.
2. The algorithm should be able to determine that testing further attributes will not improve the predictive accuracy of the decision tree.

To decide whether an attribute is relevant to the classification or not, we begin with the second requirement. If the values of an attribute a_i is randomly assigned, it cannot help us effectively to classify the instances in the training set. However, testing a_i is a meaningful step. One feasible solution is based on the chi-square test for stochastic independence. The observed classification of the attribute a_i is listed as below:

a_i \ Class	P	N
v_{i1}	p_{i1}	n_{i1}
v_{i2}	p_{i2}	n_{i2}
$\cdots$	$\cdots$	$\cdots$

Table 2.5: Attribute and classification.

We take the corresponding expected value of p_{ij} as $p_{ij}^{'} = p \cdot \frac{p_{ij}+n_{ij}}{p+n}$ for the irrelevant attribute value and $n_{ij}^{'}$ defined in the same way, then the we have

a_i \ Class	P	N
v_{i1}	p'_{i1}	n'_{i1}
v_{i2}	p'_{i2}	n'_{i2}
$\cdots$	$\cdots$	$\cdots$

Table 2.6: The expected attribute value and classification.

The statistic

$$\sum_{j=1}^{|V_i|} \frac{(p_{ij} - p_{ij}^{'})^2}{p_{ij}^{'}} + \frac{(n_{ij} - n_{ij}^{'})^2}{n_{ij}^{'}}$$

is approximately chi-square with 1 degree of freedom $((2-1)\times(2-1))$. Assumed that $p_{ij}^{'}$ and $n_{ij}^{'}$ are not very small, this statistic can thus be used to determine the confidence level with which one could refuse the proposition that a_i is independent of a given classification. Let us take instances from Table 2.1 as an example, and give the null hypothesis as below:

H_0 : the attribute is independent of the classification.

H_1 : the attribute is dependent of the classification.

Using the above equation, we can calculate the chi-square of each attribute as below:

Attribute	Chi-square
Outlook	3.5467
Temperature	0.5704
Humidity	2.8000
Windy	0.9333

Table 2.7: Chi-squares of attributes.

SL / DF	0.5	0.10	0.05	0.02	0.01	0.001
1	0.455	2.706	3.841	5.412	6.635	10.827
2	1.386	4.605	5.991	7.824	9.210	13.815
3	2.366	6.251	7.815	9.837	11.345	16.268
4	3.357	7.779	9.488	11.668	13.277	18.465
5	4.351	9.236	11.070	13.388	15.086	20.517

Table 2.8: Chi-squares distribution table.

Checking the chi-square distribution Table 2.8 with 1 degree of freedom and reading along the row we find our value of $\mathcal{X}^2$ (3.5467) lies between 2.706 and 3.841. This is below the conventionally accepted significance level of 0.05 or 5%, so the null hypothesis that the two distributions are the same is verified. In other word, when the computed $\mathcal{X}^2$ statistic exceeds the critical value in the table for a 0.05 probability level, then we can reject the null hypothesis of equal distributions. Since the statistic $\mathcal{X}^2$ (3.5467) does not exceed the critical value for 0.05 significance level, we can accept the null hypothesis of the attribute 'Outlook' is independent of the classification. Similarly, we can determine the confidence level of the proposition that a_i is dependent of the given classification for each attribute. In this chi-square test, we actually refused all the attribute, since they are irrelevant to the classification due to the results. However, 'Outlook' has the largest chi-square, which means it is more likely to be relevant to the classification. If the number of instances increase, accepting 'Outlook' as a relevant attribute would be possible, which is more or less consistent with what we discussed before (choosing 'Outlook' as the the root of the decision tree).

The tree-generating procedure could then be modified to prevent testing any attribute whose irrelevant values cannot be filtered with a very high confidence level. We know from the above example that 'Temperature' and 'Windy' are irrelevant attributes of the classification with 0.10 confidence level. We could make this conclusion more precise, when more testing data and a more detailed chi-square distribution table is available. So, this method can be used in preventing over-complex trees built up to fit the noise and will not influence the performance of the tree-generating procedure with noise-free data set.

Now we go back to the first requirement. Sometimes, we may meet up with the problems in the following: some attributes are inadequate and thus unable to classify the instances in the training set; or some attributes are determined to be irrelevant to the class of the instances.

In order to conquer these above troubles, we add a label with class information to each leaf (the instances inside may not be in the same class). Two suggestion is laid out in literatures. A subset of instances could be labelled with a probability $\frac{p}{p+n}$ in $(0, 1)$. For example, a leaf with 0.8 indicates that the instances belongs to the class P with probability 0.8. Another approach assign a leaf to class P if $p > n$, to N if $p < n$, and to either if $p = n$. These two approaches minimize the sum of either squares of the error, or the absolute errors over all instances in training set.

2.1.5 Unknown attributes values

Another unfortunate fact is that we often meet data with $unknown\ attribute\ values$ due to some technical reasons. If we are not willing to discard a significant amount of incomplete data without classifying them, we ought to amend the algorithms such that they can cope with missing attribute values impactfully. In order to achieve this goal, we modify the gain function in the following way:

$$\begin{aligned} gain(a_i) = \ & P(the\ value\ of\ a_i\ is\ known) \times (I(p,n) - E(a_i)) \\ & + P(the\ value\ of\ a_i\ is\ not\ known) \times 0 \\ = \ & P(the\ value\ of\ a_i\ is\ known) \times (I(p,n) - E(a_i)). \end{aligned}$$

Similarly, the definition of $IV(a_i)$ can be altered by regarding the cases with unknown values as an additional group. Same as before, we use this modified gain function to select an attribute to be a testing node.

To illustrate the effect of these modifications, we recall the training set of Table 2.1 and suppose that instance 12 with unknown value for 'Outlook' (denoted by '?'). Then, we have only 13 remaining instances with known value for 'Outlook':

Outlook	P (Play)	N (Don't Play)	Total
sunny	2	3	5
overcast	3	0	3
rain	3	2	5
Total	8	5	13

Table 2.9: The training set except one case with attribute of an unknown value.

Applying the corresponding data to calculate the quantities relevant to the testing on 'Outlook' with known values, we could gain:

$$I(p,n) = -\frac{8}{13}\log_2\frac{8}{13} - \frac{5}{13}\log_2\frac{5}{13} = 0.961$$

$$\begin{aligned} E(outlook) = & \frac{5}{13} \times (-\frac{2}{5} \times \log_2 \frac{2}{5} - \frac{3}{5} \times \log_2 \frac{3}{5}) \\ & + \frac{3}{13} \times (-\frac{3}{3} \times \log_2 \frac{3}{3} - \frac{0}{3} \times \log_2 \frac{0}{3}) \\ & + \frac{5}{13} \times (-\frac{3}{5} \times \log_2 \frac{3}{5} - \frac{2}{5} \times \log_2 \frac{2}{5}) \\ = & \ 0.747 \end{aligned}$$

$$gain(outlook) = \frac{13}{14} \times (0.961 - 0.747) = 0.199.$$

We can notice that the gain for 'Outlook' for this testing is slightly lower than the previous one of value 0.246. The split information is still to be determined from the whole training set and is larger than before, since on extra category for unknown outcome which has to be taken into account:

$$\begin{aligned} IV(outlook) = & -\frac{5}{14} \times -\frac{5}{14} \quad (for\ sunny) \\ & -\frac{3}{14} \times -\frac{3}{14} \quad (for\ overcast) \\ & -\frac{5}{14} \times -\frac{5}{14} \quad (for\ rain) \\ & -\frac{1}{14} \times -\frac{1}{14} \quad (for\ ?) \\ = & \ 1.809 \end{aligned}$$

which turns larger than previous 1.578. The gain ratio thus decrease from 0.156 to 0.110.

It is obvious that 13 instances of total 14 training cases with known values for attribute 'Outlook' can be partitioned by this testing without any problem. The remaining case thus should be assigned to all subsets, which are sunny, overcast and rain, with weights 5/13, 3/13 and 5/13 respectively.

We then turn to the first subset of instances corresponding to outlook=sunny,

Outlook	Temperature	Humidity	Windy	Class	Weight
sunny	hot	high	false	N	1
sunny	hot	high	true	N	1
sunny	mild	high	false	N	1
sunny	cool	normal	false	P	1
sunny	mild	normal	true	P	1
?	mild	high	true	P	5/13

Table 2.10: The subset with unknown value for attribute 'Outlook'.

Further partition of this subset on the attribute 'Humidity' are displayed as:

$humidity$	P	N
$normal$	2	0
$high$	5/13	3

The decision tree almost has the same structure as before (shown in Figure 2.4), but now there are slight changes at leaves, in form of (N) or (N/E) according to the weights in the sample. N is the sum of the fractional instances that reach the leaves and E is the fractional number of misclassified instances. For example, N (3.4/0.4) means that 3.4 training cases reached this leaf, of which 0.4 did not belong to the class N.

Another simpler alternative is to assign the value to unknown cases, which has the highest frequency in the other completely known instances. Please note that this is also related somehow to semi-supervised learning algorithms. We would compare this two strategies in the evaluation section later to see how they perform for real-life data.

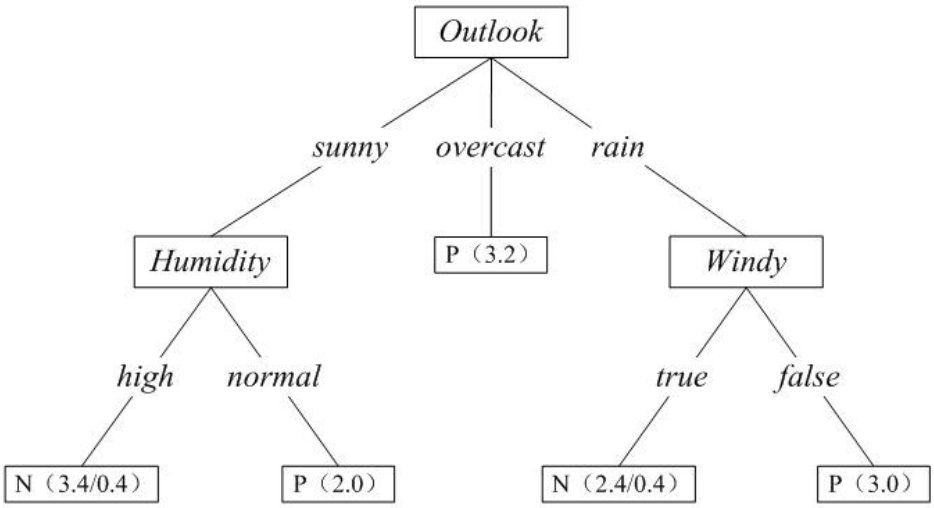


Figure 2.4: Modified decision tree with unknown attribute.

2.2 C4.5

The well-known learning algorithm of decision tree *C4.5* [16] is a successor and extension of ID3. It has the same procedure to construct decision tree from training data sets, deal with noise and treat attributes with unknown values as ID3, which we have described in the previous section. Besides, C4.5 has some special tricks for pruning, generalizing rules from decision trees and windowing, which we would like to introduce in the following contents.

2.2.1 Pruning decision trees

The recursive partitioning method of constructing decision trees will finally reach the leaves containing instances of a single class, or no test provides any improvement. The trees generated from original data sets are often too complex to be interpreted by non-experts and usually tend to overfit resulting in a bad generalization, which usually has an inefficient performance to deal with new unseen data. This means that the misclassification error increases significantly for new data to be classified. To conquer this inconvenient problem, we now present a new technique called "pruning" to adjust the original decision trees more structured.

Normally, there are two basic ways to modify the recursive partitioning method to obtain simpler trees:

1. make out restrains such that the partitioning can be stopped automatically.
2. remove retrospectively some of the structure built up by partitioning.

The former approach, sometimes named stopping or prepruning, can prevent the partitioning method from creating useless structure which will employ the tree-building procedure in a wrong direction (complex trees obtained). This technique sets some threshold to decide the occasion which attains the most appropriate leaves and reject further division. However, it is not easy to get a right stopping rules: if the threshold is too high, the division will terminate before the benefits of splits become evident; contrarily, if the threshold is too low, little effort contributes for trees simplification.

On the other hand, pruning a decision tree will inevitably cause misclassification, which means that the leaves of the pruned tree need not necessarily contain a single class instance. Therefore, we have to explore class distribution to designate the probability that a training case at the leaf belongs to that class.

Usually, we can simplify decision trees by removing one or more subtrees and replacing them with their leaves and assign the most frequent class to that leaf. Especially, C4.5 permits replacing a subtree by one of its branches. Both operations are illustrated in Figure 2.6 which a decision tree derived from congressional voting data before and after pruning. Recalling the notation (N) and (N/E) of previous section, N indicates the number of training instances in a leaf and E renders errors. As shown in Figure 2.5 and Figure 2.6, the first subtree is replaced by the leaf $democrat$ and similarly the second one has become the leaf $republican$. Meanwhile, the third subtree has been replaced by the subtree at its third branch.

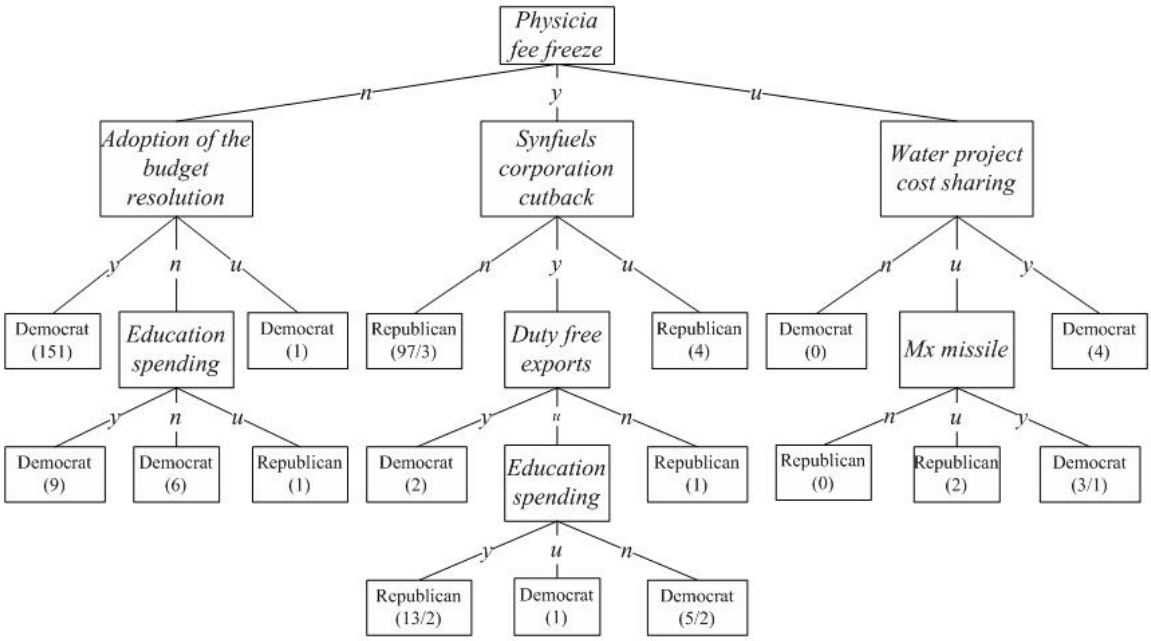


Figure 2.5: Decision tree before pruning.

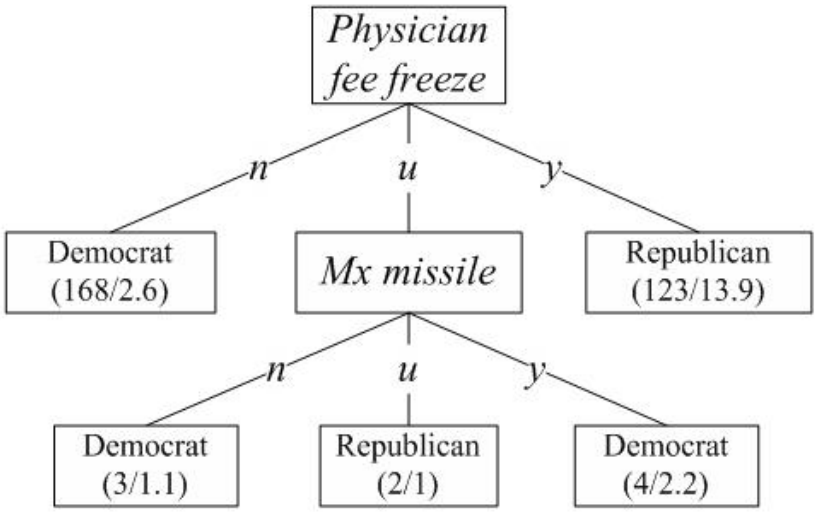


Figure 2.6: Decision tree after pruning.

