Machine Learning Algorithms Inspired by the Work of Ryszard Spencer* Michalski

Krzysztof J. Cios[1] and Łukasz A. Kurgan[2]

[1] Virginia Commonwealth University, Richmond, USA, and IITiS,
Polish Academy of Sciences, Poland
[2] University of Alberta, Edmonton, Canada

Abstract. In this chapter we first define the field of inductive machine learning and then describe Michalski's basic AQ algorithm. Next, we describe two of our machine learning algorithms, the CLIP4: a hybrid of rule and decision tree algorithms, and the DataSqeezer: a rule algorithm. The development of the latter two algorithms was inspired to a large degree by Michalski's seminal paper on inductive machine learning (1969). To many researchers, including the authors, Michalski is a "father" of inductive machine learning, as Łukasiewicz is of multivalued logic (extended much later to fuzzy logic) (Łukasiewicz, 1920), and Pawlak of rough sets (1991). Michalski was the first to work on inductive machine learning algorithms that generate rules, which will be explained via describing his AQ algorithm (1986).

1 Introduction

Machine learning (ML) is meant that machines/computers perform the learning instead of humans. The broadest definition of ML algorithms concerns the ability of a computer program to improve its own performance, in some domain, based on the past experience. Another, more specific, definition of ML is an ability of a program to generate a new data structure, different from the structure of the original data, such as a (production) IF… THEN… rule generated from numerical and/or nominal data (Kodratoff, 1988; Langley, 1996; Mitchell, 1997, Cios et al., 2007). ML algorithms are one of many data mining tools used for building models of data. However, the advantage of inductive ML algorithms is that they are one of only a few tools capable of generating user-friendly models. Namely, they generate models of the data in terms of the IF…THEN… rules that can be easily analyzed, modified, and used for training/learning purposes. This is in contrast to "black box" methods, such as neural networks and support vector machines, which generate models that are virtually impossible to interpret. Therefore, inductive ML algorithms (and their equivalent: decision trees) are preferred over other methods in fields where a decision maker needs to understand/accept the generated rules (like in medical diagnostics).

* Professor Michalski, after delivering talk on artificial intelligence at the University of Toledo, Ohio, in 1986, at the invitation of the first author, explained the origin of his second name: Spencer. Namely, he used the right of changing his name while becoming a United States citizen and adopted it after the well-known philosopher Herbert Spencer.

J. Koronacki et al. (Eds.): Advances in Machine Learning I, SCI 262, pp. 49–74.
springerlink.com

Michalski was involved in the development of algorithms that address both the supervised and unsupervised learning. Here we are concerned mainly with the supervised learning, although we also briefly comment on his work on clustering (the key unsupervised method). The supervised learning, also known as learning from examples, happens when the user/teacher provides examples (labeled data points) that describe concepts/classes. Thus, any supervised learning algorithm needs to be provided with a training data set, *S*, that consists of *M* training data pairs, belonging to *C* classes:

$$S = \{(\mathbf{x}_i, c_j) \mid i = 1,...,M; j = 1,...,C\}$$

where $\mathbf{x}_i$ is an n-dimensional pattern vector, whose components are called features/attributes, and c_j is a known class.

The mapping function f: c = f(x), is not known and a learning algorithm aims at finding/approximating this function. The training set represents information about some domain with the frequently used assumption that the features represent only properties of the examples but not relationships between the examples. A supervised ML algorithm searches the space of possible hypotheses, H, for the hypothesis (one or more) that best estimates the function f. The resulting hypotheses, or concept descriptions, are often written in the form of IF… THEN… rules.

The key concept in inductive ML is that of a hypothesis that approximates some concept. An example of a concept is, say, the concept of a hybrid car. We assume that only a teacher knows the true meaning of a concept and describes it by means of examples given to a learner (in our case a ML algorithm) whose task is to generate hypotheses that best approximate the concept. The concept of a hybrid car can be provided in terms of input-output pairs such as (gas&electric engine, hybridcar), (very low gas consumption, hybridcar), etc. We often assume that the terms concept and hypothesis are equivalent (which is not quite correct since the learner receives from a teacher only a finite set of examples that describe the concept so the generated hypotheses can only approximate it). Since hypotheses are often described in terms of rules we also use the term rule (and Michalski's notion of a cover, defined later) to denote the hypothesis.

