

Chapter 2

Inferring a Boolean Function from Positive and Negative Examples

2.1 An Introduction

A central problem in data mining is how to analyze observations grouped into two categories and infer some key patterns that may be implied by these observations. As discussed in Chapter 1, these observations describe different states of nature of the system or phenomenon of interest to the analyst.

The previous chapter had a description of some possible application areas where data from observations may be used to study a variety of natural or man-made systems. Although there may be more than two classes when analyzing a system, we assume here that we have only two. Situations with more than two classes can be transformed into a set of two-class problems. Furthermore, it is assumed that these two groups (classes) of observations are exhaustive and exclusive. That is, the system has to be in only one of these two classes at any given moment.

The goal is to somehow analyze the data in these two groups of observations and try to infer some key pattern(s) that may be implied by these data. This could be important for a number of reasons. For instance, one may have the definition of a new data point (or points) but without information of its (their) class. Then, it is of interest to use the inferred patterns and assign it (them) to one of the two classes. If the available information is not adequate, then the new point(s) may not be assigned to any of the two classes and be deemed as undecidable, or as *do not know* case(s).

In the following it is assumed that the data are binary vectors (i.e., their individual fields take on 0/1 values). This is not a real limitation as nonbinary data can easily be transferred into binary ones. As the following section illustrates, this binary data and two-class problem has been studied extensively in the literature.

This chapter is organized as follows. After the following section, which reviews some key developments from the literature, a simple method is presented as to how nonbinary data can be transferred into equivalent binary data. The fourth section introduces the required terminology and notation. Sections five and six provide some formulations to this pattern inference problem. Sections seven, eight and nine

describe some developments for solving this problem by means of a branch-and-bound search. They also provide an approach for data preprocessing. Section eleven describes some computational results. This chapter concludes with section twelve.

2.2 Some Background Information

As mentioned above, suppose that some observations are available and they describe the behavior of a system of interest. It is also assumed that the behavior of this system is fully described by a number, say n , of *attributes* (also known as *parameters*, *variables*, *criteria*, *characteristics*, *predicates*, or just *features*). Thus, vectors of size n define these observations. The i -th (for $i = 1, 2, 3, \dots, n$) element of such a vector corresponds to the value of the i -th attribute. These attributes may be of any data type. For instance, they may take on continuous, discrete, or binary (i.e., 0/1) values. Furthermore, each observation belongs to one and only one of two distinct classes. It is also assumed that the observations are *noise free*. That is, the class value associated with each observation is the correct one. In general, these two classes are called the *positive* and *negative* classes. These names are assigned arbitrarily. Thus, the examples in the positive (negative) class will be called the positive (negative) examples.

One may assume that some observations, say m , are already available. New observations (along with their class membership) may become available later but the analyst has no control on their composition. In addition to the previous scenario, the analyst may be able to define the composition of new observations (i.e., to set the values of the n attributes) and then perform a test, or ask an expert (known as an *oracle* in the literature) to determine the class membership of a new observation. The main goal is to use the available classified observations to extract the underlying behavior of the target system in terms of a pattern. Next, this pattern is used to, hopefully, accurately infer the class membership of unclassified observations.

The extraction of new knowledge in the form of some kind of a model from collections of classified data is a particular type of *learning from examples*. Learning from examples has attracted the interest of many researchers in recent years. In the typical learning problem of this type, both positive and negative examples are available and the main goal is to determine a Boolean expression (that is, a set of logical rules or clauses) which accepts all the positive examples, while it rejects all the negative examples.

This kind of learning has been examined intensively (see, for instance, [Carbonell, *et al.*, 1983], [Dietterich and Michalski, 1983], [Kamath, *et al.*, 1992], [Kearns, *et al.*, 1987], [Pitt and Valiant, 1988], [Quinlan, 1986], and [Valiant, 1984]). Typically, the knowledge base of an intelligent system can be expressed as a Boolean function either in the conjunctive normal form (CNF) or in the disjunctive normal form (DNF) (see, for instance, [Blair, Jeroslow, and Lowe, 1985], [Cavalier, Pardalos, and Soyster, 1990], [Hooker, 1988a; 1988b], [Jeroslow, 1988; 1989], [Kamath, *et al.*, 1990], [Kamath, *et al.*, 1992], [Valiant, 1984], and [Williams, 1987]).

A considerable amount of related research is today known as the *PAC* (for *Probably Approximately Correct*) learning theory (see, for instance, [Valiant, 1984], [Angluin, 1988], and [Haussler and Warmuth, 1993]). The central idea of the PAC model is that successful learning of an unknown target concept should entail obtaining, with high probability, a hypothesis that is a good approximation of the target concept (hence the term: *probably approximately correct*). The error associated with the approximation of the target concept is defined as the probability that the proposed concept (denoted as h) and the target concept (denoted as c) will disagree on classifying a new example drawn randomly from unclassified examples. Later in this chapter this notion of error is used frequently and is related to another concept used extensively in this chapter called *accuracy rate*. The hypothesis h is a good approximation of the target concept if the previous error is small (less than some quantity ε , where $1 > \varepsilon > 0$).

In the same framework of thought, a learning algorithm is then a computational procedure which takes a sample of random positive and negative examples of the target concept c and returns a hypothesis h . In the literature a learning algorithm A is a PAC algorithm if for all positive numbers ε and δ (where $1 > \varepsilon, \delta > 0$), when A runs and accesses unclassified examples, then it eventually halts and outputs a concept h with probability at least $1 - \delta$ and error at most equal to ε [Angluin, 1992].

Conjunctive concepts are properly PAC learnable [Valiant, 1984]. However, the class of concepts in the form of the disjunction of two conjunctions is not properly PAC learnable [Pitt and Valiant, 1988]. The same is also true for the class of existential conjunctive concepts on structural instance spaces with two objects [Haussler, 1989]. The classes of k -DNF, k -CNF, and k -decision lists are properly PAC learnable for each fixed k (see, for instance, [Valiant, 1985], [Rivest, 1987], and [Kearns, *et al.*, 1987]), but it is unknown whether the classes of all DNF, or CNF functions are PAC learnable [Haussler and Warmuth, 1993] and [Goldman, 1990]. In [Mansour, 1992] an $n^{O(\log \log n)}$ algorithm is given for learning DNF formulas (however, *not* of minimal size) under a uniform distribution by using membership queries.

Another related issue is the sample complexity of a learning algorithm, that is, the number of examples needed to accurately approximate a target concept. The presence of bias in the selection of a hypothesis from the hypothesis space can be beneficial in reducing the sample complexity of a learning algorithm. Usually the amount of bias in the hypothesis space H is measured in terms of the *Vapnik–Chernovenkis dimension*, denoted as $VCdim(H)$ [Haussler, 1988].

There are many reasons why one may be interested in inferring a Boolean function with the minimum (or near minimum) number of terms. In an electronic circuit design environment, a minimum size Boolean representation is the prerequisite for a successful VLSI application. In a learning from examples environment, one may be interested in deriving a compact set of classification rules which satisfy the requirements of the input examples. As mentioned in the previous chapter, this can be motivated for achieving the maximum possible simplicity (as stated succinctly by *Occam's razor*) which could lead to easy verification and validation of the derived new knowledge.

Since the very early days it was recognized that the problem of inferring a Boolean function with a specified number of clauses is NP-complete (see, for instance, [Brayton, *et al.*, 1985] and [Gimpel, 1965]). Some related work in this area is due to [Bongard, 1970]. The classical approach for dealing with this Boolean function inference problem as a minimization problem (in the sense of minimizing the number of CNF or DNF clauses) was developed in [Quine, 1952 and 1955] and [McCluskey, 1956]. However, the exact versions of the Quine–McCluskey algorithm cannot handle large-scale problems. Thus, some heuristic approaches have been proposed. These heuristics include the systems MINI [Hong, *et al.*, 1974], PRESTO [Brown, 1981], and ESPRESSO-MV [Brayton, *et al.*, 1985]. Another widely known approach in dealing with this problem is the use of *Karnaugh maps* [Karnaugh, 1953]. However, this approach cannot be used to solve large-scale problems [Pappas, 1994]. Another application of Boolean function minimization can be found in the domain of multicast [Chang, *et al.*, 1999] where one needs a minimum number of keys.

A related method, denoted as SAT (for satisfiability), has been proposed in [Kamath, *et al.*, 1992]. In that approach one first pre-assumes an upper limit on the number of clauses to be considered, say k . Then a clause satisfiability (SAT) model is formed and solved using an interior point method developed by Karmakar and his associates [Karmakar, Resende, and Ramakrishnan, 1992]. If this clause satisfiability problem is feasible, then the conclusion is that it is possible to correctly classify all the examples with k or fewer clauses. If the SAT problem (which essentially is an integer programming model) is infeasible, then one must increase k until feasibility is reached. In this manner, the SAT approach yields a system with the minimum number of clauses.

It is important to observe at this point that from the computational point of view it is much harder to prove that a given SAT problem is infeasible than to prove that it is feasible. Therefore, trying to determine a minimum size Boolean function by using the SAT approach may be computationally too difficult. Some computational results indicate that the B&B approach proposed in [Triantaphyllou, 1994] (and as described in Chapter 3 of this book) is more efficient than the previous satisfiability-based approach. Actually, that B&B approach is on the average 5,500 times faster in those tests.

In [Felici and Truemper, 2002] the authors propose a different use of the SAT model. They formulate the problem of finding a clause with maximal coverage as a minimum cost satisfiability (MINSAT) problem and solve such problem iteratively by using the logic SAT solver *Leibniz*, which was developed by Truemper [1998]. That method is proved to be computationally feasible and effective in practice. The same authors also propose several variants and extensions to that system. Further extensions on this learning approach are also discussed in [Truemper, 2004].

A very closely related problem is to study the construction of a partially defined Boolean function (or pdBf), not necessarily of minimal size, given disjoint sets of positive and negative examples. That is, now it is required that the attributes of the function be grouped according to a given scheme (called a decomposition structure)

[Boros, *et al.*, 1994]. Typically, a pdBf may have exponentially many different extensions.

It should be stated here that there are a multitude of methods for inferring a Boolean function from two sets of training examples. A review of some recent developments of methods that infer rules (which in essence are like classification Boolean functions) can be found in [Triantaphyllou and Felici, 2006].