C4.5 uses a pessimistic estimate to predict the error rate of a decision tree and its subtrees (including leaves)[18][19]. Supposed instance $X = (x_1, x_2, \ldots, x_n)$ are drawn from a point binomial distribution

$$f(x; p) = p^x(1-p)^{1-x}, \;\; x = 0,\ 1 \;\; and \;\; 0 \leq p \leq 1.$$

The maximum-likelihood estimate of p is

$$\hat{p} = y/N, \;\; y = \sum_{i=1}^{N} x_i.$$

Given N and $y = E$, we need to estimate the interval of p such that the probability that the actual parameter p falls within the interval equals the confidence level. Given confidence level $CF = 2\alpha$, the upper limit denoted by $U_\alpha(E, N)$ is the p value which satisfies

$$\sum_{i=0}^{E} \binom{N}{i} p_{ul}^i (1-p_{ul})^{N-i} = \frac{1-2\alpha}{2}.$$

Define

$$U_\alpha(E, N) = p_{ul}(E, N, confidence\ level\ is\ 2\alpha),$$

then, C4.5 simply employs this upper limit as the predicted error rate at each leaf. The task left is to minimize the observed error rate when constructing the decision tree. So, a leaf node containing N training instances with a predicted error rate $U_{CF}(E, N)$ would bring $N \times U_{CF}(E, N)$ predicted errors.

We now take a subtree from Figure 2.5 as an example to illustrate how this operation works. The subtree

$$\begin{gathered} educationspending = n : democrat(6) \\ educationspending = y : democrat(9) \\ educationspending = u : republican(1) \end{gathered}$$

has no associated errors on the training set. Let us take the first leaf as an example, setting $N = 6$, $E = 0$, and using the default confidence level of 25%, we have

$$\begin{gathered} (1-p_{ul})^6 = \tfrac{1-2\times 0.25}{2} = 0.25 \\ \Longrightarrow U_{25\%}(0, 6) = p_{ul}(0, 6, confidence\ level\ is\ 50\%) = 1 - \sqrt[6]{0.25} = 0.206 \end{gathered}$$

Thus, the predicted number of errors is 6×0.206. Similar calculations are applied to the other two leaves. Then, the total number of predicted errors for the subtree is given by

$$6 \times 0.206 + 9 \times 0.143 + 1 \times 0.750 = 3.273.$$

If the subtree were replaced by the leaf $democrat$, the corresponding predicted errors come to

$$16 \times U_{25\%}(1, 16) = 16 \times 0.157 = 2.512,$$

which is less than the original subtree. Thus, we could prune this subtree to one of its leaf.

The estimating error rates for the trees in Figure 2.5 and Figure 2.6 are presented as below:

Evaluation on training data (300 items):

Before Pruning		After Pruning		
Size	Errors	Size	Errors	Estimate
25	8(2.7%)	7	13(4.3%)	(6.9%)

Evaluation on test data (135 items):

Before Pruning		After Pruning		
Size	Errors	Size	Errors	Estimate
25	7(5.2%)	7	4(3.0%)	(6.9%)

Table 2.11: Results with congressional voting data.

This table clearly shows us that indeed the misclassification rate of the training data increases for the pruned tree; however, its generalization quality of the unseen data significantly improves. This is more important, as in practice a decision tree classifier is always applied for classifying new items.

2.2.2 Early stopping by separate test error

The method mentioned in the previous subsection was originally defined in the 80ties for small data sets, where ML approaches were applied to these kind of data sets. However, as ML systems moved out from laboratories to big industrial systems during the 90ties and 00s, mostly we have sufficient number of data samples available which can be also managed in nowadays PCs. With these large amount of data, initially, we would see that the generation of the tree get better and better, i.e., the error rate on the training set decreases. But at some point during training procedure, it actually begins to turn worse again, which means the error rate on the unseen data increases. This is what we called "overfitting". To conquer the overfitting problem, one could appeal to the alternative of *early stopping* during learning by eliciting the error on a separate test set for each new branch or leaf. This technique is widely used because it is simple to understand and implement. Below is the slight introduction on how to do early stopping by separate test error:

1. Divide the whole data set randomly into a training data set and a separate test data set (resp. sometimes at a system two or more data sets are measured independently, then simply take these two).
2. Create a initial tree only on the training set, and then evaluate the error rate on the test set.
3. Always check whether a further splitting (at an arbitrary node) decreases the error of the test data or not. If no, stop at this node and go to the next one; otherwise, further split this node.

This approach use the test set to predict the behavior in real use, assuming that the error rate on both will be similar: the test error is used as an estimate of the generalization error. For more details, see [20][21].

2.2.3 From trees to rules

Even though the pruned trees are more concise than the original ones, there is still large space for the classifiers to be simpler and more accurate. First, let us take a look at a simple example as shown in Figure 2.7 : four 0-1 valued attributes, F, G, J and K, and two classes, P and N.

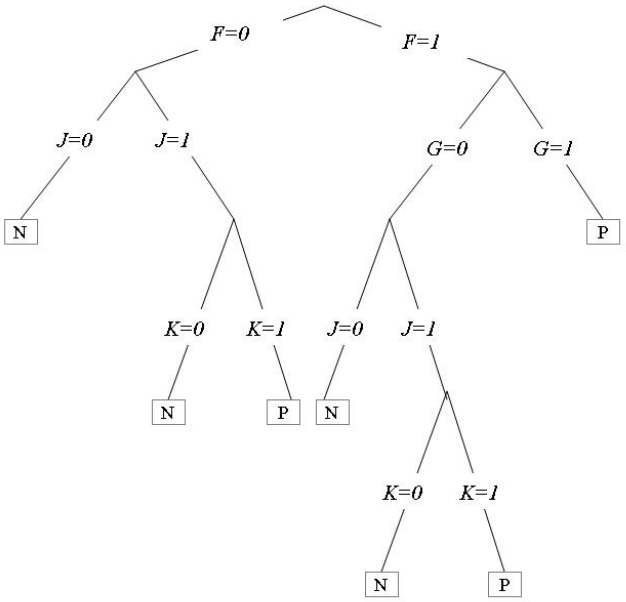


Figure 2.7: Simple decision tree for F=G=1 or J=K=1.

If a rule formulated in the following way, it exactly makes no difference from the original tree:

$$\begin{array}{ll} \textbf{if} & F = 1 \\ and & G = 0 \\ and & J = 1 \\ and & K = 1 \\ \textbf{then} & class\ P \end{array}$$

One can easily attain that the rules for this simple tree should take the form as below:

$$\begin{array}{ll} \textbf{if} & F = 1 \\ and & G = 1 \\ \textbf{then} & class\ P \end{array}$$

$$\begin{array}{ll} \textbf{if} & J = 1 \\ and & K = 1 \\ \textbf{then} & class\ P \end{array}$$

$$\textbf{otherwise} \quad class\ N.$$

How to extract rules from existing decision trees generally? Let rule R has the form

$$\textbf{if} \quad A \quad \textbf{then} \quad class\, C$$

and a more general rule R^- formulate as

$$\textbf{if} \quad A^- \quad \textbf{then} \quad class\, C,$$

in which A^- is obtained by deleting one condition X in A. Thus, each instance can be distributed to one of the following four cases:

	$belongs\ to\ class\ C$	$belongs\ to\ any\ other\ class$
$satisfies\ condition\ A$	Y_1	E_1
$satisfies\ condition\ A^-$	$Y_1 + Y_2$	$E_1 + E_2$

It is easily accessible that all cases satisfying R are also covered by R^-. So, the total number of cases satisfying R^- equals to $Y_1 + Y_2 + E_1 + E_2$. Consequently, the estimating error rate of R can be set as $U_{CF}(E_1, Y_1 + E_1)$, while that of R^- is resembled as $U_{CF}(E_1 + E_2, Y_1 + Y_2 + E_1 + E_2)$. As long as the pessimistic error rate of R^- is smaller than that of R, we could delete X to simplify the rule system. We repeat this procedure for each rule until the minimum value of the pessimistic error rate is reached. However, this greedy algorithm cannot ensure the global optimization.

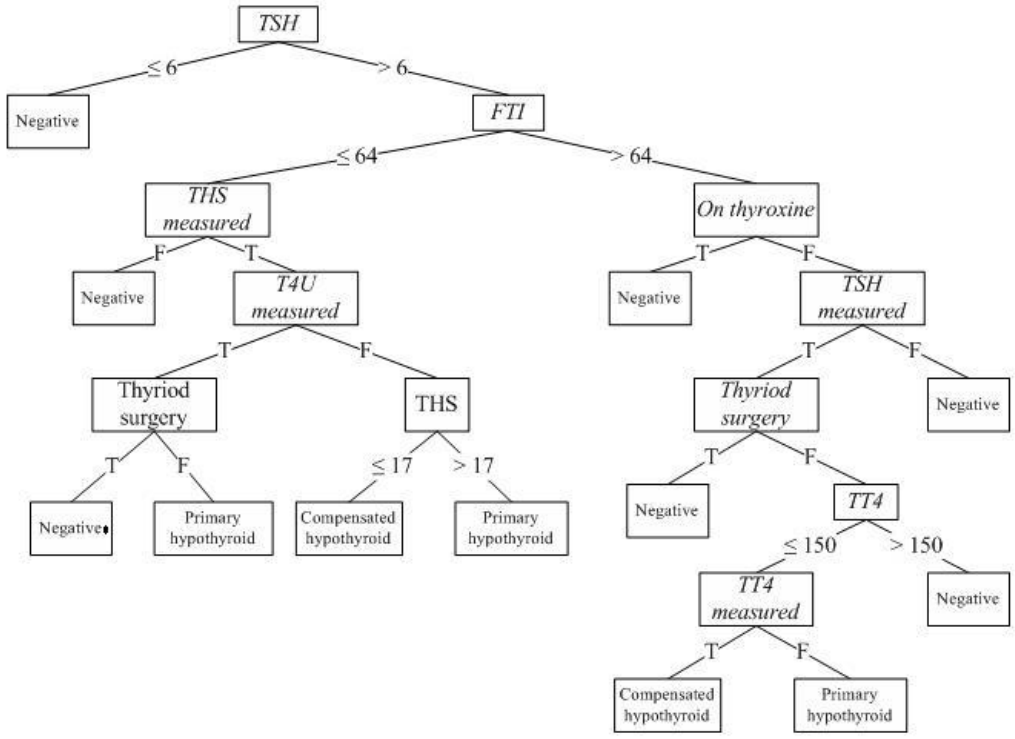


Figure 2.8: Decision tree for hypothyroid conditions.

Then, we consider a little more complicated example which is illustrated by Figure 2.8. Particularly, the leaf with black dot can produce an initial rule as below:

$$\begin{array}{cc} \textbf{if} & TSH > 6 \\ and & FTI \leq 64 \\ and & TSH\ measured = t \\ and & T4U\ measured = t \\ and & thyroid\ surgery = t \\ \textbf{then} & class\ negative \end{array}$$

which contains three training instances, and two of which are classified as negative, i.e. $Y_1 = 2$ and $E_1 = 1$. Then, the pessimistic error rate with default confidence level for this rule is $U_{25\%}(1.3) = 69\%$. We then delete each of the five conditions in this rule above and see what happens about the pessimistic error rate (as we have discussed before, it will theoretically turn to be smaller):

Condition Deleted	$Y_1 + Y_2$	$E_1 + E_2$	Pessimistic Error Rate
TSH>6	3	1	55%
$FTI \leq 64$	6	1	34%
TSH measured=t	2	1	69%
T4U measured=t	2	1	69%
thyroid surgery=t	3	59	97%

Table 2.12: Possible conditions to delete.

Table 2.8 shows that deleting condition $FTI \leq 64$ provides the lowest pessimistic error rate 34%. We follow the suggestion of Table 2.8 to delete this condition. Then, we recompute the new pessimistic error rate for each of remaining conditions and repeat this operation until the minimum rate is obtained. As a result, the final rule contains just on condition:

$$\begin{array}{cc} \textbf{if} & thyroid\ surgery = t \\ \textbf{then} & class\ negative \end{array}$$

This single condition rule covers 35 training instances with only one error and 7% pessimistic error rate.

2.2.4 Windowing

Due to the restriction of computer memory thirty years ago, Quinlan explored an indirect method called $window$ to grow trees from relative large data sets. Window, just as its name implies, provides a local view from some randomly selected data to generate a decision tree which can approximate the global distribution of the whole data set. Inevitably, the decision tree developed from small parts of the data set will give birth to misclassification. A selection of these $exceptions$ is then added to the initial window, and a second tree, built from the enlarged training set, is tested on the remaining instances. This procedure is repeated until the decision tree constructed from the latest window correctly classified all the training instances outside the window.

Let us reload the Hypothyroid data set we used before. As shown in Table 2.13 below, C4.5 treated the data set with windowing option. For the first iteration, the initial window contains 502 of 2514 training instances. It built a tree with 15 nodes, which misclassified 5 instances in the window and 15 instances outside the window. Then, we added these 15 instances into the window to retrain

the decision tree. However, it developed a new tree with 19 nodes and misclassified 37 instances in total, in which 29 are outside the window. It means that this step followed a wrong direction. Sequentially, we repeat this procedure to generate a third tree. The final window contains 546 instances (22% of the training set), and there are no error in the data set outside the window.

Iteration	Tree	Objects		Errors					
	nodes	window	remain	window	rate	remain	rate	total	rate
1	15	502	2012	5	1.0%	15	0.7%	20	0.8%
2	19	517	1997	8	1.5%	29	1.5%	37	1.5%
3	21	546	1968	8	1.5%	0	0.0%	8	0.3%

Table 2.13: Windowing method to generate trees using hypothyroid data.

Though the computer performance updates to be much more powerful than before, windowing method has some attractive strong points:

1. Sometimes, it performs successfully in reducing the time required to construct a decision tree by using only a small proportion of the training instances.
2. It can grow several alternative trees and then choose the one with the lowest predicted error rate.
3. It can create several trees and generate production rules from all of them, then construct a single rule classifier from all the available rules.

2.3 CART

The construction of *CART* (classification and regression trees) is best described in [17] and has become a common basic method for building statistical models from simple feature data. CART is powerful because it can deal with incomplete data, multiple types of features (floats, categorial sets), and the trees it produces often contain rules which are humanly readable.

The basic building algorithm starts with a set of feature vectors representing samples, at each stage all possible questions for all possible features are asked about the data finding out how the question splits the data. A measurement of impurity of each partitioning is made and the question that generates the least impure partitions is selected. This process is applied recursively on each sub-partition until some stop criteria is met (e.g. a minimal number of samples in a partition). When the complexity is too large, CART applies pruning on the original tree generated from the training set to make the decision tree easily understandable and improvable. So, in the following, we mainly introduce three aspects of splitting criterion, stopping rules and pruning.

2.3.1 Splitting rules

Different from ID3 and C4.5, CART exerts three major splitting rules in CART: the *Gini impurity*, the *twoing rule*, and the *linear combination splits*. In addition to these main splitting rules, CART users can define a number of other rules for their own analytical needs. CART uses the Gini impurity (also known as Gini diversity index) as its default splitting rule. Therefore, we mainly focus on Gini impurity, which is applied to binary trees building, and give a slight introduction for the other two splitting rules.

Gini impurity

In order to work in Gini, we need first start with *variance impurity* defined as below:

$$i(n) = p(\omega_1|n)p(\omega_2|n)$$

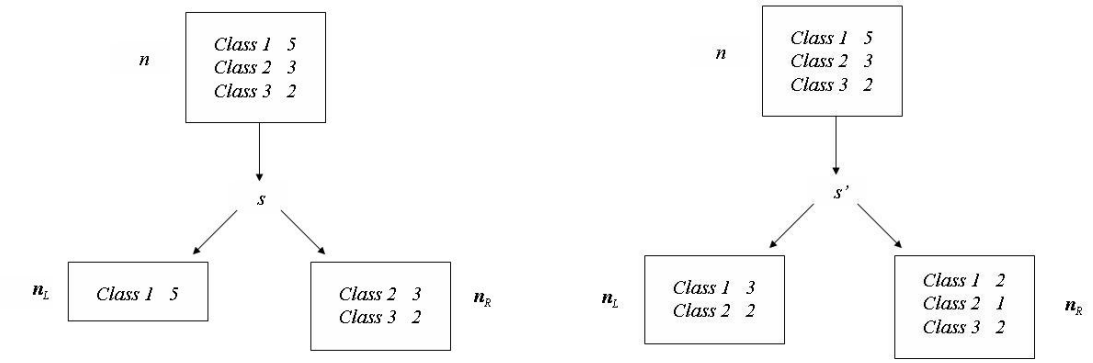
where $p(\omega_{1,2}|n)$ is the fraction of instances at node n in the class $\omega_{1,2}$. Apparently, if the node represents instances in the same class at the node n, the value of variance impurity is zero. The more general version of variance impurity, which can be applicable to two or more classes, is what we called *Gini impurity*:

$$i(n) = \sum_{i \neq j} p(\omega_i|n)p(\omega_j|n) = \frac{1}{2}[1 - \sum_k p^2(\omega_k|n)]$$

Similarly, Gini achieves its peak value when all of the instances at node n equally distribute in each class (e.g. $p(1|n) = p(2|n) = \cdots = p(N|n)$) and attains its minimum value zero when all of the instances flow to a single class.

Assumed that s is a split at node n, then, the *goodness of split* s is defined as the decrease in impurity measured by

$$\Delta i(s, n) = i(n) - p_L i(n_L) - p_R i(n_R)$$



(a) Split s of node n.

(b) Split s' of node n.