Any supervised inductive ML process has two phases:

- Learning phase, where the algorithm analyzes training data and recognizes similarities among data objects to build a model that approximates f,
- Testing phase, when the generated model (say, a set of rules) is verified by computing some performance criterion on a new data set, drawn from the same domain.

Two basic techniques for inferring information from data are deduction and induction. Deduction infers information that is a logical consequence of the information present in the data. It is provably correct if the data/examples describing some domain are correct. Induction, on the other hand, infers generalized information/knowledge from the data by searching for some regularities among the data. It is correct for the data but only plausible outside of the given data. A vast majority of the existing ML algorithms are inductive. Learning by induction is a search for a correct rule, or a set of rules, guided by training examples. The task of the search is to find hypotheses that best describe the concept. We usually start with some initial hypothesis and then search for one that covers as many input data points (examples) as possible. We say

that an example is covered by a rule when it satisfies all conditions of the IF… part of the rule. Still another view of inductive ML is one of designing a classifier, i.e., finding boundaries that encompass only examples belonging to a given class. Those boundaries can either partition the entire sample space into parts containing examples from one class only, and sometimes leave parts of the space unassigned to either of the classes (a frequent outcome).

A desirable characteristic of inductive ML algorithms is their ability (or inability) to deal with incomplete data. The majority of real datasets have records that include missing values due to a variety of reasons, such as manual data entry errors, incorrect measurements, equipment errors, etc. It is common to encounter datasets that have up to half of the examples missing some of their values (Farhanghfar et al., 2007). Thus, a good ML algorithm should be robust to missing values as well as to data containing errors, as they often have adverse effect on the quality of the models generated by the ML algorithms (Farhanghfar et al., 2008).

The rest of the chapter contains a review of Michalski's work in supervised learning and a review of our algorithms, which were inspired by his work, but first we briefly comment on Michalski's work in unsupervised learning. A prime example of unsupervised learning is clustering. However, there is a significant difference between the classical clustering and clustering performed within the framework of ML. The classical clustering is best suited for handling numerical data. Thus Michalski introduced the concept of conceptual clustering to differentiate it from classical clustering since conceptual clustering can deal with nominal data (Michalski, 1980; Fisher and Langley, 1986; Fisher, 1987). Conceptual clustering consists of two tasks: clustering itself which finds clusters in a given data set, and characterization (which is supervised learning) which generates a concept description for each cluster found by clustering. Conceptual clustering can be then thought of as a hybrid combining unsupervised and supervised approaches to learning; CLUSTER/2 by Michalski (1980) was the first well-known conceptual clustering system.

Table 1. Set of ten examples described by three features (F_1-F_3) drawn from two categories (F_4)

S	F_1	F_2	F_3	F_4 decision attribute
e_1	1	1	2	1
e_2	1	1	1	1
e_3	1	2	2	1
e_4	1	2	1	1
e_5	1	3	2	1
e_6	3	4	3	2
e_7	2	5	3	2
e_8	3	1	3	2
e_9	2	2	3	2
e_{10}	2	3	3	2

2 Generation of Hypotheses

The process of generating hypotheses is instrumental for understanding how inductive ML algorithms work. We first illustrate this concept by means of a simple example from which we will (by visual inspection) generate some hypotheses and later describe ML algorithms that do the same in an automated way. Let us define an *information system* (IS):

$$IS =< S, Q, V, f >$$

where

- S is a finite set of examples, $S = \{e_1, e_2, ..., e_M\}$ and M is the number of examples
- Q is a finite set of features, $Q = \{F_1, F_2, ..., F_n\}$ and n is the number of features
- $V = \cup V_{F_j}$ is a set of feature values where V_{F_j} is the domain of feature $F_j \in Q$
- $v_i \in V_{F_j}$ is a value of feature F_j
- $f = S \times Q \rightarrow V$ is an information function satisfying $f(e_i, F_i) \in V_{F_j}$ for every $e_i \in S$ and $F_j \in Q$

The set S is known as the *learning/training data*, which is a subset of the universe (that is known only to the teacher/oracle); the latter is defined as the Cartesian product of all feature domains V_{F_j} (j=1,2…n).