In summary, the most representative advances in distinguishing between observations in two or more classes can be classified into some distinct categories as follows: Some *common logic* approaches by [Zakrevskij, 1988; 1994; 1999; 2001; and 2006]. Clause *satisfiability* approaches to inductive inference such as the methods by Kamath, *et al.*, [1992, 1994] and [Felici and Truemper, 2002]. *Boolean function* (i.e., *logic*)-based approaches such as the methods in [Triantaphyllou, *et al.*, 1994], [Triantaphyllou, 1994] (these developments are described in detail later in this and the next chapter); some polynomial time and NP-complete cases of Boolean function decomposition by [Boros, *et al.*, 1994]; *association rules* [Adamo, 2000]; rough and fuzzy sets [Wang, Liu, Yao, Skowron, 2003]. *Decision tree*-based approaches [Quinlan, 1979; 1986], [Freitas, 2002] and [Witten and Eibe, 2005]. *Support vector machines* (SVM) by [Woldberg and Mangasarian, 1990], [Mangasarian, *et al.*, 1990], [Mangasarian, *et al.*, 1995], [Abe, 2005], and [Wang, 2005]. Knowledge-based learning approaches by *combining symbolic and connectionist machine* (neural networks)-based learning as proposed by Shavlik [1994], Fu [1993], Goldman and Sloan [1994] and Cohn, *et al.* [1994]. *Neural networks* [Arbib, 2002] and [Dayan and Abbot, 2001]. Various *rule induction* approaches as described in the edited book by [Triantaphyllou and Felici, 2006]; and finally, some *nearest neighbor* classification approaches [Hattori and Torri, 1993], [Kurita, 1991], [Kamgar-Parsi and Kanal, 1985], [Perner and Rosenfeld, 2003], [Berry, Kamath, and Skillicorn, 2004]. The above listing is not exhaustive as the field of data mining is still expanding rapidly, both in terms of theory and applications.

The main challenge in inferring a target set of discriminant classification rules from positive and negative examples is that the user may *never* be absolutely certain about the correctness of the classification rules, unless he/she has used the entire set of all possible examples which is of size 2^n in the binary case with n attributes. In the general case this number is too high. Apparently, even for a small value of n , this task may be practically impossible to realize.

Fortunately, many real-life applications are governed by the behavior of a *monotone* system or they can be described by a *combination of a small number of monotone systems*. In data mining the property of monotonicity offers some unique computational advantages. By knowing the value of certain examples, one can easily infer the values of more examples. This, in turn, can significantly expedite the learning process. This chapter discusses the case of inferring general Boolean functions from disjoint collections of training examples. The case of inferring a monotone Boolean function is discussed in Chapter 10 of this book.

2.3 Data Binarization

The main idea of how to transform any data type into binary ones is best described via a simple illustrative example. Suppose that the data in Table 2.1 represent some sampled observations of the function of a system of interest. Each observation is described by the value of two *continuous* attributes denoted as A_1 and A_2 . Furthermore, each observation belongs to one of two classes, denoted as Class 1 and Class 2. A number of problems can be considered at this point. The main problem is how to derive a pattern, in the form of a set of rules, which is consistent with these observations. As the set of rules we consider here logical clauses in the CNF (conjunctive normal form) or DNF (disjunctive normal form). That is, we seek the extraction of a Boolean function in CNF or DNF form. A more detailed description of the CNF and DNF forms is given in the next section.

Although, in general, many such Boolean functions can be derived, the focus of the proposed approach is on the derivation of a function of *minimal size*. By minimal size we mean a Boolean function which consists of the minimum number of CNF or DNF clauses. We leave it up to the analyst to decide whether he/she wishes to derive CNF or DNF functions. As explained in Chapter 7, Boolean functions in CNF (DNF) can easily be derived by using algorithms that initially derive Boolean functions in DNF (CNF).

Next we will demonstrate how the continuous data depicted in Table 2.1 can be represented by equivalent observations defined on only binary attributes. This is achieved as follows. We start with the first continuous attribute, i.e., attribute A_1 in this case, and we proceed until we cover all the continuous attributes.

From Table 2.1 it can be observed that the *ordered* set, denoted as $Val(A_1)$, with all the values of attribute A_1 is defined as the following ordered list:

$$\begin{aligned} Val(A_1) &= \{V_i(A_1), \text{ for } i = 1, 2, 3, \dots, 9\} \\ &= \{0.25, 0.50, 0.75, 1.00, 1.25, 1.50, 1.75, 2.25, 2.75\}. \end{aligned}$$

That is, $V_1(A_1) = 0.25$, $V_2(A_1) = 0.50$, $V_3(A_1) = 0.75$, \dots , $V_9(A_1) = 2.75$.

Table 2.1. Continuous Observations for Illustrative Example.

Example			Class			Example			Class		
No.	A_1	A_2	No.			No.	A_1	A_2	No.		
1	0.25	1.50	1			12	1.00	0.75	1		
2	0.75	1.50	1			13	1.50	0.75	1		
3	1.00	1.50	1			14	1.75	0.75	2		
4	0.50	1.25	1			15	0.50	0.50	1		
5	1.25	1.25	2			16	1.25	0.50	2		
6	0.75	1.00	1			17	2.25	0.50	2		
7	1.25	1.00	1			18	2.75	0.50	2		
8	1.50	1.00	2			19	1.25	0.25	2		
9	1.75	1.00	1			20	1.75	0.25	2		
10	2.25	1.00	2			21	2.25	0.25	2		
11	0.25	0.75	1								

Obviously, the cardinality of this set (i.e., the number of elements in this set) is at most equal to the number of all available observations. In this instance, the cardinality is equal to 9. Next, we introduce 9 binary attributes $A'_{1,i}$ (for $i = 1, 2, 3, \dots, 9$) as follows:

$$A'_{1,i} = \begin{cases} 1, & \text{if and only if } A_{1,i} \geq V_i(A_1), \text{ for } i = 1, 2, 3, \dots, 9, \\ 0, & \text{otherwise.} \end{cases}$$

In general, the previous formula becomes for any multivalued attribute A_j (where K is the cardinality of the set $V_i(A_j)$):

$$A'_{j,i} = \begin{cases} 1, & \text{if and only if } A_{j,i} \geq V_i(A_j), \text{ for } i = 1, 2, 3, \dots, K, \\ 0, & \text{otherwise.} \end{cases}$$

Using the above-introduced binary attributes, from the second observation (i.e., vector $(0.75, 1.50) = (A_{1,2}, A_{2,2})$) we get for its first attribute (please note that $A_{1,2} = 0.75$)

$$\{A'_{1,1}, A'_{1,2}, A'_{1,3}, A'_{1,4}, A'_{1,5}, A'_{1,6}, A'_{1,7}, A'_{1,8}, A'_{1,9}\} = \{1, 1, 1, 0, 0, 0, 0, 0, 0\}.$$

Similarly with the above definitions, for the second continuous attribute A_2 the set $Val(A_2)$ is defined as follows:

$$\begin{aligned} Val(A_2) &= \{V_i(A_2), \text{ for } i = 1, 2, 3, \dots, 6\} \\ &= \{0.25, 0.50, 0.75, 1.00, 1.25, 1.50\}. \end{aligned}$$

Working as above, for the second observation we have

$$\{A'_{2,1}, A'_{2,2}, A'_{2,3}, A'_{2,4}, A'_{2,5}, A'_{2,6}\} = \{1, 1, 1, 1, 1, 1\}.$$

The above transformations are repeated for each of the nonbinary attributes. In this way, the transformed observations are defined on *at most* $m \times n$ binary attributes (where m is the number of observations and n is the original number of attributes). The precise number of the transformed attributes can be easily computed by using the following formula:

$$\sum_{i=1}^n |Val(A_i)|,$$

where $|s|$ denotes the cardinality of set s .

The binary attributed observations which correspond to the original data (as given in Table 2.1) are presented in Table 2.2 (parts (a) and (b)).

From the way the binary attributes have been defined, it follows that the two sets of observations are equivalent to each other. However, the observations in Table 2.1 are defined on continuous attributes while the observations in Table 2.2 are defined on binary ones.

Table 2.2a. The Binary Representation of the Observations in the Illustrative Example (first set of attributes for each example).

Example No.	First set of attributes: $A'_{1,i}$, for $i = 1, 2, 3, \dots, 9$								
	$A'_{1,1}$	$A'_{1,2}$	$A'_{1,3}$	$A'_{1,4}$	$A'_{1,5}$	$A'_{1,6}$	$A'_{1,7}$	$A'_{1,8}$	$A'_{1,9}$
1	1	0	0	0	0	0	0	0	0
2	1	1	1	0	0	0	0	0	0
3	1	1	1	1	0	0	0	0	0
4	1	1	0	0	0	0	0	0	0
5	1	1	1	1	1	0	0	0	0
6	1	1	1	0	0	0	0	0	0
7	1	1	1	1	1	0	0	0	0
8	1	1	1	1	1	1	0	0	0
9	1	1	1	1	1	1	1	0	0
10	1	1	1	1	1	1	1	1	0
11	1	0	0	0	0	0	0	0	0
12	1	1	1	1	0	0	0	0	0
13	1	1	1	1	1	1	0	0	0
14	1	1	1	1	1	1	1	0	0
15	1	1	0	0	0	0	0	0	0
16	1	1	1	1	1	0	0	0	0
17	1	1	1	1	1	1	1	1	0
18	1	1	1	1	1	1	1	1	1
19	1	1	1	1	1	0	0	0	0
20	1	1	1	1	1	1	1	0	0
21	1	1	1	1	1	1	1	1	0

Table 2.2b. The Binary Representation of the Observations in the Illustrative Example (second set of attributes for each example).

Example No.	First set of attributes: $A'_{2,i}$, for $i = 1, 2, 3, \dots, 6$						Class No.
	$A'_{2,1}$	$A'_{2,2}$	$A'_{2,3}$	$A'_{2,4}$	$A'_{2,5}$	$A'_{2,6}$	
1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1
4	1	1	1	1	1	0	1
5	1	1	1	1	1	0	2
6	1	1	1	1	0	0	1
7	1	1	1	1	0	0	1
8	1	1	1	1	0	0	2
9	1	1	1	1	0	0	1
10	1	1	1	1	0	0	2
11	1	1	1	0	0	0	1
12	1	1	1	0	0	0	1
13	1	1	1	0	0	0	1
14	1	1	1	0	0	0	2
15	1	1	0	0	0	0	1
16	1	1	0	0	0	0	2
17	1	1	0	0	0	0	2
18	1	1	0	0	0	0	2
19	1	0	0	0	0	0	2
20	1	0	0	0	0	0	2
21	1	0	0	0	0	0	2

Given the above considerations, it follows that the original problem has been transformed to the binary problem depicted in Table 2.2 (parts (a) and (b)). This problem has the following two sets of positive and negative examples, denoted as E^+ and E^- , respectively.

$$E^+ = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \text{ and}$$

$$E^- = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Finally, it should be stated here that [Bartnikowski, *et al.*, 2006] present a detailed study of the general binarization problem.