Figure 2.9: Examples of different splits.

where n_L and n_R are the left and right descendent nodes, $i(n_L)$ and $i(n_R)$ are their impurities, and p_L, p_R are the proportions of instances at node n that will go into the left child node n_L or the right child node n_R with a particular split s. For example, as shown in Figure 2.9

As we have seen in Figure 2.9(a), there are 10 instances at node n from class , class 2 and class 3. The proportions for each class are 50%, 30% and 20%. Thus, the Gini impurities before and after split s are:

$$
\begin{aligned}
i(n) &= 1 - (0.5^2 + 0.3^2 + 0.2^2) = 0.62 \\
i(n_L) &= 1 - 1^2 = 0 \\
i(n_R) &= 1 - (0.6^2 + 0.4^2) = 0.52
\end{aligned}
$$

Then, the *goodness of split*s is

$$\Delta i(s, n) = 0.62 - 0.5 \times 0 - 0.5 \times 0.52 = 0.36$$

In the same calculating procedure, we get $\Delta i(s', n) = 0.18$ in Figure 2.9(b), which is smaller than $\Delta i(s, n)$. Consequently, split s is a better choice compared with split s'.

In the real world task, splitting choices are numerous especially for the continuous-valued attributes. Hence, computing the *goodness of split*'s one by one will be much too expensive and even not possible. Therefore, the problem left is "How can we find an optimal decision for a node?" On one hand, if the form of the splits is based on categorial attributes, we may execute an extensive or even exhaustive search over all possible subsets of the training set to obtain the decision maximizing Δi. On the other hand, if the attributes are continuous-valued, we perform a step-wise attribute selection based thresholds defined as the middle points between the sorted values (as done in ID3 and CART).

Twoing rule

Sometimes, we come up against multi-class binary tree generation. Then, the *twoing rule* may be a better choice. The overall aim is to find the best splitting of classes. Denote $\mathcal{C}_1$ as a candidate

"superclass" which consists of all instances in some subset of the classes, and candidate "superclass" $\mathcal{C}_2$ containing all the rest instances. Given the set of classes $\mathcal{C} = \{\omega_1, \omega_2, \ldots, \omega_c\}$, the decision splits the classes into $\mathcal{C}_1 = \{\omega_{i_1}, \omega_{i_2}, \ldots, \omega_{i_k}\}$, and $\mathcal{C}_2 = \mathcal{C} - \mathcal{C}_1$ at each node (as shown in Figure 2.9). For every split s of the node n, we compute the *goodness of split* $\Delta i(s, n, \mathcal{C}_1)$ as though it corresponded to a standard two-class problem. Then, we can find the split $s^*(\mathcal{C}_1)$ that maximizes $\Delta i(s^*(\mathcal{C}_1), n, \mathcal{C}_1)$, and then find the superclass $\mathcal{C}_1^*$ that maximizes $\Delta i(s^*(\mathcal{C}_1), n, \mathcal{C}_1)$. The significant benefit of the *twoing rule* is that it gives "strategic" splits and informs us the similarities of the classes. At each node, it separate the classes into two groups which in some sense are most dissimilar and outputs to us the optimal grouping $\mathcal{C}_1^*$ and $\mathcal{C}_2^*$ as well as the best split s^*. The word "strategic" actually implies that on the top of the tree, the criterion attempts to collect large numbers of classes which are similar to each other in some characteristic. But near the bottom of the tree, it tries to isolate single classes. To illustrate this, given a four-class problem, originally classes 1 and 2 were grouped together and split off from classes 3 and 4, resulting in a node with instances as below.

Class:	1	2	3	4
Cases:	50	50	3	1

Then, the next split of this node will separate class 1 from class 2 due to the largest potential for decrease in impurity.

Although *twoing rule* seems to be useful with a large number of classes, the computational efficiency is a big limitation. For instance $\mathcal{C}$ with n classes, there are 2^{n-1} choices of divisions to put it into two superclasses, which is intolerable. Fortunately and surprisingly, Breiman has proofed in [17] that *twoing rule* can be reduced to an overall criterion, running at around the same efficiency as *Gini*.

Linear combinations

One alternative to CART's use for splitting is *linear combinations*. In some cases, the classes are spontaneously sorted by hyper-planes which are not perpendicular to the coordinate axes. These features will result in complicated tree generation, since the algorithm attempts to approximate the hyperplanes by rectangular regions (as the conditions in the nodes are from type $x_i < thr$). In order to deal with such problems, best splits over linear combinations of variables are required.

In linear combination algorithm, assumed there are N ordered variables $(x_1, ...x_N)$ (continuous valued-attributes only and no missing data), at each node n, we take a set of weights $\mathbf{a} = (a_1, ..., a_N)$ such that $\|\mathbf{a}\| = \sum_{i=1}^{N} a_i^2 = 1$, and search for the best split of the following form

$$\sum_{i=1}^{N} a_i x_i \leq C$$

where C takes all possible values. Then, the best set of weights $\mathbf{a}^*$ of this split is the one which maximizes the corresponding decrease in impurity $\Delta i(s^*(\mathbf{a}), n)$. That is,

$$\Delta i(s^*(\mathbf{a}^*), n) = \max_{\mathbf{a}} \Delta i(s^*(\mathbf{a}), n).$$

Although the rough method is easily formulated, the explanation of an effective search algorithm for maximizing $\Delta i(s^*(\mathbf{a}), n)$ over a large set of possible values of $\mathbf{a}$ is complicated. The details

are described in the appendix of [17]. However, like many other searching algorithm, the global extrema cannot be guaranteed, and local maxima may be brought in.

For example, assumed we have already come up with the best splitting strategy, which can be turned into the following question

$$is \quad 0.3\, campus\ life + 0.7\, career\ time \leq 10?$$

If the response to the question is "yes", then the case is sent to the left node, and if the response is "no", then the case is sent to the right node. This rule is valid only for cases with no missing values on predictor variables. Further more, if categorical variables have to be included in the model, they should be converted to sets of dummy variables.

However, this algorithm is also too complicated for high-dimensional data. At each node, one has to produce a split base on a linear combination of all ordered measurement variables, some of which contribute not so much to the generation of the split. Thus, we use a backward deletion process to simplify this structure.

Taking l from 1 to N, we vary the threshold constant C and find the best split with

$$\sum_{i \neq l} a_i^* x_i \leq C_l,$$

which means, we find the best split with the set of weights $\mathbf{a}$ but deleting x_l and optimizing on the threshold C. Denote the decrease in impurity of this splitting by Δ_l and $\Delta^* = \Delta i(s^*(a^*), n)$. The least important variable is the one for which Δ_l is a maximum, and minimum when the variable is the most important one. Thus, we can measure the effectiveness of deleting the least important variable by $\Delta^* - \max_l \Delta_l$, and $\Delta^* - \min_l \Delta_l$ for the most important one. Then, if

$$\Delta^* - \max_l \Delta_l < \beta(\Delta^* - \min_l \Delta_l),$$

where β is a constant (usually 0.1 or 0.2), we delete the least important variable.

Then, we repeat this procedure on the remaining variables, until no more deletion can be done. Denoting the indices of the undeleted variables by $\{n_l\}$, the algorithm search the best split of the form

$$\sum_{i \in \{n_l\}} a_i^{**} x_i \leq C_l^*,$$

with the restriction $\sum_{i \in \{n_l\}} (a_i^{**})^2 = 1$.

Compared with stepwise regression or discrimination, the tree deletion procedure is much more stringent in eliminating variables. However, lack of interpretability of the results is an unlucky disadvantage.

2.3.2 Stopping criterion

Now we turn to the question "when to stop the growing procedure?". To get the appropriate answer, two problems should be taken into consider. On one hand, if we split the tree on and on until it is fully generated and each leaf node corresponds to the lowest impurity, then the data will

be inevitably overfit. In the extreme situation, every leaf node contains only one instance, which is nothing but a "dictionary", particularly when the noise effects at a high level. On the other hand, if the splitting is finished too early, the data will not be trained sufficiently enough to get a good performance on classifying the data.

In order to have a balance solution of the problems mentioned above, we set out several stopping criterions:

Separating test set

One traditional approach of stop criterion is *validation* or *cross-validation*. In simple validation, we randomly separate the training set into two parts: one is used as a training set (e.g. 90%), and the remaining part is used as a validation set for error testing (e.g. 10%). Then, we continue the splitting procedure until the error on the validation data is minimized. A generalization of validation is *m-fold cross-validation*. The training set is randomly divided into m disjoint subsets of equal size n/m, where n is the total number of instances in the data set. The tree generation algorithm is used m times, each with a different validation set. The estimated performance is the mean of these m errors.

Practically, validation can be applied to every classification method, and this technique is heuristic and need not give improved classifiers in every case. However, it is extremely simple and is found to improve generalization accuracy for many real-world problems.

Setting a threshold (*goodness of split*, percentage of instances or instances of resulting nodes)

Secondly, we can set a small threshold for the reduction of impurities, as $\max_s \Delta i(s, n) \leq \varepsilon$. Splitting at a node can be stopped whenever the split fulfills this restriction. There are two advantages of this method. First, unlike cross-validation, it makes use of all training set for the tree generation. Second, leaf nodes appear in different depths of the tree, which is desirable whenever the complexity of the data varies throughout the range of input. A weak point of the method is that it is often difficult to choose the most suitable ε, since there is seldom a simple relationship we can follow between the threshold value and the final performance of the decision tree.

Another solution is to stop when the current node represents fewer amounts of instances than a threshold number, say 10 (as e.g. implemented in MATLAB's treefit method), or some fixed percentage of the total training data, say 5%.

Tree size (depth)

A third suggestion is control the complexity of the tree with a global criterion function

$$\alpha \cdot tree\ size + \sum_{n \in \{leaf\ nodes\}} i(n),$$

where $tree\ size$ could stand for the number of nodes or depth of the tree, and α is some positive constant. If an impurity based on entropy is employed, from *minimum description length* principle, we know that the tree generation can be stopped when the minimum of this function is reached. The $tree\ size$ is a measurement of the complexity of the classifier itself; and the sum of the impurities at leaf nodes is the measurement of the uncertainty in the training instances given the model represented by the decision tree. Similarly, one difficulty is setting a proper α, because it is

not easy to find a simple relationship between it and the classifier performance. This method will be explained more concrete in the next section.

2.3.3 Pruning

Large trees can have two problems:

1. Although they are highly accurate, with low or zero misclassification rates, large trees provide poor results when applied to new data sets.
2. Understanding and interpreting trees with a large number of terminal nodes is a complicated process.

Hence, large trees are referred to as complex trees. The complexity of a tree is measured by the number of its terminal nodes. Departures from the ideal situation of low or zero misclassification entails a trade-off between accuracy and tree complexity. For any subtree $T \preceq T_{\max}$, define its complexity as $|\widetilde{T}|$, the number of leaf nodes in T. The relationship between tree complexity and accuracy $R(T)$ (the tree misclassification costs) can be understood with the cost complexity measure, which is defined as

$$R_\alpha(T) = R(T) + \alpha|\widetilde{T}|,$$

where $\alpha \geq 0$ is penalty per additional leaf node. Then, for each value of α, find a subtree $T(\alpha) \preceq T_{\max}$ that minimizes $R_{alpha}(T)$, i.e.

$$R_\alpha(T(\alpha)) = \min_{T \preceq T_{\max}} R_\alpha(T).$$

If α is small, the penalty for having a large number of terminal nodes is also small, and thus $T(\alpha)$ will be large. In the extreme case when $\alpha = 0$, $T_{\max}$ is the largest tree that each leaf node contains only one instance, then every instance is classified correctly, which means $R(T_{\max}) = 0$. Therefore, $T_{\max}$ is the right tree which minimizes $R_0(T)$. On the other hand, as α increases and is sufficiently large (say infinity), a tree with one leaf node (the root node) will have the lowest cost complexity, and the tree is completely pruned. The "right-sized" tree with "correct" complexity should lie between these two extremes.

No matter what values (continuous or discrete) run for α, there are at most a finite number of subtrees of $T_{\max}$. Thus, the "right-sized" tree by pruning or collapsing is some subtree of T_{max} from the bottom up, which minimizes $R_\alpha(T)$. Define the smallest minimizing subtree $T(\alpha)$ that subjects to

i. $R_\alpha(T(\alpha)) = \min_{T \preceq T_{\max}} R_\alpha(T)$,

ii. if $R_\alpha(T) = R_\alpha(T(\alpha))$, then $T(\alpha) \preceq T$.

This definition breaks ties in minimal cost-complexity by selecting the smallest minimizer of R_α. [17] shows the uniqueness and existence of such a subtree.

Let n_L, n_R be any two terminal nodes in $T_{\max}$ from their immediate ancestor node n. From a proposition in [17] we know that $R(n) \leq R(n_L) + R(n_R)$. If the equality holds, then prune off n_L and n_R. Continue this procedure until no more pruning is possible. Then, the resulting tree

T_1 is the smallest subtree of $T_{\max}$, which satisfies $R(T_1) = R(T_{\max})$. For any branch T_n of T_1, define $R(T_n) = \sum_{n' \in \widetilde{T}_n} R(n')$, where $\widetilde{T}_n$ is the set of terminal nodes of T_n. Then, the following property holds: for any non-leaf t node of T_1, $R(n) > R(T_n)$. Denote $\{n\}$ the subbranch of T_n consisting of the single node n. Set $R_\alpha(\{n\}) = R(n) + \alpha$ and define $R_\alpha(T_n) = R(T_n) + \alpha|\widetilde{T}_n|$. Solving the next inequality

$$R_\alpha(T_n) < R_\alpha(\{n\}),$$

we have $\alpha < \frac{R(n)-R(T_n)}{|\widetilde{T}_n|-1}$. Then, define a function $g_1(n)$, $n \in T_1$, by

$$g_1(n) = \begin{cases} \frac{R(n)-R(T_n)}{|\widetilde{T}_n|-1}, & n \notin \widetilde{T}_1 \\ +\infty, & n \in \widetilde{T}_1 \end{cases}$$

Then define the $weakest\ link$ $\overline{n}_1$ in T_1 as the node such that $g_1(\overline{n}_1) = \min_{n \in T_1} g_1(n)$ and set $\alpha_2 = g_1(\overline{n}_1)$. The node $\overline{n}_1$ is the weakest link means that as α increases, it is the first node such that $R_\alpha(\{n\})$ turns equal to $R_\alpha(T_n)$. Hence, $\overline{n}_1$ becomes preferable to $T_{\overline{n}_1}$, and α_2 is the value of α when it approaches equality.

Define a new tree $T_2 \prec T_1$ by pruning away the branch $T_{\overline{n}_1}$, that is

$$T_2 = T_1 - T_{\overline{n}_1}.$$

Now, repeat this procedure by using T_2 instead of T_1 and find the weakest link in T_2. If at any step there is a multiplicity of weakest links, i.e. $g_k(T_{\overline{n}_k}) = g_k(T_{\overline{n}'_k})$, define

$$T_{k+1} = T_k - T_{\overline{n}_k} - T_{\overline{n}'_k}.$$

continuing this process, we get a decreasing sequence of subtrees

$$T_1 \succ T_2 \succ T_3 \cdots \succ n_1.$$

And the minimal cost-complexity pruning is given by the following theorem

> *The $\{\alpha_k\}$ are an increasing sequence, that is, $\alpha_k < \alpha_{k+1}$, $k \geq 1$, where $\alpha_1 = 0$.*
> *For $k \geq 1$, $\alpha_k \leq \alpha < \alpha_{k+1}$, $T(\alpha) = T(\alpha_k) = T_k$.*

As seen above, the pruning process produces a series of sequentially nested subtrees along with two types of misclassification costs and cost complexity parameter values. These are the cross validated relative error cost from applying ten fold cross validation and the resubstitution relative cost generated from the learning sample. Using the resubstitution cost, CART ranks the subtrees and generates a tree sequence ordered from the most complex tree to a least complex tree with one terminal node. In other words, the tree sequence provides subtrees with a decreasing complexity (a decreasing number of terminal nodes) and an increasing cost (resubstitution relative cost). CART finally identifies the minimum cost tree, and picks an optimal tree as the tree within one standard error of the minimum cost tree. The option of a one-standard-error rule can be changed by the data analysis. But the reason for using a one-standard-error rule is that there may be other trees with cross validated error rates close to those of the minimum cost tree. [17] suggests that an optimal

tree should be the one with the smallest terminal nodes among those that lie within one-standard-error of the minimum cost tree. The minimum cost tree itself could become the "right-sized" or the optimal cost tree.

To illustrate the effect, we take a look at an example ("fisher's iris") from Matlab toolbox. It trained a data set with 150 instances, each of which has 4 attributes and classified them into 3 classes.