Now we analyze the data shown in Table 1 and generate a rule/hypothesis that describes class1 (defined by attribute F_4):

IF F_1=1 AND F_2=1 THEN class1 (or F_4=1)

This rule covers two (e_1 and e_2) out of five positive examples. So we generate another rule:

IF F_1=1 AND F_3=2 THEN class1

This rule covers three out of five positive examples, so it is better than the first rule; rules like this one are called strong since they cover a majority (large number) of positive (in our case class1) training examples. To cover all five positive examples we need to generate one more rule (to cover e_4):

IF F_1=1 AND F_3=1 THEN class1

While generating the above rules we paid attention so that none of the rules describing class1 covered any of the examples from class2. In fact, for the data shown in Table 1, this made it more difficult to generate the rules because we could have generated just one simple rule:

IF F_1=1 THEN class1

that would perfectly cover all class1 examples while not covering the class2 examples. The generation of such a rule is highly unlikely to describe any real data where hundreds of features may describe thousands of examples; that was why we have generated more rules to illustrate typical process of hypotheses generation.

As mentioned above, the goal of inductive ML algorithms is to automatically (without a human intervention) generate rules (hypotheses). After learning, the generated rules must be tested on unseen examples to assess their predictive power. If the rules fail to correctly classify (to calculate the error we assume that we know their "true" classes) a majority of the test examples the learning phase is repeated by using procedures like cross-validation. The common disadvantage of inductive machine learning algorithms is their ability to, often almost perfectly, cover/classify training examples, which may lead to the overfitting of data. A trivial example of overfitting would be to generate five rules to describe the five positive examples; the rules would be the positive examples themselves. Obviously, if the rules were that specific they would probably perform very poorly on new examples. As stated, the goodness of the generated rules needs to be evaluated by testing the rules on new data. It is important to establish a balance between the rules' generalization and specialization in order to generate a set of rules that have good predictive power. A more general rule (strong rule) is one that covers more positive training examples. A specialized rule, on the other hand, may cover, in an extreme case, only one example.

In the next section we describe rule algorithms also referred to as rule learners. Rule induction/generation is distinct from the generation of decision trees. While it is trivial to write a set of rules given a decision tree it is more complex to generate rules directly from data. However, the rules have many advantages over decision trees. Namely, they are easy to comprehend; their output can be easily written in the first-order logic format, or directly used as a knowledge base in knowledge-based systems; the background knowledge can be easily added into a set of rules; and they are modular and independent, i.e., a single rule can be understood without reference to other rules. Independence means that, in contrast to rules written out from decision trees, they do not share any common attributes (partial paths in a decision tree). Their disadvantage is that they do not show relationships between the rules as decision trees do.

3 Rule Algorithms

As already said, Michalski was the first one to introduce an inductive rule-based ML algorithm that generated rules from data. In his seminal paper (Michalski, 1969) he framed the problem of generating the rules as a set-covering problem. We illustrate one of early Michalski's algorithms, the AQ15, in which IF…THEN…rules are expressed in terms of variable-value logic (VL1) calculus (Michalski, 1974; Michalski et al. 1986). The basic notions of VL1 are that of a selector, complex, and cover. A selector is a relational statement:

$$(F_i \ \# \ v_i)$$

where # stands for any relational operator and v_i is one or more values from dom_i of attribute F_i.

A complex L is a logical product of selectors:

$$L = \cap \ (F_i \# v_i)$$

The cover, C, is defined as a disjunction of complexes:

$$C = \cup \ L_i$$

and forms the conditional part of a production rule covering a given data set.

Two key operations in the AQ algorithms are the generation of a *star* $G(e_i | E_2)$ and the generation of a *cover* $G(E_1 | E_2)$, where $e_i \in E_1$ is an element of a set E_1, and E_2 is another set such that $E_1 \cup E_2 = S$, where S is the entire training data set.

We use data shown in Table 1 to illustrate operation of the family of AQ algorithms.