2.4 Definitions and Terminology

Let $\{A_1, A_2, A_3, \dots, A_n\}$ be a set of n Boolean *attributes*. Each attribute A_i (for $i = 1, 2, 3, \dots, n$) can be either true (denoted by 1) or false (denoted by 0). Let F be a *Boolean function* defined on these attributes. For instance, the expression $(A_1 \vee A_2) \wedge (A_3 \vee \bar{A}_4)$ is such a Boolean function, where “ \vee ” and “ \wedge ” stand for the logical “OR” and “AND” operators, respectively. That is, F is a mapping from $\{0, 1\}^n \rightarrow \{0, 1\}$ which determines for each combination of truth values of the attributes $A_1, A_2, A_3, \dots, A_n$ of F , whether F is true or false (denoted as 1 or 0, respectively).

For each Boolean function F , the *positive examples* are the vectors $v \in \{0, 1\}^n$ such that $F(v) = 1$. Similarly, the *negative examples* are the vectors $v \in \{0, 1\}^n$ such that $F(v) = 0$. Therefore, given a function F defined on the n attributes

$\{A_1, A_2, A_3, \dots, A_n\}$, then a vector $v \in \{0, 1\}^n$ is either a positive or a negative example.

Equivalently, we say that a vector $v \in \{0, 1\}^n$ is *accepted* (or *rejected*) by a Boolean function F if and only if the vector v is a positive (or a negative) example of F . For instance, let F be the Boolean function $(A_1 \vee A_2) \wedge (A_3 \vee A_4)$. Consider the two vectors $v_1 = (1, 0, 0, 0)$ and $v_2 = (1, 0, 0, 1)$. Then, it can be easily verified that $F(v_1) = 1$. That is, the vector v_1 is a positive example of the function F . However, the vector v_2 is a negative example of F (since $F(v_2) = 0$).

The motivation for the following developments is best illustrated via a simple illustrative example. Consider a system of interest (represented by some Boolean function) that involves the following four attributes: A_1, A_2, A_3 , and A_4 . We do not know its structure yet. In any situation each attribute can either be *true* (denoted by 1) or *false* (denoted by 0). For instance, in example $(0, 1, 1, 0)$ the attributes $A_2, A_3, \bar{A}_1, \bar{A}_4$, are true or, equivalently, $A_1, A_4, \bar{A}_2, \bar{A}_3$, are false. There are $2^4 = 16$ possible examples (also known as states of nature) for this system (Boolean function). If a Boolean function is specified, then each of these 16 examples could be categorized either as *positive* or as *negative*.

For systems in CNF (to be formally defined below) a state of nature corresponds to a positive example if and only if it is satisfied by each clause in the system (i.e., Boolean function to be inferred). For instance, the state $(0, 1, 0, 1)$ satisfies the clause $(A_1 \vee A_2 \vee \bar{A}_3)$ and thus it corresponds to a positive example (in terms of that clause). Similarly, a state is a negative example if it violates at least one of the clauses in the system (Boolean function). Next consider the following three Boolean clauses:

$$(\bar{A}_1 \vee \bar{A}_2 \vee A_3 \vee A_4), (A_1 \vee A_2), \text{ and } (\bar{A}_1 \vee A_2 \vee A_3).$$

Then, all the 16 possible states are characterized as $(1, 1, 1, 1)$ positive, $(1, 0, 0, 0)$ negative, $(1, 1, 0, 0)$ negative, and so on.

The terms Boolean function, Boolean expression, and system would be used to denote the same concept. Also, in this chapter it is assumed, unless otherwise stated, that any Boolean expression (and consequently any clause) is expressed in the conjunctive normal form (CNF). An example of a Boolean expression in CNF is $(A_1 \vee A_3 \vee A_4) \wedge (A_2 \vee \bar{A}_7) \wedge (A_1 \vee \bar{A}_6)$, which simply is a conjunction of disjunctions. The above expression evaluates to true value if and only if all three disjunctions evaluate to true value. It evaluates to false value if and only if at least one of the disjunctions evaluates to false value. More formally, a Boolean expression is in the *conjunctive normal form* (CNF) if it is in the form (where a_i is either A_i or \bar{A}_i and ρ_j is the set of indices)

$$\bigwedge_{j=1}^k \left(\bigvee_{i \in \rho_j} a_i \right). \quad (2.1)$$

Similarly, a Boolean expression is in the *disjunctive normal form* (DNF) if it is in the form

$$\bigvee_{j=1}^k \left(\bigwedge_{i \in \rho_j} a_i \right). \quad (2.2)$$

An example of a Boolean function in DNF is $(A_1 \wedge A_2) \vee (A_3 \wedge \bar{A}_4 \wedge A_5)$, which is a disjunction of conjunctions. Such an expression evaluates to true value if and only if at least one of the conjunctions evaluates to true value. It evaluates to false value if and only if all the conjunctions evaluate to false value. In other words, a CNF expression is a conjunction of disjunctions, while a DNF expression is a disjunction of conjunctions. It should be stated here that the “boxes” (rules) discussed in some of the figures in Chapter 1 correspond to DNF systems. This is true because a single box can be viewed as a conjunction of a set of conditions (see also Section 1.4.2).

It is known [Peysakh, 1987] that any Boolean function can be transformed into the CNF or the DNF form. Chapter 7 of this book provides a simple approach of how any algorithm that derives a Boolean function in CNF (DNF) can also derive a function in DNF (CNF) by performing some simple transformations.

In summary, a set of positive examples (to be denoted as E^+ in this book) and a set of negative examples (to be denoted as E^- in this book) are assumed to be available. These data will be used as the training data to infer a Boolean function. Given these two sets of positive and negative examples, the constraints to be satisfied by a Boolean function are as follows. In the CNF case, each positive example should be accepted by all the disjunctions in the CNF expression and each negative example should be rejected by at least one of the disjunctions. In the case of DNF systems, any positive example should be accepted by at least one of the conjunctions in the DNF expression, while each negative example should be rejected by all the conjunctions.

The general problem we analyze in this chapter is the construction of a set of Boolean expressions (clauses in CNF form) which correctly classify a set of sampled examples. We assume that each of these examples can be correctly classified (by an “oracle” or “expert”) either as a *positive example* or as a *negative example*. The “expert” somehow knows the correct identification of any example. Such an expert opinion could be the result of a test, or a series of tests, which could be used to classify examples of the way the system operates. Furthermore, the underlying system of interest is not explicitly known. As illustrated in the first chapter, this can be a common and very important problem in many and diverse application areas.

The “expert” somehow can identify (probably through experience or special tests) the nature of any particular example but lacks the ability to characterize the classification rules to be used for such classifications. Thus, an important challenge is to develop methods to approximate the hidden system in situations in which the nature of finite numbers of examples is known.

We will consider this problem in a practical and applied context. Instead of four attributes, consider the scenario in which we may have, say, 50 attributes. Here the number of all the possible states (examples) is $2^{50} = 1,125,899,906,842,624$. This is more than one quadrillion. It would be impractical to generate all possible examples. However, one may be able to generate and categorize a few hundreds or even thousands of sampled examples. From this *partial* set of examples, we will determine a particular set of CNF clauses which correctly classify all the sampled examples and, hopefully, a large proportion of the remaining ones.

2.5 Generating Clauses from Negative Examples Only

Consider any example α defined on n binary attributes. For instance, if $n = 5$, then consider an example such as $(1, 0, 1, 1, 0)$. Next, observe that the CNF clause $(\bar{A}_1 \vee A_2 \vee \bar{A}_3 \vee \bar{A}_4 \vee A_5)$ is satisfied by all examples $(d_1, d_2, d_3, d_4, d_5)$, where $d_i \in \{0, 1\}$, except $(1, 0, 1, 1, 0)$. The previous observation leads to the realization that a clause C_α can always be constructed which rejects any single example α while it accepts all other possible examples in the binary space of dimension n . In order to formalize this, let $ATTRIBUTES(\alpha)$ be the set of indices of the attributes which are true in example α . For instance, $ATTRIBUTES((1, 0, 1, 1, 0)) = \{1, 3, 4\}$. If the clause C_α is defined as

$$C_\alpha = (\beta_1 \vee \beta_2 \vee \beta_3 \vee \cdots \vee \beta_N),$$

where

$$\beta_i = \left\{ \begin{array}{ll} \bar{A}_i, & \text{if and only if } i \in ATTRIBUTES(\alpha) \\ A_i, & \text{otherwise} \end{array} \right\},$$

for each $i = 1, 2, 3, \dots, n$,

then the clause C_α will reject only example α and it will accept any other example. For instance, for the vector $\alpha = (1, 0, 1, 1, 0)$ the C_α clause is the previous CNF clause $(\bar{A}_1 \vee A_2 \vee \bar{A}_3 \vee \bar{A}_4 \vee A_5)$.

Suppose that m examples are somehow generated. Define E^+ as the set of m_1 examples which have been classified as positive examples and E^- as the set of the examples which have been classified as negative examples. These are the training examples to be used to generate a Boolean function.

For each of the m_2 (where $m_2 = m - m_1$) examples in E^- , we generate the unique clause as defined above. Each of these clauses rejects one and only one example and, hence, accepts all the examples in E^+ . This set of m_2 clauses precisely satisfies the first objective of inferring a Boolean function which would accept all positive examples and reject all the negative ones. However, this approach would be *impractical* for large selections of negative examples, since it would result in large numbers of clauses.

More importantly, the above approach would suffer immensely of the *overfitting* problem. That is, the pattern (set of Boolean clauses) that would be generated as described previously, would fit perfectly well the negative data and nothing else. Its generalization capability would be almost nil. In terms of the solid-line and dotted-line boxes depicted in Figure 1.7 in Chapter 1, the above situation would be like generating solid-line boxes that cover each of the solid dark points in Figure 1.7 (assuming that the solid dark points in that figure are the negative examples). It should be mentioned here that the other extreme situation is to have *overgeneralization* of the data. Having a way to control the overfitting and overgeneralization properties of the inferred system is of high priority in data mining and the subject of ongoing research activity.

From this discussion it becomes clear that it is important to have an approach that constructs a rather small (relative to m_1 and m_2) number of clauses. This would also

be desirable for the reasons of simplicity as described earlier in Chapter 1 (based on Occam's razor, etc.). The methods described in the following sections are such approaches.