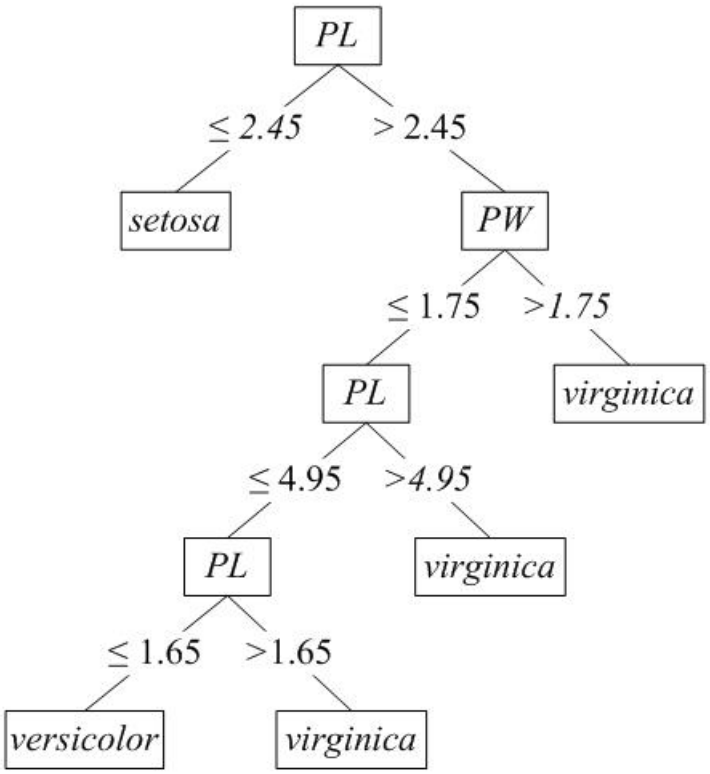


Figure 2.10: Decision tree for fisher's iris.

After training, we obtain a decision 4-depth tree with 5 leaf nodes. Although this is very simple tree, we want to actually find the best tree for the data set using cross-validation. Here we don't have another data set, but we can simulate one by doing cross-validation. We remove a subset of 10% of the data, build a tree using the other 90%, and use the tree to classify the removed 10%. We could repeat this by removing each of ten subsets one at a time. For each subset we may find that smaller trees give smaller error than the full tree. First we compute what is called the "re-substitution error," or the proportion of original observations that were misclassified by various subsets of the original tree. Then we use cross-validation to estimate the true error for trees of various sizes. As we see in Fig. 2.11, the solid line shows the estimated cost for each tree size by cross-validation, the dashed line illustrates the estimated cost for each tree size by re-substitution, and the dotted line marks 1 standard error above the minimum. This figure shows that the re-substitution error is overly optimistic. It decreases as the tree size grows, but the cross-validation results show that beyond a certain point, increasing the tree size increases the error rate. What shall we choose then? A simple rule would be to choose the tree with the smallest cross-validation error. While this may be satisfactory, we might prefer to use a simpler tree if it is roughly as good as a more complex tree. The rule we will use is to take the simplest tree that is within one standard error of the minimum. That's the default rule used by the "treetest" function in Matlab.

We can show this on the figure by computing a threshold value that is equal to the minimum cost plus one standard error. The "best" level computed by the "treetest" is the smallest tree under this threshold as shown in Fig. 2.11, the circle marks the smallest tree under the dashed line. Then, CART pruned the tree as shown in Fig. 2.12. As a result of testing, we find the best tree with only 3 leaf nodes and an acceptable standard error smaller than the threshold.

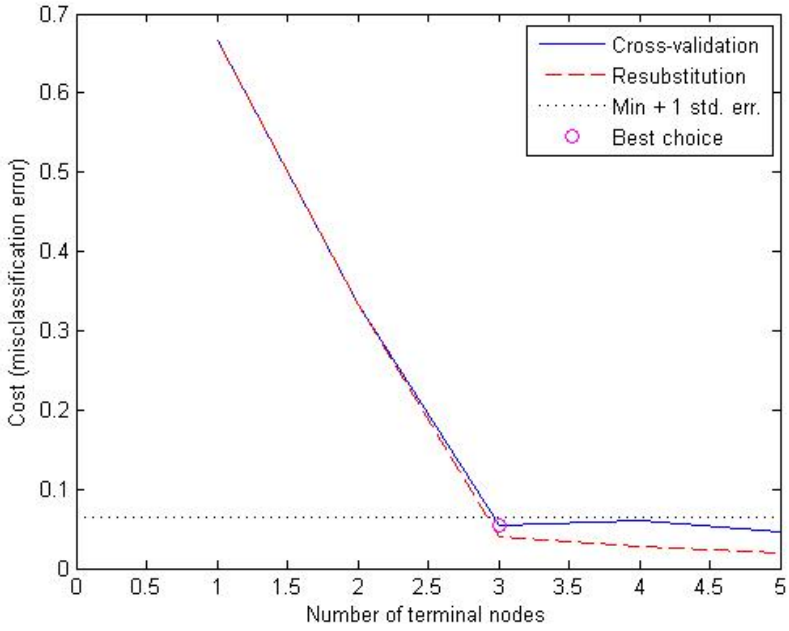


Figure 2.11: Smallest tree within 1 std. error of minimum cost tree.

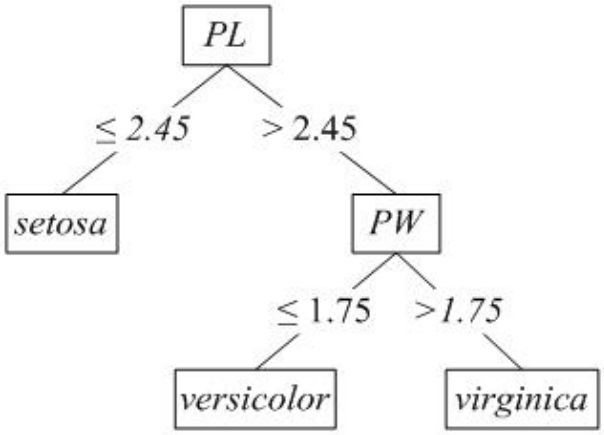


Figure 2.12: Decision tree for fisher's iris after pruning.

2.3.4 Outline

From these three subsections described above, we can finally conclude the procedure for CART as following:

1. Split the first variable at all of its possible split points (at all of the values the variable assumes in the sample). At each possible split point of a variable, the sample splits into binary or two child nodes. Cases with a "yes" response to the question posed are sent to the left node and those with "no" responses are sent to the right node.
2. Apply the goodness of split criterion to each split point and evaluates the reduction in impurity that is achieved using the formula

   $$\Delta i(s, n) = i(n) - p_L i(n_L) - p_R i(n_R),$$

   which was described earlier.
3. Select the best split of the variable as that split for which the reduction in impurity is highest.
4. Repeat steps 1-3 for each of the remaining variables at the root node.
5. Rank all of the best splits on each variable according to the reduction in impurity achieved by each split.
6. Select the variable and its split point that most reduced the impurity of the root or parent node.
7. Apply stopping criterion to new leaf nodes e.g. $\Delta i \leq \epsilon$. If fulfilled, assign that class label which has the most occurrence in the corresponding subset and make the node a terminal node, otherwise make the node a non-terminal child node. CART has a built-in algorithm that takes into account user defined variable misclassification costs during the splitting process. The default is unit or equal misclassification costs.
8. Apply steps 1-7 repeatedly to each non-terminal child node at each successive stage.
9. Stopping criterion is applied to the tree with skills as setting a threshold of goodness of split, or percentage of instances, or instance numbers of resulting nodes, controlling the complexity of the tree with a global criterion function defined as before

   $$R(T) + \alpha|\widetilde{T}|,$$

10. Prune the tree with the cost complexity measure to produce a series of sequentially nested subtrees with a decreasing complexity and an increasing cost. Then, choose the minimum cost tree as the "right-sized" one. This is the tree with minimal complexity lying within the 1-standard error band.

Chapter 3

Incremental Decision Trees

Incremental learning algorithms get sample per sample (as row-by-row in the data matrix above) as input. In this chapter, we only give a rough introduction of some typically incremental versions of ID3 and CART. Details for these algorithms can be found in works [22], [25], [23] and [26].

3.1 Incremental ID3

For non-incremental classification tasks, ID3 is often a good choice for training a decision tree and testing. However, for incremental tasks, accept instances incrementally, without creating a new decision tree at each time, is strongly desired. In this section, we will first review ID4 algorithm, which is designed to learn decision trees incrementally. Then, a new incremental tree-construction algorithm called ID5R, which builds the same decision tree as ID3 from a given training set, is presented. Finally, a powerful successor of ID5R called ITI will be introduced.

3.1.1 The ID4 algorithm

ID4 is originated to treat with data flows for addressing the incremental construction of decision trees. However, as what we will show later, there are many disadvantages of ID4, even compared with its off-line version ID3. The reasons why we still present such a low-efficiency algorithm are: First, in order to get a better understanding of the inability of ID4 to learn certain concepts, the algorithm should be explained in detail. Second, ID4 includes the basic mechanism of ID5R, which we will describe later.

Recall the notations in 2.1, we now consider a two-class problem. As shown below, ID4 accepts a training instance and then updates the decision tree. In this procedure, we have to keep the information, which is needed to compute the E-score for the possible test attribute, at each node. The information consists of (p, n) counts of each possible test attribute at each node in the current decision tree. Denote *non-test* attribute to be an attribute that have not been tested yet. With the (p, n) counts kept in each node, it is possible to compute the E-score of each attribute and pick up the lowest one. If the current test attribute does not have the lowest E-score, then is replaced by the non-test attribute with the lowest E-score. Since the counts have been maintained, re-examine the training data is not necessary.

The complete ID4 algorithm includes a χ^2 test for independence, as described in 2.1.4. The χ^2 test prevent over-fitting the training data when noise invades, which will replaces than answer node with a decision node. To include the test, step 3 of ID4 should be executed only if there is

an attribute that is also not χ^2 independent. Here we omit the χ^2 test, but only concentrate on the basic algorithm.

The following ID4 algorithm is executed, whenever new data set comes.

1. Input: training instances one by one
2. Start: current_node = root_node, call ID4(current_node)
3. For each possible test attribute at the current node, update the count of positive or negative instances for the value of that attribute in the training set.
4. If all the instances observed at the current node are positive(negative), then the decision tree is an answer node.
5. Otherwise, call function ID4(current_node)
   (a) If the current node is an answer node, then change it to a decision node containing an attribute test with the lowest E-score.
   (b) Otherwise, if the current decision node contains an attribute test that does not have the lowest E-score, then
      i. Change the attribute test to one with the lowest E-score.
      ii. Discard all existing subtrees below the decision node.
   (c) Recursively call function ID4(current_node) to update the decision tree below the current decision node along the branch of the value of the current test attribute. Grow the branch if necessary.
6. End function

ID4 can build the same tree as ID3 only when there is an attribute at each decision node that is indisputably the best choice in terms of its E-score. Whenever a non-test attribute replaces the test attribute at a node in training, step 3.(b)ii. discards all subtrees below the node. So, if the relative ordering does not stabilize with the training, the subtrees will be discarded repeatedly, rendering certain concepts un-learnable by the algorithm. One could also note, this algorithm is useless when we consider the continuous-valued attributes, since it is impossible to store the (p, n) counts for the infinite set of continuous-valued attributes.

3.1.2 The ID5R algorithm

Now, we present a new incremental algorithm ID5R, which can guarantee to generate the same decision tree as ID3 for a given training set. Unlike ID4, this algorithm can effectively apply the non-incremental method from ID3 to incremental tasks, without the unacceptable expense of generating a new tree after new training instances come. ID5R maintains sufficient information to calculate E-score for an attribute at a node as well, so that it can replace the test attribute to the one with the lowest E-score.

However, instead of discarding the subtrees below the original test attribute, ID5R restructures the tree, making the desired test attribute at the root. This process, named *pull-up*, is a tree manipulation that preserves consistency with the existing instances, and brings the implied attribute to the root node of the tree or subtree. The most significant advantage of restructuring the tree is that one can recompute the various (p,n) counts without re-examining the training instances.

We define an ID5R decision tree to be either of

1. a leaf node that contains
   (a) a class name, and
   (b) the set of instance descriptions at the node belonging to the class.
2. a non-leaf node that contains
   (a) a test attribute, with a branch to another decision tree for each possible value of the attribute, the (p,n) counts for each possible value of the attribute, and
   (b) the set of non-test attribute at the node, each with (p,n) counts for each possible value of the attribute.

The form of the decision tree defined above is different from those of both ID3 and ID4, since it preserves the training instances in each node of the tree. Thus, this tree must be analyzed in different way. When classifying an instance by the tree, it is traversed until a node containing same class instances is reached. Such a node can be either a leaf or non-leaf node. In the latter case, there should be only one class with a non-zero instance count. Then, discarding the information in the training data is unnecessary with this method for tree building. Denote the leaf node form of tree as an *unexpanded tree* and the non-leaf node form an *expanded* tree. The actual form of the tree could be the unexpanded one or an expanded one with one or more superfluous test attributes. So, the tree structure underlying a tree's interpretation may not be unique.

As described in the above definition, we use the attribute-value pairs to depict the instances at a node which have not been determined by tests above the node. Thus, it means that the descriptions of a instance is reduced to only one attribute-value pair for each attribute tested above in the tree. One may point out that record a tree in fully expanded form would be the simplest way. However, this is inefficient in both time and space since all counts for non test attributes must be maintained at each non-leaf node.

1. Input: training instances one by one
2. Start: current_node = root_node, call ID5R(current_node)
3. If the tree is empty, then define it as the unexpanded form, setting the class name to the class of the instance, and the set of instances to the singleton set containing the instance.
4. Otherwise, if the tree is in unexpanded form and the instance is from the same class, then add the instance to the set of instances kept in the node.
5. Otherwise, call function ID5R(current_node)
   (a) If the tree is in unexpanded form, then expand it one level, choosing the test attribute for the root arbitrarily.
   (b) For the test attribute and all non-test attributes at the current node, update the count of positive or negative instances for the value of that attribute in the training instance.
   (c) Otherwise, if the current decision node contains an attribute test that does not have the lowest E-score, then
      i. Restructure the tree so that an attribute with the lowest E-score is at the root.
      ii. Recursively reestablish a best test attribute in each subtree except the one that will be updated in step 5d.

(d) Recursively call function ID5R(current_node) to update the decision tree below the current decision node along the branch of the value of the current test attribute. Grow the branch if necessary.

6. End function

As shown above, ID5R algorithm describes in details how it updates a decision tree. If the tree is unexpanded and a same class instance comes, then it is added to the set kept in the node. Otherwise, the tree should be expanded one more level, and the attribute-valued counts are updated for both test and non-test attributes according to this instance. The tree is then restructured with the lowest E-score attribute, and thus this procedure recursively checks the tree and its subtree until every test attribute at the node has the lowest E-score. ID5R creates the same decision tree as ID3, given the same training data and the same method for choosing attributes among the equally good E-score.

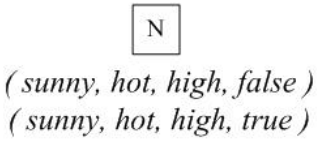
To restructure the decision tree, we use *pull-up* algorithm listed in the following to uplift the desired test attribute to the root. An *immediate subtree* of a node is a subtree that is rooted at child of the node. We reorder the attributes tests for level zero and level one to transpose a tree at the root, and then regroup level two subtrees. Since the attribute-value count at the root (level one) is already recorded from the tree update that led to the pull-up, and the trees at level two are not yet touched. We need to compute the counts for the level one test and non-test attributes. Then, we continue this update procedure recursively to the next level.

1. If the attribute a_{new} to be pulled up is already at the root, then stop.
2. Otherwise,
   (a) Recursively pull the attribute a_{new} to the root of each immediate subtree. Convert any unexpanded form as necessary, choosing attribute a_{new} as the test attribute.
   (b) Transpose the tree, resulting in a new tree with a_{new} at the root, and the old root attribute a_{old} at the root of each immediate subtree.