First, let us give examples of information functions for data shown in Table1:

$$F_1 = 1$$
$$F_3 = 1 \text{ OR } 2 \text{ OR } 3.$$

The full form of the first information function is:

$$(F_1 = 1) \text{ AND } (F_2 = 2 \text{ OR } 3 \text{ OR } 4 \text{ OR } 5 \text{ OR } 1) \text{ AND } (F_3 = 1 \text{ OR } 2 \text{ OR } 3)$$

Similarly the second information function can be rewritten as:

$$(F_1 = 1 \text{ OR } 2 \text{ OR } 3) \text{ AND } (F_2 = 2 \text{ OR } 3 \text{ OR } 4 \text{ OR } 5 \text{ OR } 1) \text{ AND } (F_3 = 1 \text{OR } 2 \text{ OR } 3)$$

A function *covers* an example if it matches all the attributes of a given example, or, in other words, it evaluates to TRUE for this example. Thus, the information function $(F_1 = 1)$ covers the subset $\{e_1, e_2, e_3, e_4, e_5\}$, while the function $(F_3 = 1 \text{ OR } 2 \text{ OR } 3)$ covers all examples shown in Table 1.

The goal of inductive machine learning, in Michalski's setting, is to generate information functions while taking advantage of a decision attribute (feature F_4 in Table 1). The question is whether the information function, IF_B, generated from a set of training examples will be the same as a true information function, IF_A (Kodratoff, 1988). In other words, the question is whether the ML algorithm (B) can learn what only the teacher (A) knows. To answer the question let us consider the function:

$$IF_A : (F_1 = 3 \text{ OR } 2) \text{ AND } (F_2 = 1 \text{ OR } 2)$$

that covers subset $\{e_8, e_9\}$. Next, we generate, via induction, the following information function:

$$IF_B : (F_1 = 3 \text{ OR } 2) \text{ AND } (F_2 = 1 \text{ OR } 2) \text{ AND } (F_3 = 3)$$

which covers the same two examples, but IF_B is different from IF_A. IF_B can be rewritten as:

$$IF_B = IF_A \text{ AND } (F_3 = 3)$$

We say that IF_B is a specialization of IF_A (it is less general). So in this case the learner learned what the teacher knew (although in a slightly different form). Note that frequently this is not the case.

In order to evaluate the goodness of the generated information functions we use criteria such as the sparseness function, which is defined as the total number of examples it can potentially cover minus the number of examples it actually covers. The smaller the value of the sparseness function the more compact the description of examples. Let us assume that we have two particular information functions IF_1 and IF_2, such that IF_1 covers subset E_1 and the other covers the remaining part of the training data, indicated by subset E_2. If the intersection of sets E_1 and E_2 is empty then we say that these two functions *partition* the training data. The goal of AQ algorithms, as well as all ML algorithms, is to find such partitions. Assuming that one part of the training data represents positive examples, and the remaining part represents negative

examples, then IF_1 becomes the rule (hypothesis) covering all positive examples. To calculate the sparseness of a partition the two corresponding information sparsenesses can be added together and used for choosing among several generated alternative partitions (if they exist). Usually we start with the initial partition in which subsets E_1 and E_2 intersect and the goal is then to come up with information functions that result in the partition of training data. The task of the learner is to modify this initial partition so that all intersecting elements are incorporated into "final" subsets, say E_{11} and E_{22}, which form a *partition*:

$$E_{11} \cap E_{22} = \varnothing \quad \text{and} \quad E_{11} \cup E_{22} = S$$

Michalski et al. (1986) proposed the following algorithm.

Given: Two disjoint sets of examples

1. Start with two disjoint sets E_{01} and E_{02}. Generate information functions, IF_1 and IF_2 from them and generate subsets, E_1 and E_2, which they cover.
2. If sets E_1 and E_2 intersect then calculate differences between sets E_1 and E_2 and the intersecting set

   $E_p = E_1 - E_1 \cap E_2$
   $E_n = E_2 - E_1 \cap E_2$

   and generate corresponding information functions, IF_p and IF_n; otherwise we have a partition; stop.
3. For all examples e_i from the intersection do:
   create sets $E_p \cup e_i$ and $E_n \cup e_i$ and generate information functions for each, IF_{pi} and IF_{ni}
4. Check if (IF_p, IF_{ni}) and (IF_n, IF_{pi}) create partitions of S
   a) if they do, choose the better partition, say in terms of sparseness (they become new E_1 and E_2), go to step 1 and take the next example e_i from the intersection
   b) if not, go to step 2 and check another example from the intersection

Result: Partition of the two sets of examples.

This algorithm does not guarantee that all examples will be assigned to one of the two subsets if a partition is not found.

We illustrate this algorithm using data from Table 1.