2.6 Clause Inference as a Satisfiability Problem

In [Karmakar, *et al.*, 1991] it is shown that given two collections of positive and negative examples, then a DNF system can be inferred to satisfy the requirements of these examples. This is achieved by formulating a satisfiability (SAT) problem, which essentially is an integer programming (IP) problem, and then solve this IP problem by using the interior point method of Karmakar and his associates [Karmakar, *et al.*, 1991] as the solution strategy. This approach requires the specification of the number of conjunctions in the DNF system. The SAT problem uses the following Boolean variables [Karmakar, *et al.*, 1991]:

$$\begin{aligned}
 s_{ji} &= \begin{cases} 0, & \text{if } A_i \text{ is in the } j\text{-th conjunction} \\ 1, & \text{if } A_i \text{ is not in the } j\text{-th conjunction} \end{cases} \\
 s'_{ji} &= \begin{cases} 0, & \text{if } \bar{A}_i \text{ is in the } j\text{-th conjunction} \\ 1, & \text{if } \bar{A}_i \text{ is not in the } j\text{-th conjunction} \end{cases} \\
 \sigma_{ji}^\alpha &= \begin{cases} s_{ji}, & \text{if } A_i = 1 \text{ in the positive example } \alpha \in E^+ \\ s'_{ji}, & \text{if } A_i = 0 \text{ in the positive example } \alpha \in E^+ \end{cases} \\
 z_j^\alpha &= \begin{cases} 1, & \text{if the positive example } \alpha \text{ is accepted by the } j\text{-th conjunction} \\ 0, & \text{otherwise} \end{cases}
 \end{aligned}$$

Then, the clauses of this SAT problem are as follows:

$$s_{ji} \vee s'_{ji}, \quad \text{for } i = 1, \dots, n, \text{ and } j = 1, \dots, k, \quad (2.1a)$$

$$\left(\bigvee_{i \in P_r} \bar{s}_{ji} \right) \vee \left(\bigvee_{i \in \bar{P}_r} s_{ji} \right), \quad \text{for } i = 1, \dots, k, \text{ and } r = 1, \dots, m_2, \quad (2.2a)$$

$$\bigvee_{j=1}^k z_j^\alpha, \quad \text{for } \alpha = 1, \dots, m_1, \quad (2.3a)$$

$$\sigma_{ji}^\alpha \vee \bar{z}_j^\alpha, \quad \text{for } i = 1, \dots, n, j = 1, \dots, k, \text{ and } \alpha = 1, \dots, m_1, \quad (2.4a)$$

where P_r is the set of indices of A for which $A_i = 1$ in the negative example $r \in E^-$. Similarly, \bar{P}_r is the set of indices of A for which $A_i = 0$ in the negative example $r \in E^-$.

Clauses of type (2.1a) ensure that both A_i and \bar{A}_i will never appear in any conjunction. Clauses of type (2.2a) ensure that each negative example is rejected by all conjunctions. Clauses of type (2.3a) ensure that each positive example is accepted

by at *least one* conjunction. Finally, clauses of type (2.4a) ensure that $z_i^\alpha = 1$ if and only if the positive example α is accepted by the j -th disjunction. In general, this SAT problem has $k(n(m_1 + 1) + m_2) + m_1$ clauses, and $k(2n(1 + m_1) + nm_2 + m_1)$ Boolean variables. A detailed example of this formulation can be found in [Karmakar, *et al.*, 1991].

2.7 An SAT Approach for Inferring CNF Clauses

The SAT formulation for deriving CNF systems is based on the original SAT formulation for deriving DNF systems as described in the previous section. The variables used in the new formulation are similar to the ones used in the DNF case. They are defined in a similar way as in the previous section as follows:

$$\begin{aligned}
 s_{ji} &= \begin{cases} 0, & \text{if } A_i \text{ is in the } j\text{-th disjunction} \\ 1, & \text{if } A_i \text{ is not in the } j\text{-th disjunction} \end{cases} \\
 s'_{ji} &= \begin{cases} 0, & \text{if } \bar{A}_i \text{ is in the } j\text{-th disjunction} \\ 1, & \text{if } \bar{A}_i \text{ is not in the } j\text{-th disjunction} \end{cases} \\
 \sigma_{ji}^\beta &= \begin{cases} s_{ji}, & \text{if } A_i = 1 \text{ in the negative example } \beta \in E^- \\ s'_{ji}, & \text{if } A_i = 0 \text{ in the negative example } \beta \in E^- \end{cases} \\
 z_j^\beta &= \begin{cases} 1, & \text{if the negative example } \beta \text{ is accepted by the } j\text{-th disjunction} \\ 0, & \text{otherwise} \end{cases}
 \end{aligned}$$

The clauses of the SAT formulation for deriving a CNF system which has up to k disjunctions are as follows (where n is the number of attributes):

$$s_{ji} \vee s'_{ji}, \quad \text{for } i = 1, \dots, n, \text{ and } j = 1, \dots, k, \quad (2.1b)$$

$$\left(\bigvee_{i \in P_r} \bar{s}_{ji} \right) \vee \left(\bigvee_{i \in \bar{P}_r} s'_{ji} \right), \quad \text{for } i = 1, \dots, k, \text{ and } r = 1, \dots, m_1, \quad (2.2b)$$

$$\bigvee_{j=1}^k z_j^\beta, \quad \text{for } \beta = 1, \dots, m_2, \quad (2.3b)$$

$$\sigma_{ji}^\beta \vee \bar{z}_j^\beta, \quad \text{for } i = 1, \dots, n, j = 1, \dots, k, \text{ and } \beta = 1, \dots, m_2, \quad (2.4b)$$

where P_r is the set of indices of A for which $A_i = 1$ in the positive example $r \in E^+$. Similarly, \bar{P}_r is the set of indices of A for which $A_i = 0$ in the positive example $r \in E^+$.

Clauses of type (2.1b) ensure that *both* A_i and \bar{A}_i will *never* appear in any disjunction at the same time. Clauses of type (2.2b) ensure that each positive example will be accepted by *all* k disjunctions. Clauses of type (2.3b) ensure that each negative example will be rejected by at *least one* of the k disjunctions. Finally, clauses of

type (2.4b) ensure that $z_i^\beta = 1$ if and only if the negative example is rejected by the j -th conjunction. In general, this problem has $k(n(m_2 + 1) + m_1) + m_2$ clauses, and $k(2n + m_2)$ binary variables.

Next, suppose that the following are the two sets (it is assumed that $n = 4$) E^+ and E^- with the positive and negative examples of cardinality m_1 and m_2 , respectively. These data were used to derive the integer programming (IP) model of the SAT formulation and its solution, for the CNF case, shown in the Appendix of this chapter. This IP model is written for the LINDO integer programming solver and can be easily adapted to fit many other IP solvers.

$$E^+ = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad E^- = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}.$$

2.8 The One Clause At a Time (OCAT) Concept

The simple approach for creating clauses (disjunctions for a CNF expression) from negative examples described in Section 2.5, is very inefficient and ineffective. However, it provides some interesting insights. First of all, it is clear that in inferring a Boolean function from training data, one could start with a single data point and then move to another one and so on until all the points are covered. In the method described in Section 2.5 that strategy took place for the negative examples only. That is why that method is very inefficient as the derived systems suffer from extreme overfitting. The above idea provides the foundation for a sequential strategy. This is also the notion of the sequential covering algorithm discussed in [Tan, Steinbach, and Kumara, 2005].

Given a single clause, defined as before one could alleviate its problem of overfitting by removing items (i.e., attributes or their negations) off the definition of the clause. In that way, the clause would become less specific and more generalizing. That is, the modified clause would cover more than just a single negative example. This is how those clauses could be expanded in a gradual manner. In terms of the “boxes” (rules) idea discussed in Figure 1.7, this means that now the boxes would cover more than just a single negative example. One could keep removing such items off the definition of the clause until positive examples are covered too or some threshold value is reached. This idea can be explored from various implementation points of view but the end result is usually a rather large number of clauses.

Even after the above modification, the derived system of clauses may not be the best. There is no attention paid to the number of clauses derived to be small or, ideally, minimal. The strategy discussed next was first proposed in [Triantaphyllou, *et al.*, 1994] and [Triantaphyllou, 1994] and provides a greedy algorithm for achieving this goal. That approach is conceptually very close to Occam’s razor and computationally superior to the SAT approach discussed in the previous two sections.

```

Input: Training data sets  $E^+$  and  $E^-$ 
 $i = 0$ ;  $C = \emptyset$ ; {initializations}
DO WHILE ( $E^- \neq \emptyset$ )
    Step 1:  $i \leftarrow i + 1$ ;
    Step 2: Find a clause  $c_i$  which accepts all members of  $E^+$  while it
               rejects as many members of  $E^-$  as possible;
    Step 3: Let  $E^-(c_i)$  be the set of members of  $E^-$  which are rejected
               by  $c_i$ ;
    Step 4: Let  $C \leftarrow C \wedge c_i$ ;
    Step 5: Let  $E^- \leftarrow E^- - E^-(c_i)$ ;
REPEAT;
Output: A CNF expression  $C$  which accepts all examples in set  $E^+$  while it
               rejects all examples in set  $E^-$ 

```

Figure 2.1. The One Clause At a Time (OCAT) Approach (for the CNF case).

As mentioned in the previous section, the problem of deriving a Boolean function from sets of observations has been extensively studied in the literature. In our setting each example is a binary vector of size n (number of binary attributes). The proposed *One Clause At a Time (or OCAT)* approach is based on a greedy algorithm. It uses as input data the two collections of disjoint positive and negative examples. It determines a set of CNF clauses that, when taken together, reject all the negative examples and each of them accepts all the positive examples.

The OCAT approach is sequential. In the first iteration it determines a clause in CNF form (in the current implementation) that accepts all the positive examples in the E^+ set while it rejects as many negative examples in the current E^- set as possible. In the second iteration it performs the same task using the original E^+ set but the current E^- set has only those negative examples that have not been rejected by any clause so far. The iterations continue until a set of clauses is constructed which reject all the negative examples. Figure 2.1 summarizes the iterative nature of the OCAT approach.

The core of the OCAT approach is Step 2 in Figure 2.1. The way this step is defined in Figure 2.1, implies the solution of an optimization problem. This is how the number of the inferred clauses could be controlled and not allowed to increase too much. In the next section a branch-and-bound (B&B)-based algorithm is presented that solves the problem posed in Step 2. Another faster B&B algorithm is presented in Chapter 3. However, the first B&B algorithm is presented to motivate the introduction of the second one. A fast heuristic for solving the problem posed in Step 2 of the OCAT approach is described in Chapter 4. The OCAT approach returns the set of desired clauses as set C .