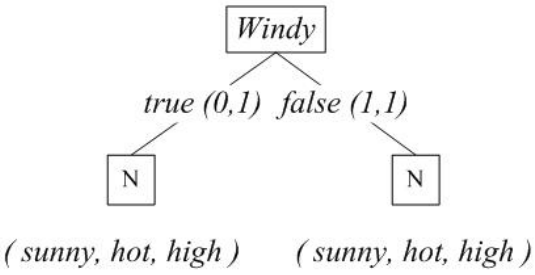
To illustrate how it works, we recall the case in Table 2.1. Assumed that the instance comes one by one as a data stream, then for the first instance (N, sunny, hot, high, false), the tree is updated simply be a leaf node

(sunny, hot, high, false)

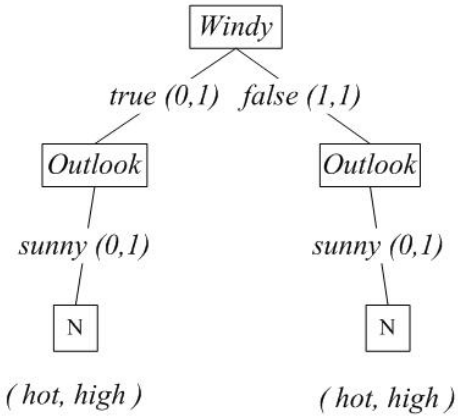
The second instance (N, sunny, hot, high, true) is also negative, so it is added to the same leaf node as, giving



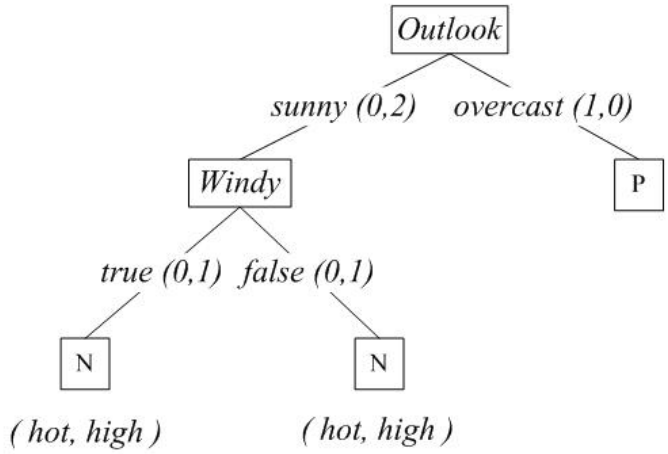
The third instance (P, overcast, hot, high, false) is positive, so the tree should be expanded. We arbitrarily choose attribute "Windy" as the test attribute. The (P, N) counts are updated at level zero, and the non-test attribute "Outlook" is found to have the lowest E-score. At this moment, the tree is



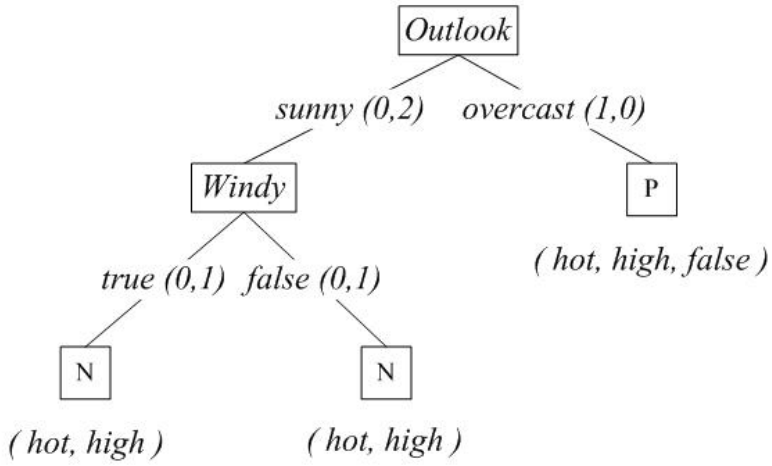
So the attribute "Outlook" should be pulled to the root. Note that the tree is in the process of being updated, so level zero is based on three training instances while level one and below are based on two training instances. When the update process is finished, the complete tree would be based on three training instances as well. First, we restructure the immediate subtrees so that "Outlook" is the test attribute for each one. for the current tree, the subtrees are unexpanded, so expand them with the test attribute "Outlook". At this point, the tree presents



But then, the tree is transposed to

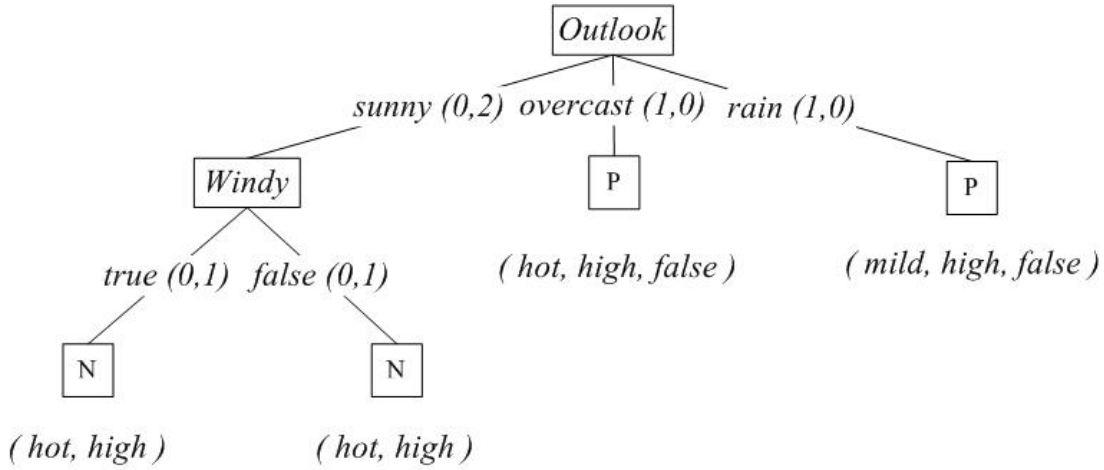


The branch of attribute "Outlook" in value appears, because all possible instance counts at level zero have already been updated (level one not). Note that the subtree below *Outlook=overcast* is empty, since the rest of the training instance has not been processed. Now the algorithm needs to ensure that a best test attribute is chosen at the root of each immediate subtree, except the one below *overcast*. The current test attribute "Windy" at the root of the subtree *Outlook=sunny* is due to transposition, but not have the lowest E-score. The subtree will be created when level one is updated. It happens that no tree restructuring is needed. the rest of the instance is processed recursively at level one, resulting in

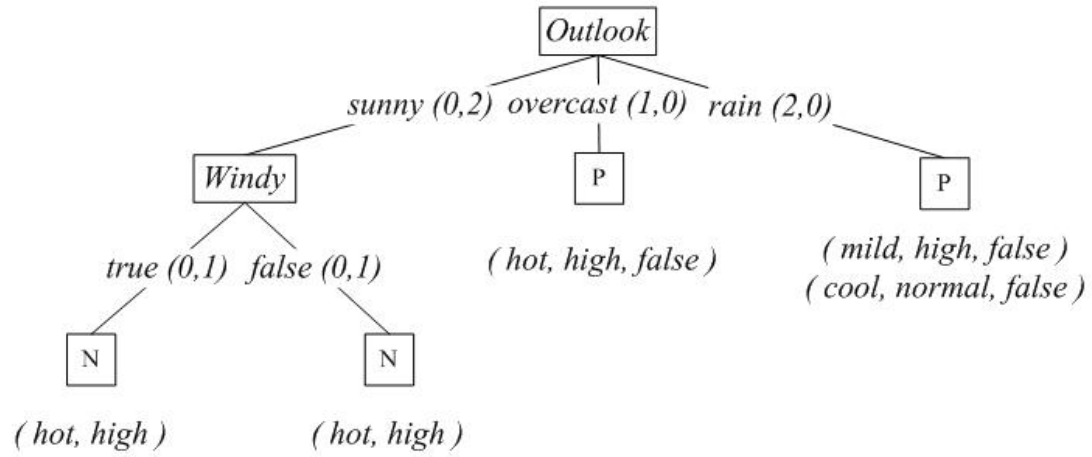


Note that the left subtree with "Windy" at its root could be contracted to unexpanded form, since all the instances (actually two) below comes from the same class. However ID5R algorithm does not include a contraction method, because it is not known when such a contraction is worthwhile. Though unexpanded tree is cheaper to update and needs less space, it is expensive in expanding a tree to compute its (P, N) count. The previous experience shows us that contraction to unexpanded form is generally not worthwhile.

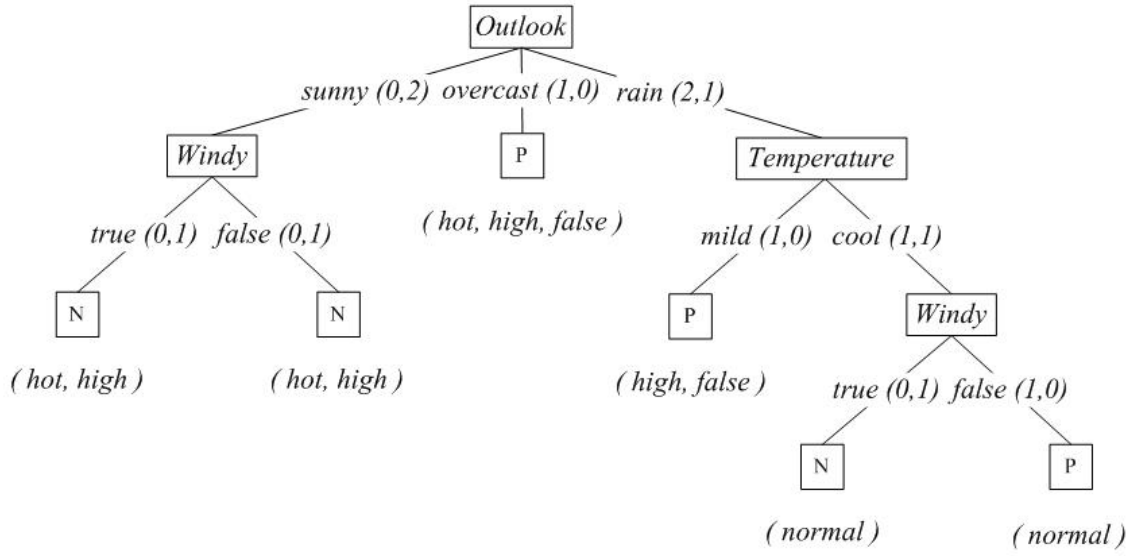
The fourth instance (P, rain, mild, high, false) leads to grow a branch for the level-zero tree, since a new value "rain" of attribute "Outlook" comes about. It happens again that no tree restructuring is needed ("Outlook" and "Windy" are the best choice for the root of its subtree all the same). Then, the tree presents



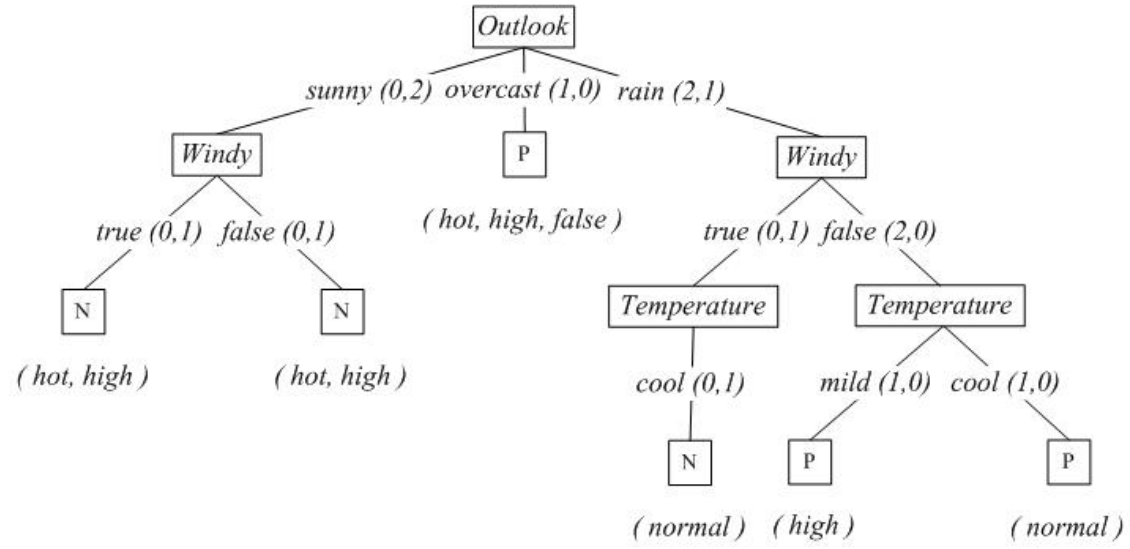
Because the right subtree *Outlook=rain* is in unexpanded form and the fifth instance (P, rain, cool, normal, false) comes from the same class, we just add it into the leaf-node and update the (P, N) counts, giving



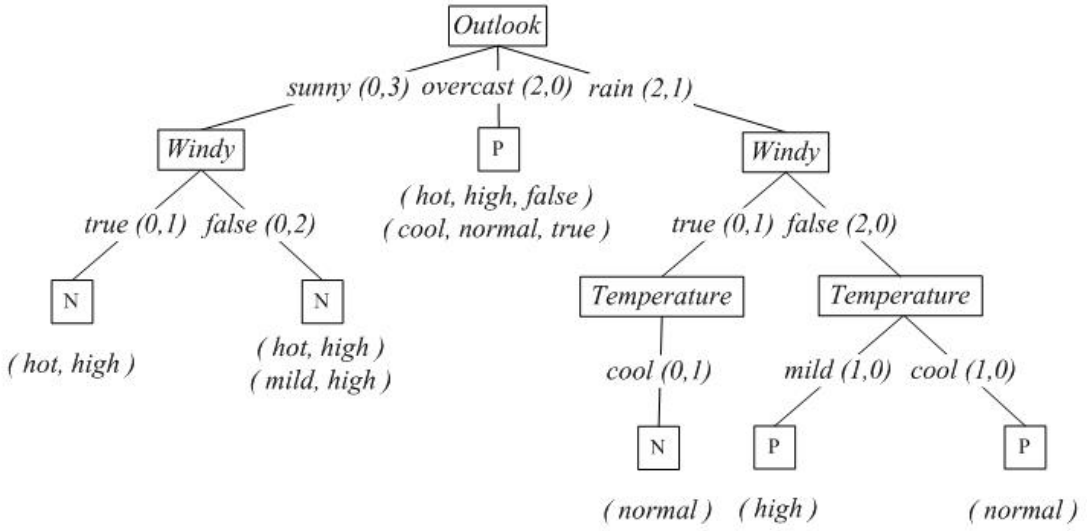
The sixth instance (N, rain, cool, normal, true) causes an expansion of the right level-one subtree and selection of "Temperature" arbitrarily as the test attribute, and "Windy" as the test attribute for the level-two subtree *Temperature=cool*, giving



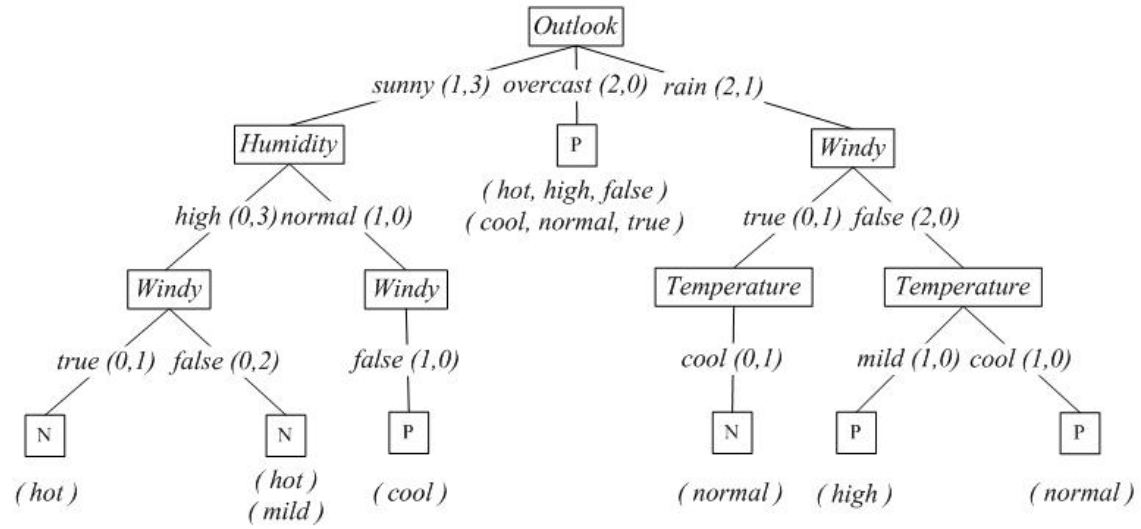
Since the "Temperature" does not have the lowest E-score and "Windy" is found to be the best choice, we pull-up "Windy" to the node of branch *Outlook=rain*. Then, the tree is transposed to be



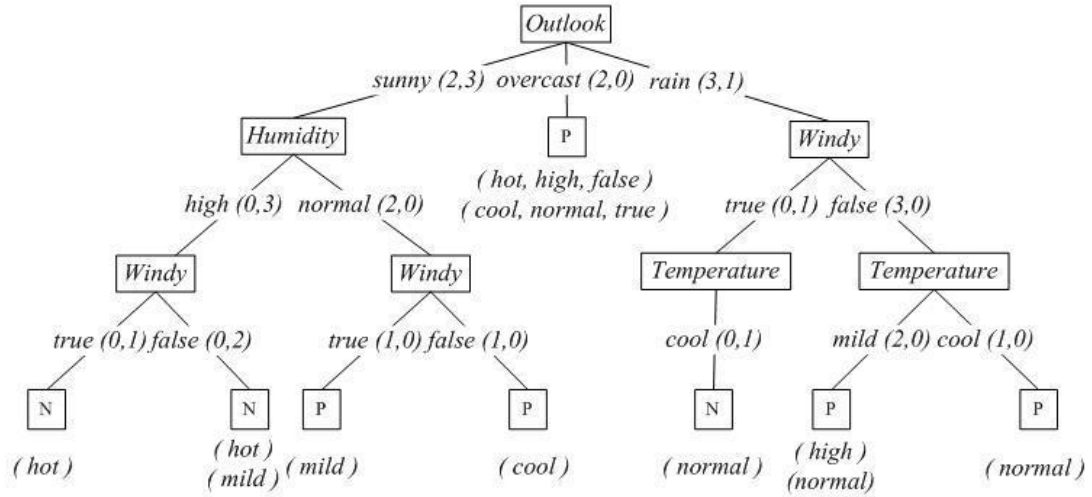
The seventh and eighth instances updates only the various count but do not cause any tree revision. After the eighth instance, the tree is