1. Assume that the initial subsets are $\{e_1, e_2, e_3, e_4, e_5, e_6\}$ and $\{e_7, e_8, e_9, e_{10}\}$.

Notice that we are not as yet using a decision attribute/feature F_4. We will use it later for dividing the training data into subsets of positive and negative examples. We only try to illustrate how to move the intersecting examples so that the resulting subsets create a partition.

The information functions generated from these two sets are:

IF_1 : (F_1 = 1 OR 3) AND (F_2 = 1 OR 2 OR 3 OR 4) AND (F_3 = 1 OR 2 OR 3)
IF2 : (F_1 = 3 OR 2) AND (F_2 = 5 OR 2 OR 1 OR 3) AND (F_3 = 3)

Function IF_1 covers set $E_1 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_8\}$ and function IF_2 covers set $E_2 = \{e_7, e_8, e_9, e_{10}\}$.

2. Since $E_1 \cap E_2 = \{e_8\}$, which means that they intersect, hence we calculate:

$E_p = E_1 - \{e_8\} = \{e_1, e_2, e_3, e_4, e_5, e_6\}$
$E_n = E_2 - \{e_8\} = \{e_7, e_9, e_{10}\}$

and generate two corresponding information functions:

IF_p : (F_1 = 1 OR 3) AND (F_2 = 1 OR 2 OR 3 OR 4) AND (F_3 = 1 OR 2 OR 3)
IF_n : (F_1 = 2) AND (F_2 = 5 OR 2 OR 3 AND (F_3 = 3)

Note that IF_p is exactly the same as IF_1 and thus covers E_1, while IF_n covers only E_n.

3. Create the sums $E_p \cup e_i = E_1$ and $E_n \cup e_i = E_2$, where $e_i = e_8$, and generate the corresponding information functions. The result is $IF_{pi} = IF_1$ and $IF_{ni} = IF_2$.

4. Check if pairs (IF_p, IF_2) and (IF_n, IF_1) create a partition.

The first pair of information functions covers the subset $\{e_1, e_2, e_3, e_4, e_5, e_6, e_8\}$ and subset $\{e_7, e_8, e_9, e_{10}\}$. Since the two subsets still intersect, this is not a partition yet. The second pair covers the subsets $\{e_7, e_9, e_{10}\}$ and $\{e_1, e_2, e_3, e_4, e_5, e_6, e_8\}$; since they do not intersect, and sum up to S, this is a partition.

Note that in this example we have chosen the initial subsets arbitrarily and our task was just to find a partition. In inductive supervised ML, however, we use the decision attribute to help us in this task. We observe that we have generated information functions not by using any algorithm, but by visual inspection of Table 1.

The AQ search algorithm (as used in AQ15) is an irrevocable top-down search which generates a decision rule for each class in turn. In short, the algorithm at each step starts with selecting one positive example, called a seed, and generates all complexes (a star) that cover the seed but do not cover any negative examples. Then by using criteria such as the sparseness and the length of complexes (shortest complex first) it selects the best complex from the star, which is added to the current (partial) cover. The pseudocode, after Michalski et al. (1986), follows.

Given: Sets of positive and negative training examples
While partial *cover* does not cover all positive examples do:

1. Select an uncovered positive example (a seed)
2. Generate a star, that is determine maximally general complexes covering the seed and no negative examples
3. Select the best complex from the star, according to the user-defined criteria
4. Add the complex to the partial cover

While partial *star* covers negative examples do:

1. Select a covered negative example
2. Generate a partial star (all maximally general complexes) that covers the seed and excludes the negatie example
3. Generate a new partial star by intersecting the current partial star with the partial star generated so far
4. Trim the partial star if the number of disjoint complexes exceeds the predefined threshold, called maxstar (to avoid exhaustive search for covers which can grow out of control)

Result: Rule(s) covering all positive examples and no negative examples

Now we illustrate the generation of a cover using the decision attribute, F_4. However, we will use only a small subset of the training data set consisting of four examples: two positive (e_4 and e_5) and two negative (e_9 and e_{10}), to be able to show all the calculations. The goal is to generate a cover that covers all positive (class 1) examples (e_4 and e_5) and excludes all negative examples (e_9 and e_{10}). Thus, we are interested in generating a cover properly identifying subset $E_1 = \{e_4, e_5\}$ and rejecting subset $E_2 = \{e_9, e_{10}\}$; such a cover should create a partition of $S = \{e_4, e_5, e_9, e_{10}\}$.