For the DNF case one needs to modify Step 2 by deriving a clause which rejects all the negative examples while it accepts as many positive examples in the current version of the E^+ set as possible. Next, one needs to update the set of the positive examples (in modified Steps 3 and 5) by keeping only those examples which have not been accepted so far and repeat the loop. Step 4 needs to be modified too so the

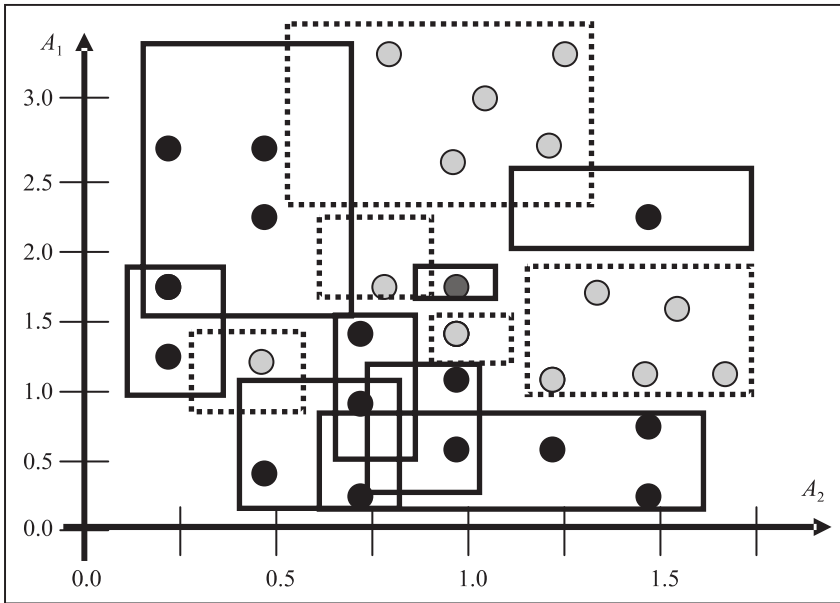


Figure 2.2. Some Possible Classification Rules for the Data Depicted in Figure 1.4.

derived system is a disjunction of conjunctions. The process is terminated when no positive examples are left in the E^+ set.

Next, we will try to get an idea about the difficulty of this Boolean function inference problem. Suppose that a learning problem involves a total of m training examples each of which is a binary vector of size n . Then, as Section 5.4 proves, there are 2^L , where $L = 2^n - m$, different Boolean functions which satisfy the requirements of these training data. This is an astronomically large number of possible solutions. The illustrative example later in Section 5.5 attempts to give a practical feeling of how incredibly large this solution space can be even for trivially small size learning problems. Thus, which Boolean function should one try to determine out of this huge number of possible solutions? The OCAT approach uses the greedy algorithm described in Figure 2.1 which is based on what is called the *principal of maximum simplicity*. This principle is best expressed by Occam's razor described in Section 1.4.2 as OCAT tries to infer a Boolean function of minimal or near-minimal number of clauses.

If one revisits the ideas behind Figure 1.7 (which is repeated here as Figure 2.2) about the two sets of boxes (rules) which cover the two groups of observations, then the OCAT approach solves a type of a *set covering problem*. Furthermore, it has the following interpretation.

Suppose that we start with the task of first covering the solid black points in Figure 2.2. We would like to derive a set of solid boxes. Such boxes correspond to classification rules which, as was shown in Section 1.4.2, are nothing but DNF expressions, although not defined on binary data. Then, according to the OCAT

algorithm, and for the DNF case, the first step is to determine a box (rule) which covers the largest concentration of solid black points without covering any of the gray points. Next, determine a second box which covers the second largest concentration of solid black points, also without covering any of the gray points. We repeat this step successively, until a set of boxes is derived which, when are taken together, cover all the solid black points without covering any of the gray ones.

This set of boxes is the “solid black” set of classification rules. In an analogous manner, the “gray” (or “dotted”) set of classification rules can be derived as well. In the following section a strategy is presented on how to classify new examples as being in either class or whether they should be deemed as undecidable (i.e., unclassifiable) cases.

Next, suppose that the cardinality (size) of the set of negative examples E^- is equal to m_2 . Then, the following theorem [Triantaphyllou, Soyster, and Kumara, 1994] states a critical property of the OCAT approach.

Theorem 2.1. *The OCAT approach terminates within m_2 iterations.*

Proof. From Section 2.5 it follows that it is always possible to construct a clause C_α that rejects only one negative example while it accepts any other possible example. At worst, Step 2 of the OCAT approach could propose a clause that rejects only one negative example at a given iteration. Therefore, the maximum number of iterations of the OCAT approach is m_2 . ■

In Sections 2.6 and 2.7 we discussed a Boolean inference algorithm based on a satisfiability (SAT) formulation. In the above version of the OCAT approach, Boolean functions are derived in CNF. The two approaches have a major difference.

The OCAT approach, as defined in Figure 2.1, attempts to minimize the number of disjunctions in the proposed CNF system. However, the SAT approach *pre-assumes* a given number, say k , of conjunctions in the DNF (or disjunctions in the CNF) system to be inferred and solves an SAT problem. If this SAT problem is infeasible, then the conclusion is that there is no DNF system which has k or fewer conjunctions and satisfies the requirements imposed by the examples. It should be emphasized here that it is not very critical whether an inference algorithm determines a CNF or DNF system (i.e., CNF or DNF Boolean function). As shown in [Triantaphyllou and Soyster, 1995b] and also presented in detail in Chapter 7, either a CNF or DNF system can be derived by using *either algorithm*.

2.9 A Branch-and-Bound Approach for Inferring a Single Clause

Branch-and-bound (B&B) is a search strategy which can be used to solve a wide spectrum of problems. It takes different forms depending on the specific problem under consideration. For Step 2 of the OCAT approach (for the CNF case), a B&B approach is given in [Triantaphyllou, Soyster, and Kumara, 1994]. It can be best described via an illustrative example. Suppose that the following are the two sets

(it is assumed that $n = 4$, i.e., the system involves 4 attributes) E^+ and E^- with the positive and negative examples of cardinality m_1 and m_2 , respectively.

$$E^+ = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad E^- = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}.$$

We number the positive examples as (1, 2, 3, 4) and the negative examples as (1, 2, 3, 4, 5, 6). For instance, the set of the negative examples {1, 3} means the set of the first and the third negative examples (i.e., vectors (1, 0, 1, 0) and (1, 1, 1, 1), respectively). The B&B approach will determine a single clause (in CNF) that accepts all the positive examples in the E^+ set, while rejecting as many negative examples from the current E^- set as possible. Before proceeding with the description of the B&B approach, it is instructive to compare it with a complete enumeration methodology (or brute force approach).

Consider the first positive example (0, 1, 0, 0). One can observe that in order to accept this positive example at least one of the four attributes A_1, A_2, A_3, A_4 must be specified as follows: ($A_1 = \text{false}$, i.e., $\bar{A}_1 = \text{true}$), ($A_2 = \text{true}$), ($A_3 = \text{false}$, i.e., $\bar{A}_3 = \text{true}$), and ($A_4 = \text{false}$, i.e., $\bar{A}_4 = \text{true}$). Hence, any valid CNF clause must include at least one of the following attributes: $\bar{A}_1, A_2, \bar{A}_3$, or \bar{A}_4 . Similarly, the second positive example (1, 1, 0, 0) implies that any valid CNF clause must include at least one of the following attributes: A_1, A_2, \bar{A}_3 , or \bar{A}_4 . In this manner, it can be concluded that any valid CNF clause must include at least one attribute as specified from each of the following four sets:

$$\begin{aligned} &\{\bar{A}_1, A_2, \bar{A}_3, \bar{A}_4\}, \\ &\{A_1, A_2, \bar{A}_3, \bar{A}_4\}, \\ &\{\bar{A}_1, \bar{A}_2, A_3, A_4\}, \text{ and} \\ &\{A_1, \bar{A}_2, \bar{A}_3, A_4\}. \end{aligned}$$

As mentioned in the previous section, this is a special case of the set covering problem which we denote as the *minimum cardinality problem* (or MCP). Let $|s|$ denote the cardinality of a set s . For the clause inference problem, the corresponding MCP problem takes the following general form:

Problem MCP (the initial formulation):

$$\text{minimize } \left| \bigcup_{i=1}^{m_1} \beta_i \right|$$

Subject to:

$$\beta_i \in B_i, \text{ for } i = 1, 2, 3, \dots, m_1,$$

where the sets B_i are defined next.

Table 2.3. The $NEG(A_k)$ Sets for the Illustrative Example.

Attribute	Set of Negative Examples	Attribute	Set of Negative Examples
A_1	$NEG(A_1) = \{1, 3, 5, 6\}$	\bar{A}_1	$NEG(\bar{A}_1) = \{2, 4\}$
A_2	$NEG(A_2) = \{3, 6\}$	\bar{A}_2	$NEG(\bar{A}_2) = \{1, 2, 4, 5\}$
A_3	$NEG(A_3) = \{1, 3, 6\}$	\bar{A}_3	$NEG(\bar{A}_3) = \{2, 4, 5\}$
A_4	$NEG(A_4) = \{2, 3\}$	\bar{A}_4	$NEG(\bar{A}_4) = \{1, 4, 5, 6\}$

The MCP formulation for the current clause inference problem (in CNF) is developed as follows. Define as $NEG(A_k)$ the set of the negative examples which are accepted by a clause when the attribute A_k is included in that clause. For the illustrative example in this section the $NEG(A_k)$ sets are presented in Table 2.3.

In the light of the definition of the $NEG(A_k)$ set and the $ATTRIBUTES(\alpha)$ set (as defined in Section 2.5), the sets B_i in problem MCP are defined as follows:

$$B_i = \{NEG(A_k), \text{ for each } A_k \in ATTRIBUTES(\alpha_i)\},$$

where α_i is the i -th positive example in E^+ .

Therefore, the previous minimization problem takes the following more precise form:

Problem MCP (more detailed formulation):

$$\text{Minimize } \left| \bigcup_{i=1}^{m_1} \beta_i \right| \quad (2.3)$$

Subject to:

$$\beta_i \in B_i, \text{ for } i = 1, 2, 3, \dots, m_1,$$

where $B_i = \{NEG(A_k), \text{ for each } A_k \in ATTRIBUTES(\alpha_i)\}$, and α_i is the i -th positive example in E^+ .

By using the data presented in Table 2.3, formulation (2.3) takes the following form for the case of the current illustrative example:

$$\text{Minimize } \left| \bigcup_{i=1}^4 \beta_i \right|$$

Subject to:

$$\beta_1 \in B_1, \text{ where } B_1 = \{\{2, 4\}, \{3, 6\}, \{2, 4, 5\}, \{1, 4, 5, 6\}\},$$

$$\beta_2 \in B_2, \text{ where } B_2 = \{\{1, 3, 5, 6\}, \{3, 6\}, \{2, 4, 5\}, \{1, 4, 5, 6\}\},$$

$$\beta_3 \in B_3, \text{ where } B_3 = \{\{2, 4\}, \{3, 6\}, \{1, 2, 4, 5\}, \{1, 3, 6\}, \{2, 3\}\},$$

$$\beta_4 \in B_4, \text{ where } B_4 = \{\{1, 3, 5, 6\}, \{1, 2, 4, 5\}, \{2, 4, 5\}, \{2, 3\}\}.$$

An *exhaustive enumeration* approach to solve this MCP problem is to construct a tree that has nodes arranged in $4(= m_1)$ levels. In the description of the search that follows, we call these levels *stages*. These levels correspond to the four positive examples enumerated as $\{1, 2, 3, 4\}$ in E^+ . Each interior node (i.e., a node with descendants), say at level h (where $1 \leq h < 4$), is connected to n nodes in the next higher level via n arcs. These n arcs represent the attributes that are true at the h -th positive example (i.e., the members of the set $ATTRIBUTES(\alpha_h)$, where α_h is the h -th positive example), as described in Section 2.5. The nodes (or *search states*) in this tree represent sets of negative examples. In our illustrative example these are subsets of the set $\{1, 2, 3, 4, 5, 6\}$.