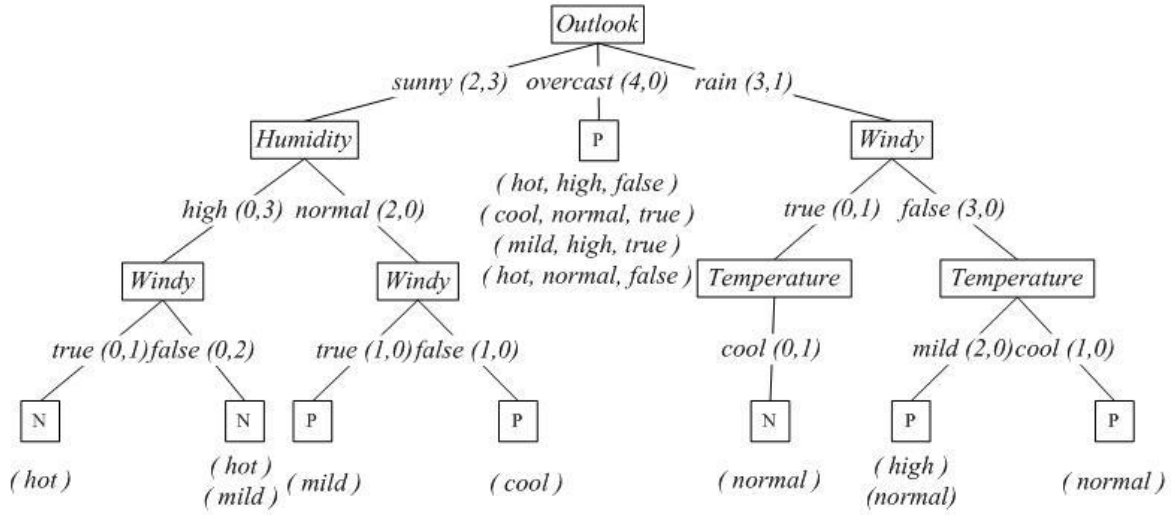
The ninth instance (P, sunny, cool, normal, false) causes the test attribute "Windy" of the left subtree rising in E-score. Thus, "Humidity" with the lowest E-score should be pulled-up to the root of the subtree, resulting in



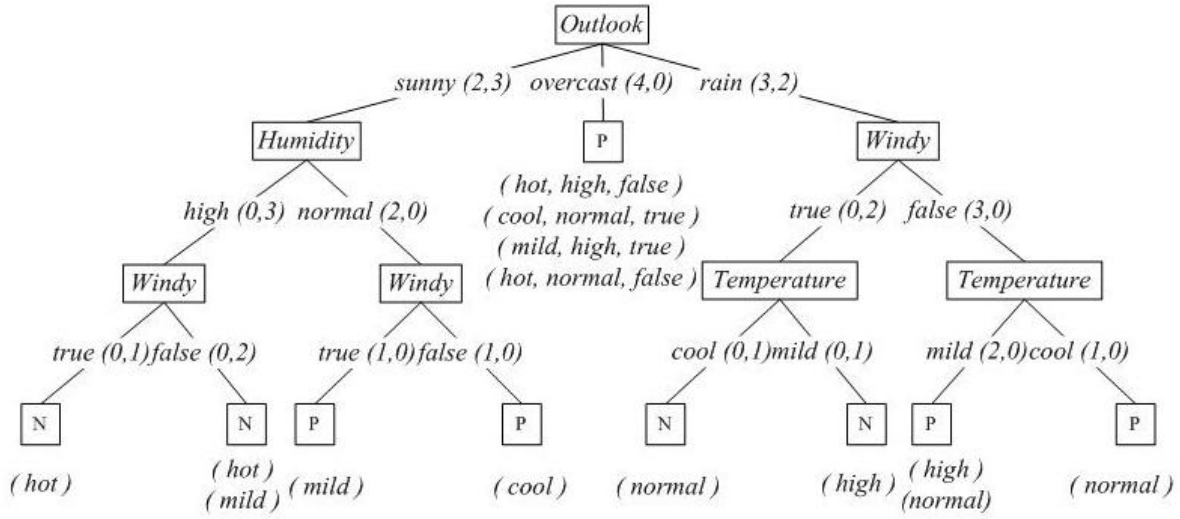
The tenth instance does not make any change to the tree, and the eleventh instance adds a branch below *Humidity=normal*. After the eleventh instance, the tree presents



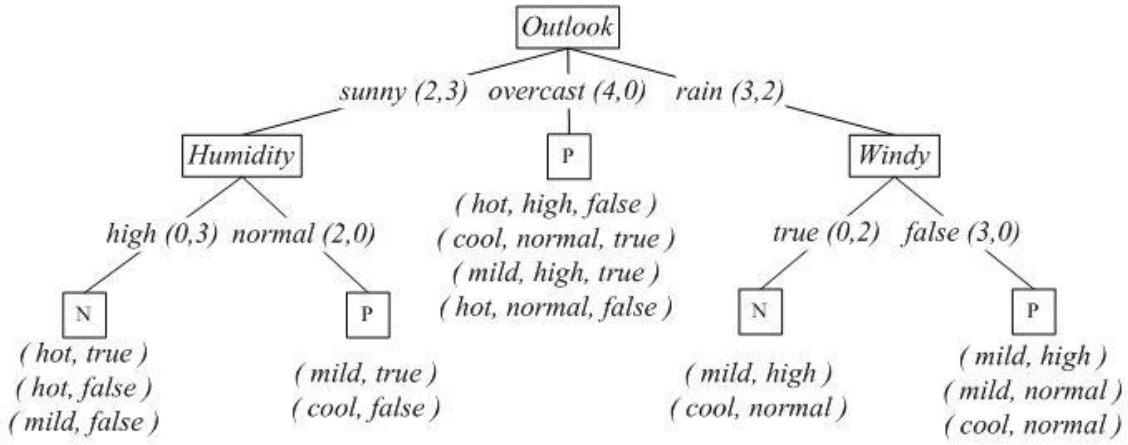
The twelfth and thirteenth instances come from the same class with previous ones *Outlook=overcast*, and thus are added to the leaf-node directly.



The last instance adds a new branch below the subtree *Windy=true*, and nothing else changes for the tree



If the tree could be contracted to the unexpanded form where possible, the result would be



However, this algorithm does not perform this step. As shown above, the final tree is equivalent to the one that the basic ID3 algorithm would generate for all these fourteen training instances (recall Fig. 2.1).

3.1.3 The ITI algorithm

Due to many significant disadvantages, ID5R has been now superseded by ITI (short for *Incremental Tree Induction*) [23], which makes a big improvement and can handle much more complex problems as below:

1. Accept instances described by categorical attributes or numeric attributes, or mixed of both.
2. Treat instances with multiple classes, not only two.
3. Not in favor of a attribute because it has a larger value set.
4. Deal with inconsistent training instances and missing values.
5. Avoid overfitting noise in the instances.
6. Execute efficiently in both time and space.

While ID5R algorithm can only accept categorical attributes described in previous section, ITI encodes numeric ones using a threshold in the manner similar to C4.5 (i.e., for two adjacent points, choosing "$corresponding\ attribute\_value < midpoint$" as the split). Differen from ID5R, ITI uses Gain Ratio as the selection criteria, which is also implemented exactly as described for C4.5.

ITI assumes that every possible test at a decision node has a binary outcome. For numeric attributes, the threshold automatically splits the node into two branches without any problem. For categorical attributes, especially non-binary ones, ITI maps them to an equivalent set of propositional attributes. For example, the attribute "Outlook={sunny, rain, windy}" is converted to "Outlook=sunny", "Outlook=rain" and "Outlook=windy". This adoption has two benefits: firstly, there will be no bias among the tests which have a large number of outcomes. Secondly, binary split is more conservative for partitioning, since one can keep a larger number of instances available in each branch.

Inconsistency means that if two instances are described by the same attribute values but have different class labels. When these kind of instances occur, ITI simply send both of them into the same leaf and keep it as an impure one. This cause no trouble for classification since the tree classifies unseen data as the majority class of the instances at that leaf. A missing value is treated as a special value which does not satisfy the test at a decision node. For example, if the test is "Outlook=sunny", then a value "sunny" satisfies the test, and neither the value "windy" or "?" satisfies it. So, with binary tests, all instances with missing values will be sent down to the false branch. This approach can simplify the computation of the splitting selection.

ITI employs MDL (*Minimum Description Length principle*) for pruning, which can effectively avoid overfitting the training data, especially when the data are known to contain attribute or classification errors. For more details, see [23] and [24].

At each decision node, ITI sustains a list of possible tests that could be used as the test at the node. For each binary test based on a categorical attribute, a table of frequency counts for each class and value combination is maintained. For each numeric attribute, a sorted list of the value seen, tagged with its class, is maintained with the attribute, and along with its best threshold.

3.2 Incremental CART

In the previous chapter, we introduced the CART algorithm with only batch cases. Now, we will shortly present a revision of CART[26], which can extend this approach to incremental instances flow learning. It treat newly acquired cases appropriately to update the existing tree as below:

1. When a new set of instances comes, recompute $\Delta i(s,n)$ (defined as $\Delta i(s,n)_{new}$) with the update data set at each existing node for the optimal split s_{opt}.
2. A hypothetical choice of split s_{hyp}, may send the new data set to a different descendant other than s_{opt} currently does. Then, the goodness of impurity for s_{hyp} is computed as $\Delta i(s,n)_{hyp}$.
3. (a) If $\Delta i(s,n)_{new} \geq \Delta i(s,n)_{hyp}$, then sent the new set of instances to proper descendant.
   (b) If $\Delta i(s,n)_{new} < \Delta i(s,n)_{hyp}$, and
      i. if $s_{opt} = s_{hyp}$, then sent the new set of instances to proper descendant.
      ii. If $s_{opt} \neq s_{hyp}$, then prune away the existing descendants of n and re-partition the new instances according to s_{hyp}. Run CART on each new descendant.

The problem with the initial algorithm is that there is no method to ascertain when s_{hyp} is sufficiently different from s_{opt} to warrant recalling CART, and when s_{opt}, even though slightly differ from s_{hyp}, could continue to be used. This deficiency induces a slight modified version of incremental CART described as below.

1. When a new set of instances comes, recompute $\Delta i(s,n)$ (defined as $\Delta i(s,n)_{new}$) with the update data set at each existing node for the optimal split s_{opt}.
2. A hypothetical choice of split s_{hyp}, may send the new data set to a different descendant other than s_{opt} currently does. Then, the goodness of impurity for s_{hyp} is computed as $\Delta i(s,n)_{hyp}$.
3. (a) If $\Delta i(s,n)_{new} \geq \Delta i(s,n)_{hyp}$, then sent the new set of instances to proper descendant.
   (b) If $\Delta i(s,n)_{new} < \Delta i(s,n)_{hyp}$, and
      i. if $s_{opt} = s_{hyp}$, then sent the new set of instances to proper descendant.
      ii. if $s_{opt} \neq s_{hyp}$, and
         A. if no neighborhood of "acceptable splits" exists, then invoke bootstrapping procedure (see details in [26]) to generate one.
         B. if s_{hyp} falls within the neighborhood, the s_{opt} is used as before.
         C. otherwise, prune away the existing descendants of n and re-partition the new instances according to s_{hyp}. Run CART on each new descendant.

Chapter 4

Evaluation

Here an empirical comparison between the different batch methods and the incremental methods. We shall first take a close look at the performance of ID3, C4.5 and CART with different parameters settings in the learning procedure (e.g. different prune levels, different selection resp. stopping criteria and so on) . Then, we will see that the incremental version of CART can come close to the batch version of it with respect to classification accuracy. For instance, if the batch learning version of CART (standard CART) produces a classification accuracy of 95% and the incremental version 93%, it is a quite good incremental approach (when taking into account that it is usable during on-line learning and parameter adaptation tasks). Furthermore, it should be shown that the incremental version is not much slower than the batch version and hence applicable for on-line training tasks, where operator's feedback should be dynamically incorporated etc. The tests will be carried out in the three following subsections by using two standard classification data sets from UCI repository and also by real-world data sets (e.g. image data from Sony). The structure of the empirical comparison is listed as below:

- compare ID3, C4.5 and CART with respect to classification accuracy elicited

  1.) with a CV procedure on the training data sets

  2.) with a separate test data set

- compare ID3, C4.5 and CART with respect to the complexity of the tree generated (into the same tables with the accuracy listed), e.g. by number of nodes, tree-depth etc.
- compare performance of different selection methods, different stopping criteria, and also different improved approaches like windowing, rules and pruning
- different costs for the attributes in CART, also by making some values in the data matrices unknown
- compare batch trained version of decision trees with incremental ones, by re-training, by keeping it static after loading the first dozens/hundreds of samples into memory and performing an initial training with the batch approaches
- compare the computational speed of incremental and batch trained trees

4.1 Evaluation on "Image Segmentation"

Firstly, we exploit "Image Segmentation" from Vision Group, University of Massachusetts to make a comparison between ID3, C4.5, CART and their incremental versions, and see how they perform in decision learning procedure. The instances were drawn randomly from a database of 7 outdoor images. The images were hand segmented to create a classification for every pixel. Each instance is a 3x3 region. The data set includes 210 training instances and 2100 testing ones, each of which has 19 continuous attributes with no missing value. There are 30 instances per class in the training data and 300 instances per class in the testing data.

Classification Approach	Tree Complexity		Training Accuracy(210)		Testing Accuracy(2100)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
ID3	29	7	5	2.4%	222	10.6%
C4.5	25	7	6	2.9%	189	9.0%
CART	27	8	5	2.4%	204	9.7%

Table 4.1: Comparison of trees generated by different classification approaches.

As shown in the table above, these three approaches performed more or less the same in both complexities and error rates on training data. However, C4.5 is slightly better than CART and much better than ID3. As we know, the full binary tree with 31 nodes has only 4 depth and 7 depth full binary tree has 255 nodes. We note that the decision trees obtained here are very unbalanced, since different instances bias to attributes related to their property in both structure and colors. It can be easily understood that the images like "sky" and "brickface" distributed very dense due to their more continuous structure and well-proportioned colors than "window" and "cement".

Classification Approach	Tree Complexity		Training Accuracy(210)		Testing Accuracy(2100)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
ID3 with Gain	25	6	6	2.9%	221	10.5%
C4.5 with Gain	25	6	6	2.9%	221	10.5%
CART with Twoing	29	6	5	2.4%	247	11.8%
CART with Deviance	27	7	4	1.9%	199	9.5%

Table 4.2: Comparison of different selection methods.

The default selection criterion for ID3 and C4.5 is gain ratio. If we choose gain as a selection method, the tree trained by these two approaches are totally the same, since there is no prune employed at all. Compared with Table 4.1, ID3 with Gain decreases the complexity of the tree but with almost the same accuracy, while C4.5 with Gain performed much worse in the testing side. In Table 4.2 we can also see, CART with "Deviance" ("Gini impurity" as default) acts more accurately than all the other approaches in both training and testing aspects.

Then, we apply different stopping criteria to the approaches. Recalling Table 4.1, a default minimum number of instances is set, 2 for ID3 and C4.5, and 10 for CART. Here, we fix the "splitmin" to be the same value of 5 with no unfairness, to see how they perform. As a result, ID3 and C4.5 generated a simpler same tree but with more errors. On the contrary, as we expected, CART created a more complex tree but with increased accuracy. Therefore, we can conclude that at the same "splitting level", CART takes more effort to train a more elaborate tree with a lower error rate than the other two approaches.

Classification Approach	Tree Complexity		Training Accuracy(210)		Testing Accuracy(2100)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
ID3 (splitmin=5)	21	6	12	5.7%	225	10.7%
C4.5 (splitmin=5)	21	6	12	5.7%	225	10.7%
CART (splitmin=5)	31	9	3	1.4%	200	9.5%
CART (cross-validation)	27	8	27	12.9%	110	5.3%

Table 4.3: Comparison of different stopping criteria.

For the cross-validation, CART partitions the data set into 10 subset, chosen randomly but with roughly equal size at each time. We executed the program 100 times and an average error rate is recorded. As the above table shown, although CART trained a very low accurate tree than the others, the testing result is extraordinary excellent(only 110 errors out of 2100 instances).

Classification Approach	Tree Complexity		Training Accuracy(210)		Testing Accuracy(2100)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
C4.5 Windowing	27	6	3	1.4%	233	11.1%
C4.5 Rules	12.2	-	4.3	2.0%	204.1	9.7%
CART Pruned	23	8	7	3.3%	195	9.3%

Table 4.4: Comparison of different improved approaches.

The C4.5 program can generate trees in two ways: batch mode (the default) or iterative mode. In iterative mode which is called "Windowing", the program starts with a randomly-selected subset of the data (the window). Then, it generates a trial decision tree, and adds some misclassified objects. This procedure continues until the trial decision tree correctly classifies all objects not in the window or until it appears that no progress is being made. Since iterative mode starts with a randomly-selected subset, multiple trials with the same data can be used to generate more than one tree. Here, 10 trials are carried out and the best tree from trial 6 is saved.

Another special execution for C4.5 is "Rules". It reads the decision tree or trees produced by C4.5 first and then generates a set of production rules from each tree and from all trees together. For each tree that it finds, the program generates a set of pruned rules, and then examines this set to find the most useful subset of it. If more than one tree was found, all subsets are then merged and the resulting composite set of rules is then examined. Here, C4.5 Rules got an average result on 11 trails. 12.2 means it betakes 12.2 rules on average for decision making and testing.

As we have mentioned in the last part of Chapter 2, the prune program of CART computes all error rate of each pruning level, and accepts the one with 1 standard error smaller than the pri-set threshold.