Generation of a cover involves three steps:

For each positive example $e_i \in E_1$, where E_1 is a positive set:

1. Find $G(e_i \mid e_j)$ for each $e_j \in E_2$, where E_2 is a negative set
2. Find a star $G(e_i \mid E_2)$. It is THE conjunction of $G(e_i \mid e_j)$ terms found in step 1. When there is more than one such term (after converting it into a disjunctive form) select the best one according to some criteria, like the sparseness.
3. Finding a cover of all positive examples against all negative examples $G(E_1 \mid E_2)$. It is the disjunction of stars found in step 2. The final cover covers all positive examples and no negative examples.

Let us start with finding $G(e_4 \mid e_9)$. It is obtained by comparing the values of the features in both examples, skipping those which are the same, and making sure that the values of features in e_9 are different from those of e_4, and putting them in disjunction. Thus,

$$G(e_4 \mid e_9) = (F_1 \neq \text{black}) \text{ OR } (F_3 \neq \text{large})$$

Note that it is the most general information function describing e_4 since it makes sure that only example e_9 is not covered by this function.

Next, we calculate the star $G(e_i \mid E_2)$, for all $e_i \in E_1$, against all e_j from E_2. A star for e_i is calculated as the conjunction of all G(.)s and constitutes a cover covering e_i

$$G(e_i \mid E_2) = \cap G(e_i \mid e_j) \quad \text{for all } e_j \in E_2$$

Since we started with $e_i = e_4$ we will obtain a cover of e_4 against e_{10}, and combine it using the conjunction with the previous cover; this results in

$$G(e_4 \mid E_2) = ((F_1 \neq 2) \text{ OR } (F_3 \neq 3)) \text{ AND } ((F_1 \neq 2) \text{ OR } (F_2 \neq 3) \text{ OR } (F_3 \neq 3))$$

The expression is converted into the disjunctive form:

$$\begin{aligned} G(e_4 \mid E_2) = {} & ((F_1 \neq 2) \text{ AND } (F_1 \neq 2) \text{ OR } ((F_1 \neq 2) \text{ AND } (F_2 \neq 3)) \text{ OR} \\ & ((F_1 \neq 2) \text{ AND } (F_3 \neq 3)) \text{ OR } ((F_3 \neq 3) \text{ AND } (F_1 \neq 2)) \text{ OR} \\ & ((F_3 \neq 3) \text{ AND } (F_2 \neq 3)) \text{ OR}((F_3 \neq 3) \text{ AND } (F_3 \neq 3)) \end{aligned}$$

Next, by using various laws of logic it is simplified into:

$$\begin{aligned} G(e_4 \mid E_2) = {} & (F_1 \neq 2) \quad \text{OR} \quad (28) \\ & ((F_1 \neq 2) \text{ AND } (F_2 \neq 3)) \text{ OR} \quad (23) \\ & ((F_1 \neq 2) \text{ AND } (F_3 \neq 3)) \text{ OR} \quad (18) \\ & ((F_3 \neq 3) \text{ AND } (F_2 \neq 3)) \text{ OR} \quad (23) \\ & ((F_3 \neq 3)) \end{aligned}$$

and after calculating the sparseness (shown in parentheses) the best is kept:

$$G(e_4 \mid E_2) = (F_1 \neq 2) \text{ AND } (F_3 \neq 3)$$

Then we repeat the same process for $G(e_5 | E_2)$:

$$G(e_5 | E_2) = ((F_1 \neq 2) \text{ OR } (F_2 \neq 2) \text{ OR } (F_3 \neq 3)) \text{ AND}((F_1 \neq 2) \text{ OR } (F_3 \neq 3))$$

which is next converted into the disjunctive form:

$$\begin{aligned} G(e_5 | E_2) = {} & ((F_1 \neq 2) \text{ AND } (F_1 \neq 2) \text{ OR } ((F_1 \neq 2) \text{ AND } (F_3 \neq 3)) \text{ OR} \\ & ((F_2 \neq 2) \text{ AND } (F_1 \neq 2)) \text{ OR } ((F_2 \neq 2) \text{ AND } (F_3 \neq 3)) \text{ OR} \\ & ((F_3 \neq 3) \text{ AND } (F_1 \neq 2)) \text{ OR } ((F_3 \neq 3) \text{ AND } (F_3 \neq 3)) \end{aligned}$$