For instance, the state $\{2, 3, 5\}$ refers to the second, third, and fifth negative examples in the set E^- . The set of negative examples that corresponds to a node (state) is the set of all the negative examples accepted by the attributes that correspond to the arcs that connect that node with the root node. That is, if one is at node (search state) Y_k and one follows the arc that corresponds to attribute A_i , then the resulting state, say Y_L , is

$$Y_L = Y_k \cup NEG(A_i).$$

If the above strategy is followed, then the current illustrative example would create $4 \times 4 \times 4 \times 4 = 256$ terminal nodes and, in the general case, n^{m_1} terminal nodes (where $m_1 = |E^+|$). Then, a clause which accepts all the positive examples and rejects as many negative examples as possible can be found by simply selecting a terminal node that corresponds to a search state with the minimum cardinality. This is true because such a state accepts the *minimum* number (or equivalently, rejects the *maximum* number) of negative examples.

Apparently, an exhaustive enumeration strategy is impractical. This is true because an exhaustive enumeration would require one to construct a search tree with n^{m_1} different terminal nodes (final states). However, this B&B approach, which is based on the previous tree, is much faster because it is capable of *pruning* this tree rather efficiently. As is explained next, each node of the tree is examined in terms of two tests. If any of these two tests succeeds, then that node is *fathomed* and it is not expanded further.

The tree of this search is shown in Figure 2.3. Consider the two nodes which correspond to the two states $\{2, 4\}$ and $\{2, 4, 5\}$ in the second stage of the search tree (see also Figure 2.3). Clearly, the states that correspond to the leaves (terminal nodes) that have the state $\{2, 4, 5\}$ as an ancestor are going to have *at least as many* members (i.e., negative examples) as the states of the leaves (terminal nodes) that have as ancestor the state $\{2, 4\}$. This is true because subsequent states are derived by performing union operations on these two states with the same sets. Therefore, if at any stage of building the search tree there is a state that has another state (in the current stage) as a subset, then that state (node) can be fathomed without eliminating any optimal solutions. This characterization of the states is formalized by the following definitions of dominated and undominated states, which is derived from the above discussion.

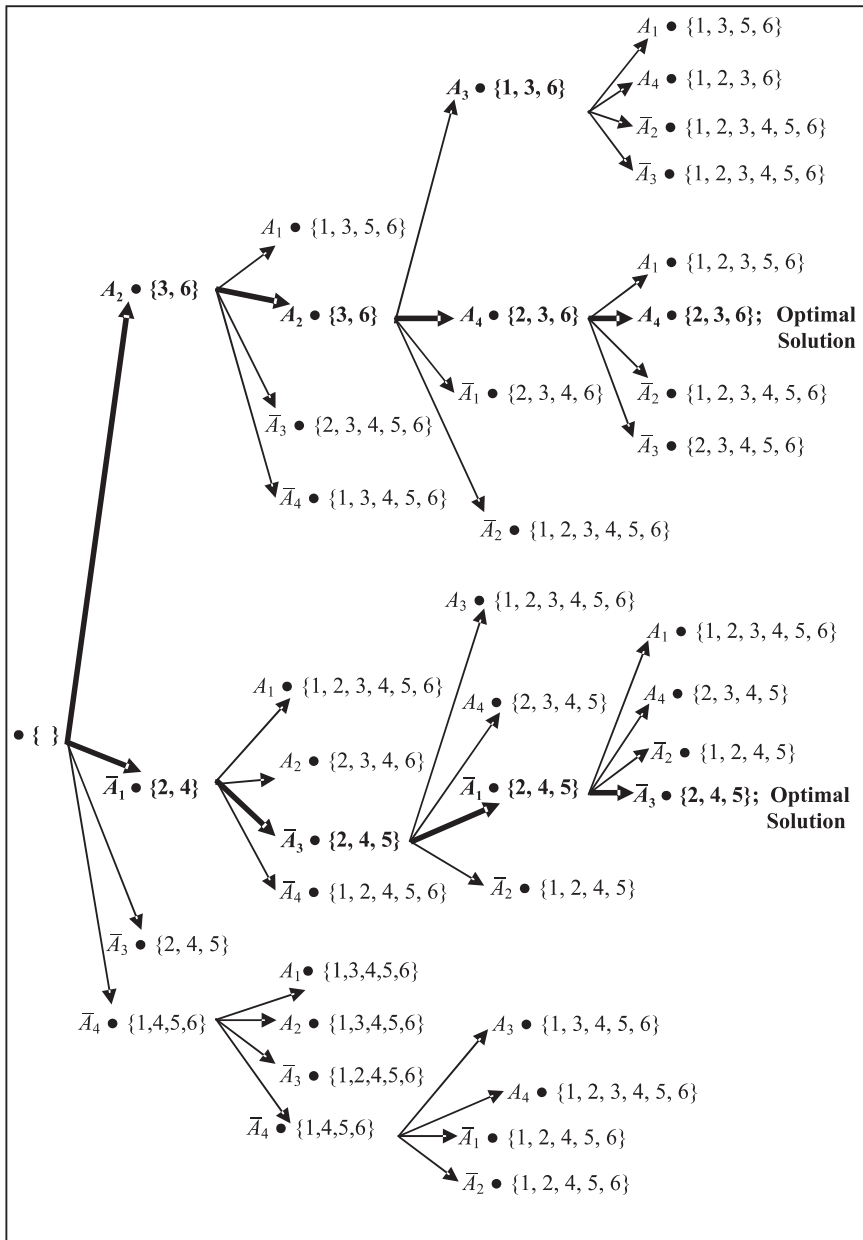


Figure 2.3. The Branch-and-Bound Search for the Illustrative Example.

Definition 2.1. A state S_k is a **dominated state** if there is another state S_j in the same stage which is a **proper subset** of S_k , i.e., if $S_i \subset S_k$. Otherwise, the state S_k is an **undominated state**.

The notion of dominated states leads to an important simplification of the MCP problem. Define as MCP' the problem derived from MCP when all dominated states are eliminated.

Problem MCP' :

$$\text{Minimize } \left| \bigcup_{i=1}^{m_1} \beta_i \right|$$

Subject to:

$$\beta_i \in B'_i, \text{ for } i = 1, 2, 3, \dots, m_1,$$

where B'_i (for $i = 1, 2, 3, \dots, m_1$) is the set that has as members only the *undominated* members of the set B_i .

Then, the previous definitions and discussion about dominated and undominated states lead to the following theorem [Triantaphyllou, Soyster, and Kumara, 1994]:

Theorem 2.2. *An optimal solution to MCP' is also optimal to MCP.*

The following corollary is a direct implication of Theorem 2.2:

Corollary 2.1. *The optimal solutions of the original MCP problem, given as (2.3), and the previous MCP' problem are identical.*

The previous corollary can be used for *problem preprocessing*. That is, when an MCP problem formulated as (2.3) is given, then it is beneficial to first transform it to the problem MCP' . In this way, the number of options (arcs in the B&B search graph) available at each node (search state) of the search graph will be the same or *smaller* than in the original MCP problem. Clearly, this means that the search can be done faster than in the original MCP problem.

The states in the last stage (i.e., the leaves of the tree) with the minimum number of negative examples indicate an optimal solution (see also Figure 2.3). In this example there are two such minimum size states. These are the states $\{2, 3, 6\}$ and $\{2, 4, 5\}$. The first optimal state (i.e., $\{2, 3, 6\}$) is derived from the clause $(A_2 \vee A_4)$. This is true because the attributes A_2 and A_4 are the only attributes (as indicated by the B&B search) which are involved in the decisions that generate the state $\{2, 3, 6\}$. Similarly, the second optimal state (i.e., $\{2, 4, 5\}$) is derived from the clause $(\bar{A}_1 \vee \bar{A}_3)$.

Next we discuss some other ways for making this B&B search (and possibly other B&B algorithms which share similar principles) even more efficient. One way to do so for this B&B formulation is to keep in memory only the nodes (states) of the current level (stage). Then, when an optimal state S is determined at the last stage, the optimal clause can be found by simply including in the definition of the current clause all the attributes along the path of the arcs which connect the optimal node with the root node.

Note that the optimal solution $(A_2 \vee A_4)$ does not reject the second, third, and sixth of the current negative examples in E^- . Hence, the remaining negative examples are

$$E^- = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}.$$

Similarly, the second OCAT iteration, when applied to the E^+ set and the new E^- set, yields the clause $(\bar{A}_2 \vee \bar{A}_3)$. Now the remaining negative examples are

$$E^- = [0 \ 0 \ 0 \ 1].$$

Iterating further, the third OCAT iteration yields the clause $(A_1 \vee A_3 \vee \bar{A}_4)$. That is, the CNF clauses which are generated from the original E^+ and E^- training examples are as follows:

Clause 1 : $(A_2 \vee A_4)$

Clause 2 : $(\bar{A}_2 \vee \bar{A}_3)$

Clause 3 : $(A_1 \vee A_3 \vee \bar{A}_4)$.

Thus, the inferred Boolean function is

$$(A_2 \vee A_4) \wedge (\bar{A}_2 \vee \bar{A}_3) \wedge (A_1 \vee A_3 \vee \bar{A}_4).$$

It can be easily verified that the previous three clauses, when taken together, reject all the negative examples in E^- . Moreover, each of the three clauses accepts all the positive examples in E^+ . That is, this function satisfies the desired requirements.

This is the Boolean function derived from the original positive and negative training examples. Thus, we will call it the “positive” Boolean function or just the “positive” system. Next, one can treat the original negative examples as positive and the original positive examples as negative and apply the OCAT approach with the previous B&B algorithm on this reversed set of data. Then, a “negative” system can be derived in a similar manner. These two systems, that is, the “positive” and the “negative” system, correspond to the idea of the “solid” and “dotted” rules depicted in Figure 1.7 (or, equivalently, in Figure 2.2). They together can be used to classify new examples of unknown class value.

Given these two systems, and a new example of unknown class value, then the following four scenarios are possible when the new example is classified by these two systems:

- 1) It is accepted by the “positive” system and rejected by the “negative” system. Then, this new example would be characterized as a positive one.
- 2) It is accepted by the “negative” system and rejected by the “positive” system. Then, this new example would be characterized as a negative one.
- 3) It is accepted by both the “positive” and the “negative” system. Then, this new example would be characterized as a *do not know* case (i.e., as undecidable/unclassifiable due to limited information).
- 4) It is rejected by both the “positive” and the “negative” system. Then, this new example would be characterized as a *do not know* case (i.e., as undecidable/unclassifiable due to limited information).