In Table 4.4, we can see that "C4.5 Windowing" generated a slight simpler tree with lower error rate than the normal procedure. However, the testing result is not good enough. "C4.5 Rules" seems to be more effective in reducing both complexity of generated tree and error rate of training and testing. As Fig. 4.1 illustrated, CART deleted two branches with 4 leaf nodes of the original tree, which increased the training error rate but reduced the testing one.

Here, we carry out robustness tests on these three approaches to see their performance in the cases of the noisy training data and the scarcity of training cases. Since all of the attributes value are continuous, we just make deviations from original values. For example, P% noise level means that

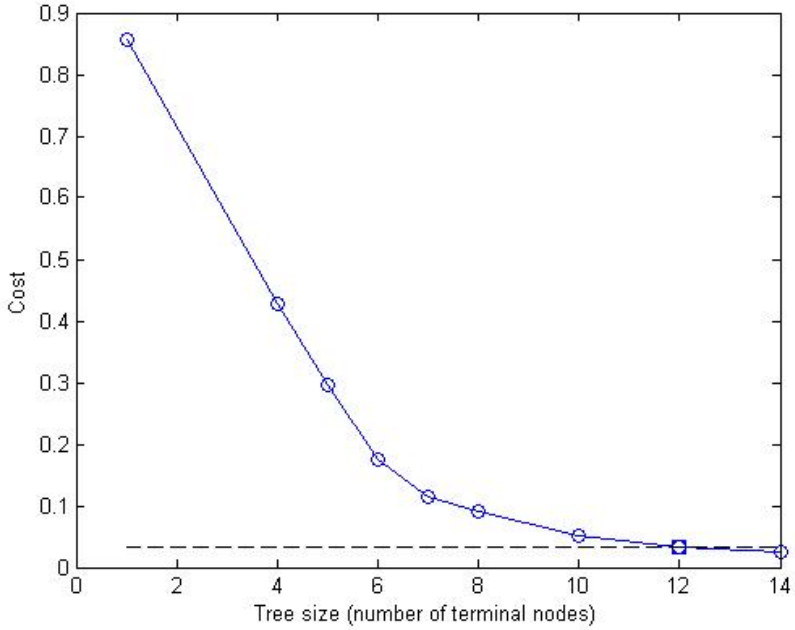


Figure 4.1: Smallest tree within 1 std. error of minimum cost tree for "Image Segmentation".

Error Rate	ID3		C4.5		CART	
Noisy level	Training	Testing	Training	Testing	Training	Testing
0	2.4%	10.6%	2.9%	9.0%	0	10.4%
5%	14.3%	17.5%	12.9%	14.3%	11.0%	13.7%
10%	22.4%	24.2%	20.5%	20.5%	19.0%	19.4%
15%	33.9%	34.1%	30.5%	29.8%	27.1%	28.0%
20%	42.4%	46.2%	41.5%	44.3%	34.3%	39.4%
25%	50.5%	52.6%	51.5%	52.3%	39.5%	43.2%
30%	52.4%	54.1%	54.3%	55.6%	39.5%	43.3%
40%	58.1%	56.9%	60.1%	60.3%	42.9%	44.2%
50%	62.0%	61.3%	63.9%	65.2%	51.0%	49.8%
60%	65.8%	65.7%	66.7%	70.4%	54.8%	58.0%
70%	72.9%	69.9%	75.3%	75.1%	57.6%	60.6%
80%	75.3%	71.9%	77.7%	77.3%	59.5%	62.9%
90%	77.7%	77.4%	81.0%	83.1%	59.5%	63.6%
100%	81.0%	80.1%	84.8%	85.6%	62.9%	65.6%

Table 4.5: Tree stability of different noise levels.

we multiply the input matrix with a factor $1 \pm P\%$ such that every attribute value has a disturbance.

The training and classification results are shown in Table 4.5, respectively. As we see in the table, 5% noise level produces more then ten percentage degradation in each of these three approaches for the data training (testing the original training data with the tree generated from noise added data set), while classification performance degrades from 3.3% to 6.9% for each. Apparently, CART acts much more stable than ID3 and C4.5: when the noise data increase to 100%, both of

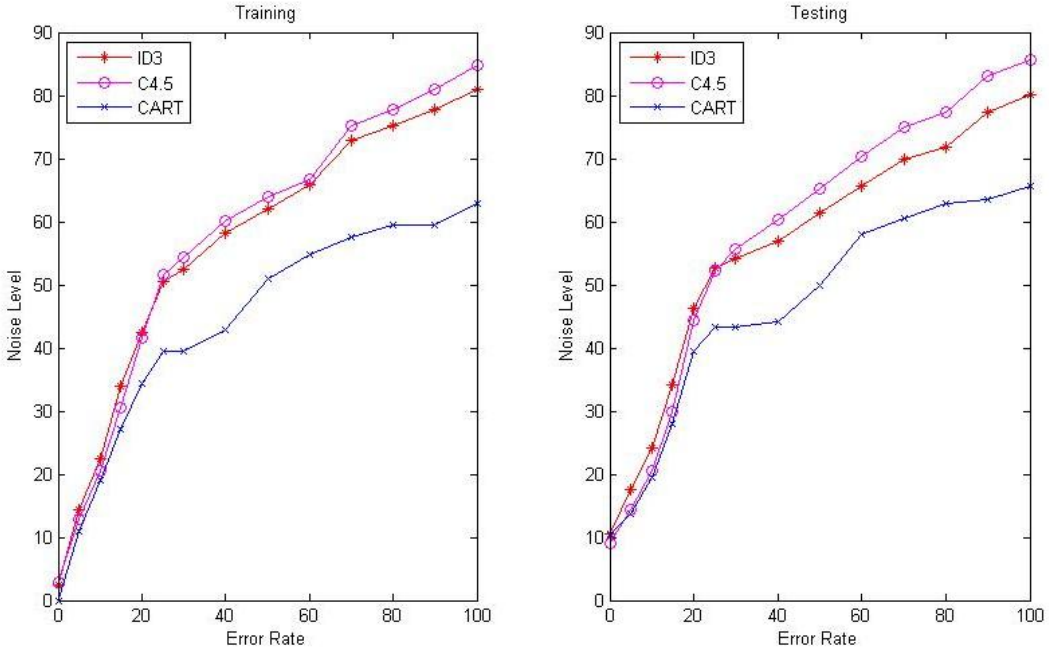


Figure 4.2: Classification performance of different noise level.

training and testing results of ID3 and C4.5 behave as randomly selecting a image from a uniform distribution data set. As we have mentioned in the very beginning of this section, there are 7 image for both training and testing, and each type of image has the same data size. So, the probability of choosing one from 7 approximately has the error rate 85.7%, which is very close to the result we have seen above. However, even with 100% noise level, CART could remain one third chance for classifying the right one.

Classification Approach	Tree Complexity		Training Accuracy(210)		Testing Accuracy(2100)
Algorithm	Nodes	Depth	Error Rate	CPU Times	Error Rate
ID3	29	7	2.4%	0.01	10.6%
C4.5	25	7	2.9%	0.01	9.0%
CART	41	9	0	0.34	10.4%
ITI-1	39	11	1.9%	0.04	9.1%
ITI-2	39	11	1.9%	1.34	9.1%
ITI-3	43	8	2.9%	0.87	9.4%

Table 4.6: Comparison of trees generated by batch and incremental approaches.

The comparison test of performance is listed in the above table. Here, in all testing node, at least two branches must contain a minimum number of instances, which is set as 2 for each algorithms. Except Gini for CART, every algorithm uses Gain-ratio as its splitting criteria. ITI-1 stands the batch mode for ITI, while ITI-2 is the complete incremental mode of ITI. ITI-3 makes a mixed use of batch mode and incremental mode, which separates the training data set into two halves, doing an initial batch training of the tree with the first half, and then incrementally incorporates the instances from the second half. We can see from above, ITI-2 generated exactly the same

tree as ITI-1, but with much more CPU efforts. Since ITI-3 took an initial training with batch mode, it saves much more CPU times than ITI-2, but compensate for a little higher error rates of both training and testing. In all approaches, CART created the most accurate tree, while C4.5 performance excellent in both CPU times and testing accuracy. However, ITI-1 seems to be good enough on both sides, and also ITI-2, excluding the CPU efforts, of course.

Normally, the mixed method of batch and incremental approaches probably helps for more stability during the learning process compared to an incremental training from scratch! Mostly in industrial systems, incremental training from scratch is hardly recommended as the operators/experts first want to get a feeling which performance is achievable by classifiers at all (is the task learnable at all) before deciding to install the whole classification system in their on-line production line; this means the incremental training is mostly used for a further adaptation and refinement of already generated classifiers in order to incorporate new system states and operating conditions and hence to improve on-line classification performance.

4.2 Evaluation on "Soybean Classification"

In this section, we focus on a more complicated data set "Soybean" from [27]. Although the data set we employed is smaller than the one we mentioned before, it includes more classes and attributes, and also missing values are emerged. There are 19 classes, but only the first 15 of which have been used in prior work. The last four classes are unjustified by the data, since they have so few examples. There are 35 categorical attributes, some nominal and some ordered. Initially, there are 307 training instances and 376 testing instances, each of which consists 35 attributes. For the reason of lack enough training data for the last four classes, we use only 290 instances for training and 340 instances for testing.

Classification Approach	Tree Complexity		Training Accuracy(290)		Testing Accuracy(340)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
ID3	65	15	10	3.4%	44	12.9%
C4.5	61	15	11	3.8%	44	12.9%
CART	59	9	40	13.8%	31	9.0%

Table 4.7: Comparison of trees generated by different classification approaches.

As shown above, ID3 and C4.5 trained precise trees with very low error rate. However, since CART generated a simpler tree(only 9 depth with 59 nodes), even with more errors, it perform much better on unseen data. 3.9% more accurate shows the stability of CART and makes it dominate in these three approaches.

Classification Approach	Tree Complexity		Training Accuracy(290)		Testing Accuracy(340)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
ID3 with Gain	73	12	7	2.4%	40	11.8%
C4.5 with Gain	73	12	7	2.4%	40	11.8%
CART with Twoing	61	12	40	13.8%	32	9.3%
CART with Deviance	59	10	36	12.4%	29	8.4%

Table 4.8: Comparison of different selection methods.

"Gain" brought more elaborate trees compared with "Gain Ration" to ID3 and C4.5, and meanwhile, the prediction effect improve. For CART, a result to be observe: the first level of pruning induces more accurate decision making when using "Deviance" as selection methods, especially on unseen data(the best result from all applications).

Classification Approach	Tree Complexity		Training Accuracy(290)		Testing Accuracy(340)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
ID3 (splitmin=5)	49	12	21	7.2%	37	10.9%
C4.5 (splitmin=5)	37	8	25	8.6%	35	10.3%
CART (splitmin=5)	73	10	33	11.4%	38	11.1%
CART (cross-validation)	59	9	45	15.5%	55	16.1%

Table 4.9: Comparison of different stopping criteria.

At the same splitting level, C4.5 shows its power on both accuracy and stability for training and

testing procedure, and thus is the best choice for "Soybean" classification. After 100 times run, a high average errors for training and testing do not indicate any superiority of the cross-validation for CART.

Classification Approach	Tree Complexity		Training Accuracy(290)		Testing Accuracy(340)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
C4.5 Windowing	65	17	4	1.4%	30	8.8%
C4.5 Rules	21.6	-	14	4.8%	30.5	9.0%
CART Pruned	59	9	40	13.8%	31	9.0%

Table 4.10: Comparison of different improved approaches.

The two advanced function of C4.5 present outstanding results: the error rate of "Windowing" is the lowest one for all training procedure and 8.8% is as well nearly the lowest for testing. "Rules" also offered an acceptable outcome. For "CART Pruned", as shown in Fig. 4.2, no pruning is executed, since the original one is already the best choice.

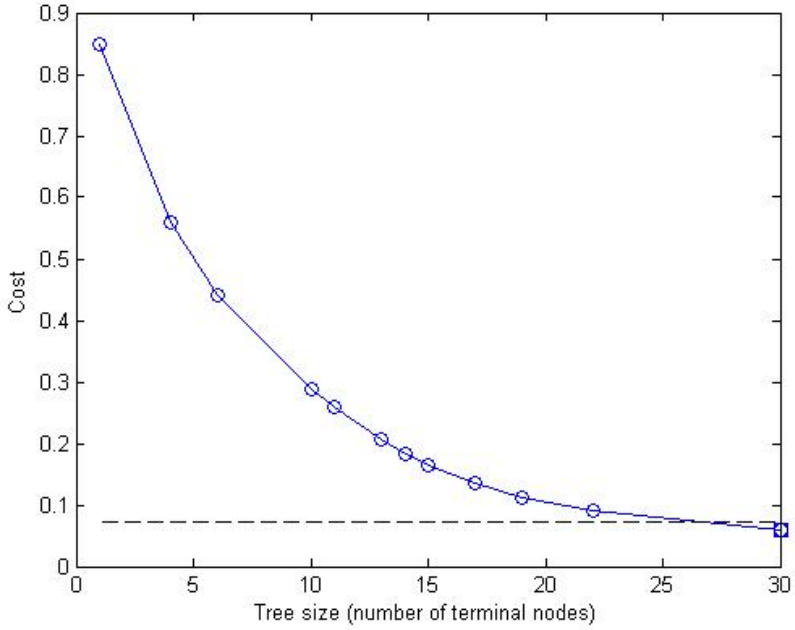


Figure 4.3: Smallest tree within 1 std. error of minimum cost tree for "Soybean".

For continuous attribute values, we can take the robust test by adding a disturbance on each original value, as the previous section described. However, for categorical attributes, adding a noise in percentage form has nonsense. For instance, we cannot say the outlook is "1+5% rainy". Hence, the only thing we could do here is that, we replace some of the attribute values with unknowns. In practice, P% noise level means that we assign P% single attribute values with "?" in ID3 and C4.5 or "NaN" in CART. Say, there are 10 instances and each of 10 attributes. 10% noise then is 10 values from 100 is unknown. However, due the limitation of software, we only got the results from ID3 and C4.5.

Error Rate	ID3		C4.5	
Noisy level	Training	Testing	Training	Testing
0	3.4%	12.9%	3.8%	12.9%
5%	5.6%	10.4%	8.4%	9.6%
10%	7.8%	12.5%	11.2%	10.2%
15%	9.7%	12.7%	13.7 %	10.1%
20%	11.7%	13.3%	16.4%	10.5%
25%	14.2%	13.4%	19.8%	10.9%
30%	17.2%	14.0%	23.8%	12.0%
40%	25.9%	16.3%	34.5%	16.5%
50%	37.0%	21.1%	48.5%	28.0%
60%	49.4%	28.7%	62.0%	42.7%
70%	59.3%	38.4%	72.5%	57.5%
80%	67.7%	50.7%	79.8%	67.2%
90%	76.8%	61.9%	83.8%	72.1%
100%	86.2%	85.9%	86.2%	85.9%

Table 4.11: Tree stability of different noise levels.

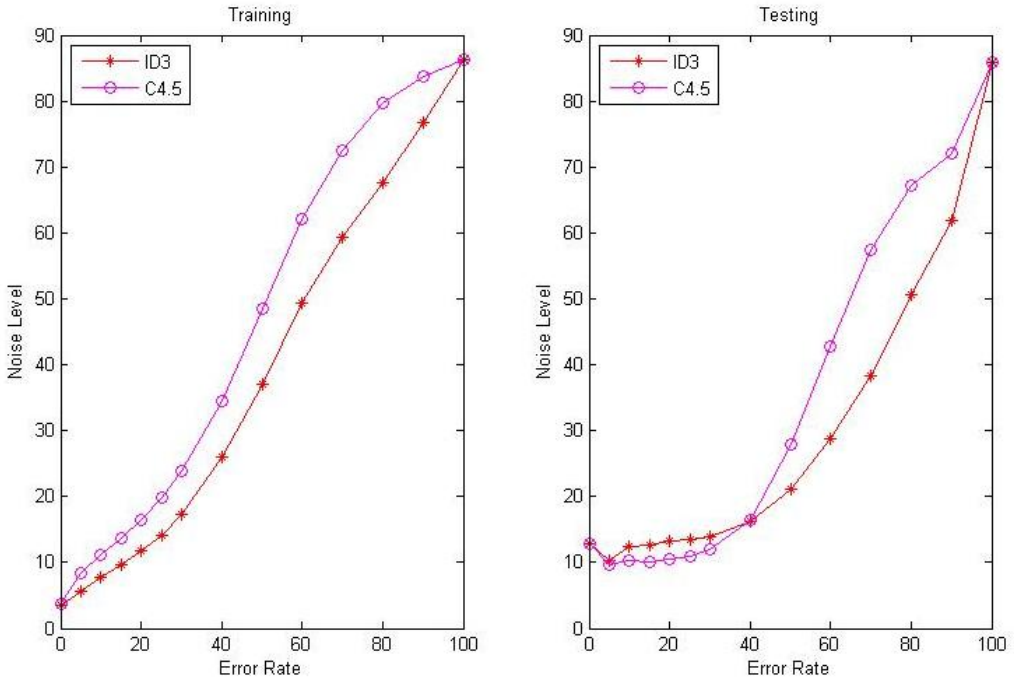


Figure 4.4: Classification performance of different noise level.