Please recall that the rules derived this way correspond to CNF expressions.

Regarding the algorithmic steps of the previous B&B approach, there is another observation that allows for further reduction on the number of states in the B&B search. Suppose that it is known (possibly via a heuristic) that one of the terminal states in the B&B search (not necessarily an optimal one) has k elements. Then, at any stage of the B&B approach, all states which have more than k elements can be deleted from further consideration. This is a valid step because any descendent of a state may only get larger at subsequent stages. This observation is summarized in the following theorem [Triantaphyllou, Soyster, and Kumara, 1994]:

Theorem 2.3. *Suppose some feasible solution to MCP (or MCP') has cardinality k . Then, an optimal solution to a modified B&B search in which all states that have more than k members are deleted, is also optimal for MCP (or MCP').*

Corollary 2.2. *The optimal solutions of the original MCP problem, given as (2.3), and the following problem are identical.*

$$\text{Minimize } \left| \bigcup_{i=1}^{m_1} \beta_i \right|$$

Subject to:

$$\beta_i \in B'_i, \text{ for } i = 1, 2, 3, \dots, m_1,$$

where B'_i (for $i = 1, 2, 3, \dots, m_1$) is the set that has as members only the members of the original set B_i which have *less than or equal* to k members (defined as above).

2.10 A Heuristic for Problem Preprocessing

The last corollary can be used for *problem preprocessing*. That is, when an MCP problem is formulated as (2.3), then it is a good idea first to run a heuristic (as will be described next) that very quickly yields a good feasible solution of size k (i.e., k is small) to the original MCP problem. When a value for k is available, the B&B search does not need to expand nodes in the search graph that have cardinality greater than k . This is true even for nodes that correspond to undominated states. In this way, the number of nodes to be expanded in the B&B search tree will, in general, be *smaller* than those in the original MCP problem. This step has the potential to expedite the B&B search.

Theorem 2.3 can further improve the performance of the proposed B&B search. When the B&B search is performed, the number of states at each stage (i.e., level of the search tree) may also increase dramatically. Therefore, the time and memory requirements of the search may increase dramatically. An efficient way to overcome this complication is to run the B&B search in two or more phases. In the first phase the B&B search is applied by allowing up to a small number, say 5, of states (i.e., nodes in the search tree) to be considered at any stage (i.e., level of the search tree).

These 5 states are the ones with the smallest cardinalities. That is, if more than 5 states (nodes) are formed at any stage, then only the 5 states with the smallest cardinalities will be considered for the next stage. This type of search is used in the AI literature often and is called *beam search* (see, for instance, [Dietterich and Michalski, 1981]).

Since up to 5 states are allowed to be considered at any stage of the B&B search and the number of stages is equal to the number of positive examples, it follows that the first phase will terminate quickly. Furthermore, the terminal nodes (final states) of the search tree will tend to represent states which have a tendency to have small cardinalities. This is expected to be the case because at each stage only the 5 states with the smallest cardinalities are considered (any ties are broken arbitrarily).

Suppose that in the first phase of the B&B process more than 5 states were generated at some stage. Let k be the cardinality of the smallest state that is dropped from further consideration due to the upper limit of 5 states per stage. Then, if one of the terminal nodes has cardinality *less than* k , then one can conclude that this node (state) represents an optimal solution. This is true because in this case none of the deleted states could lead to a terminal state with cardinality less than k . If there is no terminal state with cardinality less than k , then a terminal node (search state) with the minimal cardinality represents a potentially good feasible solution which may or may not be optimal. It should be emphasized here that by an optimal solution we mean the one that represents a single clause in CNF (i.e., a single disjunction) which accepts all the positive examples in E^+ while it rejects as many negative examples in the current E^- set as possible.

If after the first phase optimality is not provable, then the second phase is initiated. In the second phase, the B&B process is repeated with a higher limit, say 20, states per stage. As in the first phase, these 20 states are the states with the 20 smallest cardinalities. Suppose that L is the cardinality of the best solution obtained in the first phase. Then in the second phase, Theorem 2.3 is applied by eliminating any state that has cardinality greater than L . However, memory limitations may prohibit this B&B search from reaching an optimal solution. It should be stated here that if a too large number of states were allowed to be considered at any stage, then the B&B approach would take excessive time in ranking these states. The previous limit of 20 states was empirically found to be a reasonable choice.

As was done in the first phase, if more than 20 states are generated at any stage, then only 20 states are allowed at each stage. Similarly to the first phase, let k be the cardinality of the smallest state that was dropped from further consideration due to the upper limit of 20 states per stage. Then, if one of the terminal nodes has cardinality *less than* k , one can conclude that this node (state) represents an optimal solution. Otherwise optimality is not provable. In this case one may want to proceed with a third phase, or a fourth phase until optimality is eventually reached.

Some computational experiments indicate that Theorems 2.2 and 2.3 provide a rather efficient way for keeping the states at each stage in a manageable number and the resulting CPU requirements are dramatically reduced. For instance, a case with n equal to 10, 50 positive examples, and 170 negative examples required more than 1,100 CPU seconds on an IBM ES/3090-600S machine (Penn State's mainframe

computer in the 1980s and 1990s) running an integer programming implementation of the OCAT approach by using the MPSX software. However, the same problem took less than 30 CPU seconds with the proposed B&B formulation. Other similar comparisons also demonstrated significant improvement in time performance for this B&B approach.

2.11 Some Computational Results

In order to gain some computational experience with the OCAT approach and this B&B formulation, some random problems were generated and tested. The derived computational results are depicted in Table 2.4. For these problems, n , the number of attributes, was set equal to 30. First a set of 40 random clauses (disjunctions) was generated (the number 40 is arbitrary). Each such clause included, on the average, 5 attributes (as was the case with the experiments reported in [Hooker, 1988b]). The range of the number of variables per clause was from 1 to 10. Next, a collection E^o of random examples was generated. In these experiments we generated groups of 100, 200, 300, . . . , 1,000 random examples.

Each such random example was classified, according to the previous 40 clauses, either as a positive or as a negative example. With 40 clauses, this process resulted in more negative than positive examples. Because the stages in the B&B algorithm correspond to positive examples, problems with higher percentages of positive examples would demand more CPU time.

Next, the OCAT approach was applied on the previous positive and negative examples. The computational results are shown in Table 2.4. In this table the number of clauses derived by OCAT is denoted as S . The CPU time of the OCAT approach was recorded as well. This simulation program was written in the PL/I programming language and run on an IBM ES/3090-600S computer.

Each entry in Table 2.4 represents the performance of a single test problem, rounded to the nearest integer. Recall that $|s|$ indicates the *size* (or cardinality) of a set s . The computational results in Table 2.4 strongly suggest that the B&B approach is computationally tractable. For instance, no test problem took more than 836 CPU seconds (with an average of 96.17 CPU seconds). As was anticipated, the number of clauses created by this B&B search increases with the number of input examples.

It is also interesting to observe the behavior of the CPU time used by OCAT under the B&B formulation. Since the number of stages in the B&B search is equal to the number of positive examples, the CPU time increases with the size of the set of the positive examples. Furthermore, the total number of examples $|E^o|$ is critical too.

In these test problems the B&B formulation was applied as follows. During the first phase up to 5 states were allowed. If after the final stage optimality was not proved, then the best (i.e., the one with the smallest cardinality) solution available at this point was kept and the B&B approach was repeated by allowing up to 20 B&B states per stage (20 was an upper limit for memory considerations). These 20 states were selected as follows. If more than 20 B&B states were generated at some

Table 2.4. Some Computational Results When $n = 30$ and the OCAT Approach Is Used.

$ E^0 $	$ E^+ $	$ E^- $	S	Time	$ E^0 $	$ E^+ $	$ E^- $	S	Time
100	9	91	4	2	400	10	390	6	10
100	5	95	4	2	400	7	393	6	10
100	15	85	4	7	400	36	364	13	282
100	7	93	4	6	400	47	353	6	97
100	3	97	4	1	400	49	351	12	400
100	8	92	4	2	400	15	385	5	7
100	7	93	4	2	400	5	395	5	3
100	1	99	4	1	400	17	383	6	23
100	7	93	4	3	400	16	384	6	8
100	5	95	4	3	500	35	465	12	194
200	5	195	4	2	500	16	484	5	38
200	2	198	5	1	500	7	493	6	15
200	18	182	5	18	500	34	466	7	73
200	6	194	5	2	500	13	487	6	8
200	1	199	4	1	500	20	480	5	19
200	11	189	7	38	500	6	494	5	13
200	19	181	4	4	600	83	517	6	300
200	51	149	2	212	600	49	551	15	315
200	10	190	4	6	600	44	556	5	41
200	4	196	5	2	600	8	592	6	16
300	22	278	8	70	600	23	577	12	184
300	14	286	6	25	600	11	589	6	15
300	14	286	7	29	700	56	644	16	467
300	2	298	5	1	700	18	682	6	30
300	22	278	1	102	700	19	681	6	15
300	36	264	1	243	700	19	681	9	60
300	24	276	4	12	700	13	687	6	26
300	71	229	4	524	800	64	736	18	739
300	3	297	5	2	900	72	828	17	836
300	17	283	1	107	1,000	47	953	14	80

NOTE: The time is in seconds.

stage, then these states were ranked in descending order according to the number of elements (negative examples) per state and the top 20 states were selected.

In this second phase of the B&B search, the best solution found at the end of the first phase was used to reduce the state space at each stage (i.e., Theorem 2.3 was applied to reduce the memory requirements). The process was terminated after this second phase (in which the 20 states per stage limit was imposed) regardless of whether the current best solution could be confirmed as optimal or not. It should be mentioned here that if a higher limit of states was used, then the B&B approach takes more time because at each stage more states need to be considered. Some computational tests indicated that the previous limits (i.e., 5 and 20 states) seem to be reasonable. In 83% of the problems examined, confirmation of optimality could

Table 2.5. Some Computational Results When $n = 16$ and the SAT Approach Is Used.

$ E^0 $	Problem ID	K	Vars	Clauses	Time
100	16A1	15	1,650	19,368	2,039
100	16C1	20	1,580	16,467	758
200	16D1	10	1,230	15,901	1,547
200	16E1	15	1,245	14,766	2,156
300	16A2	6	1,602	23,281	608
300	16B1	8	1,728	24,792	78
400	16B2	4	1,076	16,121	236
400	16C2	4	924	13,803	521
400	16D2	4	836	12,461	544
400	16E2	4	532	7,825	376

NOTE: The time is in seconds.

Table 2.6. Some Computational Results When $n = 32$ and the SAT Approach Is Used.