The noise are generated randomly, so we run 500 times for each testing and average error rates are recorded. The classification performance is even better with the noise level as large as 15% for ID3 and 30% for C4.5, since the initial decision tree created by non-noisy data is overfitting. Even if half of the data are missing, both of the approaches can classified around three quarters instances correctly. The classification error increases sharply for C4.5 when noise level larger than 50%,

and finally if all of the training data are randomly generated, about 86% misclassification appears in testing, which is still better than what we expected. Since the expectation of the accuracy for randomly classifying one instances is 9.0%, and subsequently the error rate expectation should be 91.0%, which is 5 more percentage then our observation.

Classification Approach	Tree Complexity		Training Accuracy(290)		Testing Accuracy(340)
Algorithm	Nodes	Depth	Error Rate	CPU Times	Error Rate
ID3	65	15	3.4%	0.01	12.9%
C4.5	61	15	3.8%	0.01	12.9%
CART	97	12	8.3%	0.23	12.3%
ITI-1	73	15	3.1%	0.02	11.5%
ITI-2	73	15	3.1%	0.37	11.5%
ITI-3	73	15	3.1%	0.57	11.5%

Table 4.12: Comparison of trees generated by batch and incremental approaches.

All of the parameter settings are the same as previous section. As we noted, all three modes of ITI generate exact the same decision tree, and the complete incremental mode used even less CPU effort than the mixed one, because after a initial training of batch mode, plenty of tree revision are taken during the incremental training procedure. Furthermore, ITI dominates all the others approaches in both training and testing performance, without spending too much time. CART trained an exhaustive tree but do not represent best in the classification.

4.3 Evaluation on "CD Imprint Inspection"

In this section, we imply a real world data set "CD Imprint" from Sony. There are 776 training instances and 388 testing instances, each of which consists 74 numerical (and no categorical) attributes, some continuous and some ordered. No missing value presents in both training and testing data sets, and only two classes (0 or 1) are labeled.

Classification Approach	Tree Complexity		Training Accuracy(776)		Testing Accuracy(388)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
ID3	57	12	5	0.6%	35	9.0%
C4.5	49	11	7	0.9%	32	8.2%
CART	37	9	5	0.6%	34	8.7%

Table 4.13: Comparison of trees generated by different classification approaches.

Distinctly different from previous two cases, each of these three approaches trained a very precise decision tree with error rate even smaller than 1%, since there are much more attributes (74) with full informed value in the training set than before. Therefore, all of these three approaches performed better than previous ones in both training and testing. Among the results, CART can build a simplest tree with lowest error rate, while C4.5 could perform a little better on unseen data.

Classification Approach	Tree Complexity		Training Accuracy(776)		Testing Accuracy(388)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
ID3 with Gain	27	7	7	0.9%	37	9.5%
C4.5 with Gain	27	7	7	0.9%	37	9.5%
CART with Twoing	37	9	5	0.6%	34	8.7%
CART with Deviance	31	8	6	0.8%	38	9.9%

Table 4.14: Comparison of different selection methods.

"Gain" brought much simpler trees compared with "Gain Ration" to ID3 and C4.5 in this case, but with a lightly lower prediction accuracy. For CART, using "Towing Rules" as a selection methods makes no difference from default "Gini" methods. However, "Deviance" trained a simpler tree with more error rate as a cost for training and testing.

Classification Approach	Tree Complexity		Training Accuracy(776)		Testing Accuracy(388)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
ID3 (splitmin=5)	31	10	23	3.0%	32	8.2%
C4.5 (splitmin=5)	29	9	23	3.0%	33	8.5%
CART (splitmin=5)	41	9	3	0.4%	34	8.7%
CART (cross-validation)	37	9	43	5.5%	42	10.7%

Table 4.15: Comparison of different stopping criteria.

At the same splitting level, C4.5 shows its power on tree generating simpleness, while CART trains a more accurate one. On unseen instances, these three approaches perform more or less the same. After 100 times run, a high average errors for training and testing do not indicate any superiority of the cross-validation for CART.

Classification Approach	Tree Complexity		Training Accuracy(776)		Testing Accuracy(388)	
Algorithm	Nodes	Depth	Errors	Rate	Errors	Rate
C4.5 Windowing	31	10	6	0.8%	28	7.2%
C4.5 Rules	10.8	-	9.5	1.2%	26.5	6.8%
CART Pruned	31	9	6	0.8%	34	8.7%

Table 4.16: Comparison of different improved approaches.

Trial 1 is recorded for "C4.5 Windowing" function, and as shown in the above table, both tree complexity and testing accuracy are improved evidently. "C4.5 Rules" shows the best outcome for unseen data when taking slightly more errors as a penalty for training. For "CART Pruned", as illustrated in Fig. 4.3 , CART deletes one branch of the original tree and merges 4 leaf nodes into one. However, it do not make any help for training and testing accuracies.

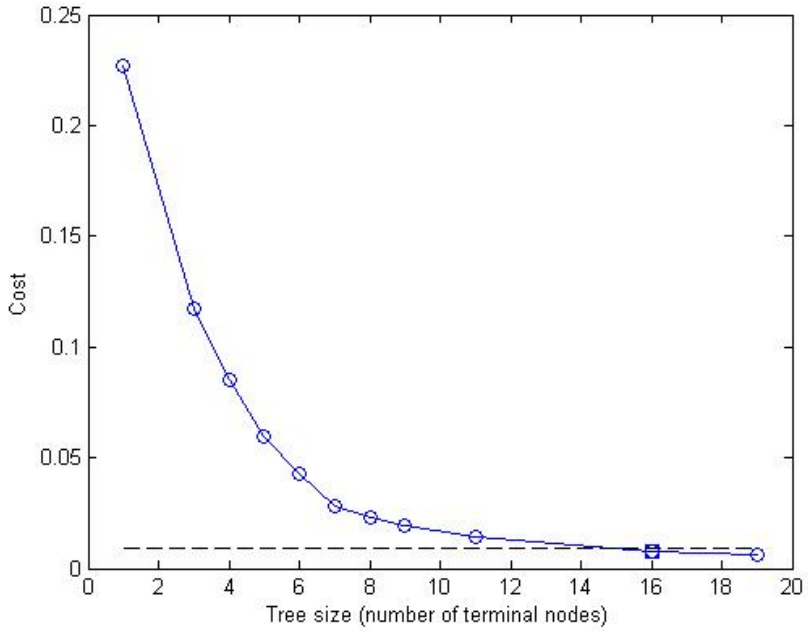


Figure 4.5: Smallest tree within 1 std. error of minimum cost tree for "CD Imprint".

In the table and figure below, we can see that for training, ID3 performs slightly better than C4.5, and C4.5 appears a little better than CART. For classification testing, when noise level alters from 0 to 40%, C4.5's outcomes are the best; when noise level varies from 50% to 80%, ID3 is the best choice; lastly when noise level changes from 90% to 100%, CART acts superiorly. Furthermore, we found the stability of these three approaches represent much better than what we have seen in "Segmentation" testing. The reason is, in the previous section, there are as many as 7 classes in the data sets, while in this section, we have only two, the most simple and typical P-N classes. Then, we can conclude that the less the classes of the instances are, the more accurate is the classification method.

Error Rate	ID3		C4.5		CART	
Noisy level	Training	Testing	Training	Testing	Training	Testing
0	0.6%	9.0%	0.9%	8.2%	0	8.9%
5%	2.3%	9.5%	2.4%	8.8%	2.8%	9.7%
10%	3.6%	9.5%	3.9%	8.8%	4.9%	11.1%
15%	6.2%	11.6%	6.4%	10.8%	5.3%	12.1%
20%	7.5%	12.4%	7.6%	11.6%	7.7%	13.7%
25%	9.0%	12.4%	9.1%	12.1%	9.3%	14.0%
30%	9.8%	12.1%	9.9%	11.9%	10.4%	13.8%
40%	11.9%	13.4%	12.2%	13.4%	13.9%	15.4%
50%	13.3%	14.2%	13.3%	14.4%	16.0%	15.9%
60%	15.2%	15.5%	15.5%	16.0%	16.9%	16.7%
70%	16.5%	16.5%	16.4%	17.0%	17.1%	17.7%
80%	16.6%	17.3%	16.6%	17.8%	17.8%	17.7%
90%	17.1%	18.6%	17.0%	19.1%	18.4%	18.3%
100%	19.1%	20.9%	19.3%	21.1%	18.8%	19.0%

Table 4.17: Tree stability of different noise levels.

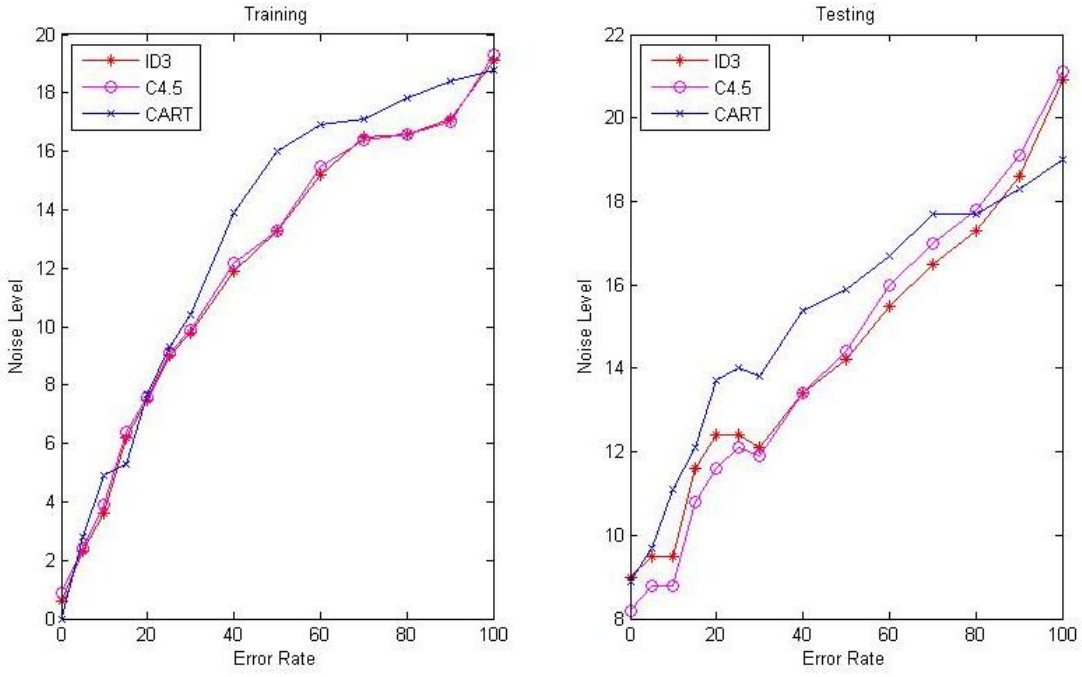


Figure 4.6: Classification performance of different noise level.

Classification Approach	Tree Complexity		Training Accuracy(776)		Testing Accuracy(388)
Algorithm	Nodes	Depth	Error Rate	CPU Times	Error Rate
ID3	57	12	0.6%	0.09	9.0%
C4.5	49	11	0.9%	0.09	8.2%
CART	37	9	0.6%	0.95	8.4%
ITI-1	77	9	1.2%	0.28	10.3%
ITI-2	77	9	1.2%	31.14	10.3%
ITI-3	73	10	2.2%	17.46	10.3%

Table 4.18: Comparison of trees generated by batch and incremental approaches.

In all of the decision tree generating approaches, C4.5 has the highest overall rank, considering tree complexity, CPU effort, training and testing error rate. CART also trained an excellent tree, regardless the CPU times. Compared with the other three algorithms, ITI here does not exhibit well enough.

Chapter 5

Conclusion and Outlook

In the previous chapters, we have theoretically and practically discussed the batch learning approaches of decision trees and their incremental versions. In the analysis of their performance for classification, we could not easily say which one is absolutely superior than the others concerning different data types. Roughly, we can conclude that, C4.5 presents to be more accurate on numeric-valued attribute data set and CART does well on the categorical ones. However, for the robust testing, their performance reversed. Furthermore, we found that ITI is an excellent incremental algorithm for decision tree generating. It maintains all the functions (actually even stronger) of its offline ancestor ID3, and does not take too much computer effort, especially for the mixed mode. So, ITI is very suitable for online data dealing, such as knowledge maintenance systems.

We have presented a small introduction of the incremental CART algorithm. However, this approaches is expensive and proposes an alternative that invokes tree rebuilding less often. We do not make an implementation analysis here in this paper, which is deserved for further study. A commercial version of C4.5, called "See5", is created by Quinlan and several capabilities are improved. We can take some inside look of it in the future research. And we can also make wide comparisons with other classification methods, such like "Discriminant Analysis", "Neural Network" and "K-nearest Neighbor".

Bibliography

[1] M.J.A. Berry and G. Linoff, Data Mining Techniques for Marketing, Sales, and Customer Support, Wiley, New York (1997).

[2] Michael Doumpos and Constantin Zopounidis, Model combination for credit risk assessment: A stacked generalization approach. *Annals of Operations Research*,2006.

[3] H.J. Zimmermann, *Fuzzy Set Theory–and Its Applications*. Springer, 2001.

[4] Alberto Freitas, Altamiro Costa-Pereira and Pavel Brazdil, Cost-Sensitive Decision Trees Applied to Medical Data. *Lecture Notes in Computer Science*,2007.

[5] D. J. Kerbyson, E. Papaefstathiou and G. R. Nudd, Application execution steering using on-the-fly performance prediction. *High-Performance Computing and Networking*,1998.

[6] E. Lughofer, On-Line Evolving Image Classifiers and Their Application to Surface Inspection. *submitted to Image and Vision Computing*, 2009.

[7] D. P. Filev and F. Tseng, Novelty Detection Based Machine Health Prognostics. Proc. of the 2006 International Symposium on Evolving Fuzzy Systems, Lake District, UK, pp. 193–199, 2006

[8] E. Lughofer and C. Guardiola, On-line fault detection with data-driven evolving fuzzy models. *Journal of Control and Intelligent Systems*, vol.36 (2), 2008.

[9] P. Angelov, V. Giglio, C. Guardiola, E. Lughofer, and J.M, Luján. An approach to model-based fault detection in industrial measurement systems with application to engine test benches. *Measurement Science and Technology*, 17:1809–1818, 2006.

[10] R. Duda, P. Hart, and D. Stork, *Pattern Classification - Second Edition*. Southern Gate, Chichester, West Sussex PO 19 8SQ, England: Wiley-Interscience, 2000.

[11] S. Haykin, *Neural Networks: A Comprehensive Foundation (2nd Edition)*, Prentice Hall Inc., Upper Saddle River, New Jersey 07458, 1999

[12] T. Hastie and R. Tibshirani and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer Verlag, New York, Berlin, Heidelberg, Germany, 2001

[13] L. Kuncheva. *Fuzzy Classifier Design*. Physica-Verlag, Heidelberg, 2000

[14] M.Stone, Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society*, 36:111–147, 1974.

[15] R. Quinlan, Induction of decision trees, *Machine Learning*, vol. 1, pp. 81–106, 1986.

[16] J. R. Quinlan, *C4.5: Programs for Machine Learning*. U.S.A.: Morgan Kaufmann Publishers Inc, 1993.

[17] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classification and Regression Trees*. Boca Raton: Chapman and Hall, 1993.

[18] A. M. Mood and F. A. Graybill, *Introduction to the Theory of Statistics, 2nd edition*, New York: McGraw-Hill, 1963, Section 11.6.

[19] M. Abramowitz and I. Stegun ed., *Handbook of Mathematical Functions*, New York Dover, 1964. Sections 26.5 and 26.2.

[20] L.Prechelt, Automatic early stopping using cross validation: quantifying the criteria *Neural Networks*, 1998, Elsevier.

[21] L.Prechelt, Early Stopping-But When? *Lecture Notes In Computer Science*, 1998, Vol.1524.

[22] P.Utgoff, Incremental induction of decision trees, *Machine Learning*, vol. 4, pp. 161–186, 1989.

[23] P.Utgoff, Decision Tree Induction Based on Efficient Tree Restructuring, NC Berkman, JA Clouse-*Machine Learning*, 1997, Springer.

[24] J.Rissanen, Modeling by shortest data description, *Automatica*, 1978.

[25] D. Kalles and T. Morris, Efficient incremental induction of decision trees, *Machine Learning*, vol. 24, pp. 231–242, 1996.

[26] S. Crawford, Extensions to the CART algorithm, *International Journal of Man-Machine Studies*, vol. 31, pp. 197–217, 1989.

[27] R.S. Michalski and R.L. Chilausky, Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis, *International Journal of Policy Analysis and Information Systems*, Vol. 4, No. 2, 1980.

Appendix

THE DATASETS

Data sets "Segmentation" and "Soybean" are obtained from UCI Machine Learning Repository, available website: `http://mlearn.ics.uci.edu/databases/`. Data set "CD imprint Inspection" is provided by Sony Co..

IMPLEMENTATION

The experiments in Chapter 4 were all carried out using NEC VERSA P550, 1.86GHz. Source codes of ID3 and C4.5 come from Quinlan's personal website: `http://www.rulequest.com/Personal/` and ITI from Machine Learning Laboratory: `http://www-lrn.cs.umass.edu/iti/index.html`. All the other tests are programmed by myself using Matlab.