$ E^0 $	Problem ID	k	Vars	Clauses	Time
50	32B1	3	228	1,374	5
50	32C1	3	225	1,280	24
50	32D1	4	332	2,703	66
50	32E1	3	222	1,186	8
100	32B2	3	261	2,558	57
100	32C2	3	249	2,182	9
100	32D2	4	404	5,153	178
100	32E2	3	267	2,746	10
150	32C3	3	279	3,272	14
200	32E3	3	330	5,680	133
250	32A1	3	459	9,212	177
250	32B3	3	348	5,734	190
300	32B4	3	381	6,918	259
300	32E4	3	387	7,106	277
400	32D3	4	824	19,478	1,227
400	32E5	3	450	9,380	390
1,000	32C4	3	759	20,862	155

NOTE: The time is in seconds.

be made. The low CPU times indicate that this B&B approach is rather efficient both in terms of CPU time and memory requirements.

Tables 2.5 and 2.6 present some computational results when the SAT approach is used. These results are the ones originally reported in [Kamath, Karmakar, *et al.*, 1992]. The CPU times are approximated to the closest integer value (in seconds). Those experiments were performed on a VAX 8700 running UNIX and that computer program was written in a combination of FORTRAN and C codes. The strategy of generating and testing the random problems is similar to the one mentioned in the

OCAT case. The only difference is that now the “hidden system” is in DNF form and consists of a few conjunctions (three to four). Please recall that in the OCAT case the “hidden logic” was a system in CNF form consisting of 40 randomly generated disjunctions.

The main point with the SAT results is that even for a small number of (positive and negative) examples the CPU times are rather high. This happens because the resulting SAT problems (as was indicated in formulas presented in Section 2.6) require many variables and clauses (as is shown under the “Vars” and “Clauses” columns in Tables 2.5 and 2.6). In Table 2.6 the test problems considered 32 attributes. The CPU times are smaller than the ones with 16 attributes (in Table 2.5) because now k was allowed to take much smaller values (3 or 4). In the 16-attribute case, however, k was allowed to take relatively speaking larger values (4 to 20).

In other words, the CPU requirements increase dramatically with the number of conjunctions assumed in the SAT formulation (denoted as k). This behavior is in direct agreement with the formulas mentioned in Section 2.6. However, if the original k value is *too small*, then infeasibility will be reached and the SAT problem needs to run again (with a larger k value) until a feasible solution is reached. This situation may increase the *actual* CPU requirements even more dramatically than the numbers shown in Tables 2.5 and 2.6.

2.12 Concluding Remarks

This chapter examined the problem of inferring a Boolean function from two sets of disjoint binary data. This is a fundamental problem in data mining and knowledge discovery and thus has received lots of attention by the scientific community. It may be hard to determine what is the best way to solve this problem. A computationally demanding approach is to formulate this problem as a satisfiability (SAT) problem. In this way a Boolean function of minimal size (in terms of the number of CNF or DNF clauses that comprise it) can be inferred. However, the computational cost may make the SAT approach impractical for large size problems.

We have chosen to work with CNF or DNF because any Boolean function can be transferred into these two forms [Blair, Jeroslow, and Lowe, 1986]. DNF expressions can be visualized easily as convex polyhedral shapes in the space of the attributes, while CNF expressions offer more intuitive formulation capabilities.

The approach proposed in this chapter helps to quickly infer a Boolean function in CNF or DNF. It is termed OCAT (for *One Clause At a Time*). It is a greedy approach for inferring a Boolean function by means of one clause at a time. A key step of the OCAT approach involves the solution of an optimization problem and thus the OCAT approach may lead to systems comprised of a few clauses. That would make it consistent with the desire to infer the system of *maximum simplicity* as Occam’s razor would dictate.

The previous optimization problem, as part of the OCAT approach, was solved according to a branch-and-bound (B&B) algorithm. This B&B algorithm can be expedited by exploiting certain key properties of the problem. These properties could

be used with other B&B algorithms in the future. Solving this problem is the foundation to inferring a Boolean function from training data.

At this point it should be pointed out that one may consider different approaches besides the one which tries to minimize the inferred CNF expression. One such reasonable approach would be to derive a Boolean function which would minimize a weighted average of the false-positive, false-negative, and undecidable rates. More on this idea is discussed later in Chapter 4, Section 4.5.

As stated earlier, this Boolean function inference problem is open-ended. One will always have a strong incentive to develop new methods that would be faster and methods to partition large-scale inference problems. The most important aspect is to have methods which would indeed capture the real essence of the input data, and thus the actual nature of the system or phenomenon that generated these data. Thus, this problem will always be one of keen interest to the research and practitioners communities in the field of data mining and knowledge discovery from data.

Appendix

The SAT Formulation and Solution for the Illustrative Example in Section 2.7 (for the CNF case)

```
! *****
!   This is the integer IP formulation (to run
!   on LINDO) for the illustrative example presented
!   in this chapter (for the CNF case).
!   The variable names are not identical, but they
!   closely reflect the notation used in this chapter.
! *****
! Note:
! We are interested in checking for feasibility.
! Thus, any objective function is applicable here.
! *****
```

MIN S11

ST

!.....

S11 + SP11 >= 1

S21 + SP21 >= 1

S12 + SP12 >= 1

S22 + SP22 >= 1

S13 + SP13 >= 1

S23 + SP23 >= 1

!.....

SS11 + SS12 + SSP13 >= 1

SS21 + SS22 + SSP23 >= 1

SSP11 + SSP12 + SSP13 >= 1

SSP21 + SSP22 + SSP23 >= 1

!.....

! NEXT ARE THE NEGATIONS

S11 + SS11 <= 1

S21 + SS21 <= 1

S12 + SS12 <= 1

S22 + SS22 <= 1

S13 + SS13 <= 1

S23 + SS23 <= 1

SP11 + SSP11 <= 1

SP21 + SSP21 <= 1

SP12 + SSP12 <= 1

SP22 + SSP22 <= 1

SP13 + SSP13 <= 1


```

SP23 + SSP23 <= 1
!.....
Z11 + Z12 >= 1
Z21 + Z22 >= 1
Z31 + Z32 >= 1
!.....
! NEXT ARE MORE NEGATIONS
Z11 + ZZ11 <= 1
Z12 + ZZ12 <= 1
Z13 + ZZ13 <= 1
Z21 + ZZ21 <= 1
Z22 + ZZ22 <= 1
Z23 + ZZ23 <= 1
Z32 + ZZ32 <= 1
Z31 + ZZ31 <= 1
!.....
SP11 + ZZ11 >= 1
S12 + ZZ11 >= 1
SP13 + ZZ11 >= 1
!.....
SP21 + ZZ12 >= 1
S22 + ZZ12 >= 1
SP23 + ZZ12 >= 1
!.....
SP11 + ZZ21 >= 1
S12 + ZZ21 >= 1
S13 + ZZ21 >= 1
!.....
SP21 + ZZ22 >= 1
S22 + ZZ22 >= 1
S23 + ZZ22 >= 1
!.....
S11 + ZZ31 >= 1
SP12 + ZZ31 >= 1
SP13 + ZZ31 >= 1
!.....
S21 + ZZ32 >= 1
SP22 + ZZ32 >= 1
SP23 + ZZ32 >= 1
!.....
END
INTEGER 40

```

This is the corresponding solution as generated by LINDO

NEW INTEGER SOLUTION OF .000000000 AT BRANCH 0 PIVOT 46
 LP OPTIMUM FOUND AT STEP 46
 OBJECTIVE VALUE = .000000000
 ENUMERATION COMPLETE. BRANCHES= 0 PIVOTS= 46

LAST INTEGER SOLUTION IS THE BEST FOUND
 RE-INSTALLING BEST SOLUTION...

OBJECTIVE FUNCTION VALUE

1) .000000000

VARIABLE	VALUE	REDUCED COST
S11	.000000	1.000000
SP11	1.000000	.000000
S21	1.000000	.000000
SP21	.000000	.000000
S12	1.000000	.000000
SP12	.000000	.000000
S22	.000000	.000000
SP22	1.000000	.000000
S13	1.000000	.000000
SP13	1.000000	.000000
S23	.000000	.000000
SP23	1.000000	.000000
SS11	1.000000	.000000
SS12	.000000	.000000
SSP13	.000000	.000000
SS21	.000000	.000000
SS22	1.000000	.000000
SSP23	.000000	.000000
SSP11	.000000	.000000
SSP12	1.000000	.000000
SSP21	1.000000	.000000
SSP22	.000000	.000000
SS13	.000000	.000000
SS23	.000000	.000000
Z11	1.000000	.000000
Z12	.000000	.000000
Z21	1.000000	.000000
Z22	.000000	.000000
Z31	.000000	.000000
Z32	1.000000	.000000

ZZ11	.000000	.000000
ZZ12	1.000000	.000000
Z13	.000000	.000000
ZZ13	.000000	.000000
ZZ21	.000000	.000000
ZZ22	1.000000	.000000
Z23	.000000	.000000
ZZ23	.000000	.000000
ZZ32	.000000	.000000
ZZ31	1.000000	.000000

ROW	SLACK OR SURPLUS	DUAL PRICES
2)	.000000	.000000
3)	.000000	.000000
4)	.000000	.000000
5)	.000000	.000000
6)	1.000000	.000000
7)	.000000	.000000
8)	.000000	.000000
9)	.000000	.000000
10)	.000000	.000000
11)	.000000	.000000
12)	.000000	.000000
13)	.000000	.000000
14)	.000000	.000000
15)	.000000	.000000
16)	.000000	.000000
17)	1.000000	.000000
18)	.000000	.000000
19)	.000000	.000000
20)	.000000	.000000
21)	.000000	.000000
22)	.000000	.000000
23)	.000000	.000000
24)	.000000	.000000
25)	.000000	.000000
26)	.000000	.000000
27)	.000000	.000000
28)	.000000	.000000
29)	1.000000	.000000
30)	.000000	.000000
31)	.000000	.000000
32)	1.000000	.000000
33)	.000000	.000000
34)	.000000	.000000

35)	.000000	.000000
36)	.000000	.000000
37)	.000000	.000000
38)	.000000	.000000
39)	.000000	.000000
40)	1.000000	.000000
41)	.000000	.000000
42)	.000000	.000000
43)	.000000	.000000
44)	.000000	.000000
45)	.000000	.000000
46)	.000000	.000000
47)	.000000	.000000
48)	.000000	.000000
49)	1.000000	.000000
50)	.000000	.000000
51)	.000000	.000000
52)	.000000	.000000

NO. ITERATIONS= 46

BRANCHES= 0 DETERM.= -1.000E 0

Data Mining and Knowledge Discovery via Logic-Based
Methods

Theory, Algorithms, and Applications

Triantaphyllou, E.

2010, XXXIV, 350 p. 91 illus., 9 illus. in color., Hardcover

ISBN: 978-1-4419-1629-7