and simplified to:

$$G(e_5 | E_2) = (F_1 \neq 2) \text{ AND } (F_3 \neq 3)$$

Finally in step 3 we need to combine the two stars (rules) into a cover:

$$G(E_1 | E_2) = (F_1 \neq 2) \text{ AND } (F_3 \neq 3)$$

From the knowledge of the feature domains we can write the final cover, or rule, covering all positive examples as:

$$G(E_1 | E_2) = (F_1 = 1 \text{ OR } 3) \text{ AND } (F_3 = 1 \text{ OR } 2)$$

The cover is actually written as:

$$<F_1 \neq 2 > <F_3 \neq 3>$$

which reads

$$\text{IF } (F_1 = 1 \text{ OR } 3) \text{ AND } (F_3 = 1 \text{ OR } 2) \text{ THEN class positive}$$

As one can see the generation of a cover is computationally very expensive. In terms of a general set covering problem creating a cover $G(E_1 | E_2)$, while using the $\geq$ operators in the description of selectors, means that we are dividing the entire space, into subspaces in such a way that in one subspace we will have all the positive examples while all the negative examples will be included in another, nonintersecting subspace.

A substantial disadvantage of AQ algorithms is that they handle noise outside of the algorithm itself, by rule truncation.

4 Hybrid Algorithms

After reviewing Michalski's rule algorithms we concentrate on the description of our hybrid algorithm, the CLIP4 (Cover Learning (using) Integer Programming). The CLIP4 algorithm is a hybrid that combines ideas (like its predecessors the CLILP3 and CLIP2 algorithms) of Michalski's rule algorithms and decision trees. More precisely, CLIP4 uses a rule-generation schema similar to Michalski's AQ algorithms, as well as the tree-growing technique to divide training data into subsets at each level of a (virtual) decision tree similar to decision tree algorithms (Quinlan, 1993). The main difference between CLIP4 and the two families of algorithms is CLIP4's extensive use of our own algorithm for set covering (SC), which constitutes its core operation. SC is performed several times to generate the rules. Specifically, the SC algorithm is used to select the most discriminating features, to grow new branches of the tree, to select data subsets from which CLIP4 generates the least overlapping

rules, and to generate final rules from the (virtual) tree leaves, which store subsets of the data. An important characteristic that distinguishes CLIP4 from the vast majority of ML algorithms is that it generates production rules that involve inequalities. This results in generating a small number of compact rules, especially in domains where attributes have large number of values and where majority of them are associated with the target class. In contrast, other inductive ML algorithms that use equalities would generate a large number of complex rules for these domains.

CLIP4 starts by splitting the training data in a decision-tree-like manner. However, it does so not by calculating any index of "good" splitting, like entropy, but it selects features and generates rules by solving an Integer Programming (IP) model. CLIP4 uses the training data to construct an IP model and then uses a standard IP program to solve it. CLIP4 differs from the decision tree algorithms is that it splits the data into subsets in several ways, not just in one "best" way. In addition, there is no need to store the entire decision tree in CLIP4. It keeps only the leaf nodes of the "tree" (the tree, in fact, does not exist). This results in the generation of simpler rules, a smaller number of rules, and a huge memory saving. Another advantage is that the solution of the IP model for splitting the data is relatively quick, as compared to the calculation of entropies. The solution returned from the IP model indicates the most important features to be used in the generation of rules. IP solutions may include preferences used in other machine learning algorithms (Michalski and Larson, 1978), like the *largest complex first* where IP solution can generate features that cover the largest number of positive examples. Or, the *background knowledge first*, where any background knowledge can be incorporated into the rules by including user-specified features, if it is known that they are crucial in describing the concept.

4.1 Our Set Covering Algorithm

As we mentioned above, several key operations performed by CLIP4 are modeled and solved by the set covering algorithm, which is a simplified version of integer programming (IP). IP is used for function optimization that is subject to a large number of constraints. Several simplifications are made to the IP model to transform it into the SC problem: the function that is the subject of optimization has all its coefficients set to one; their variables are binary, x_i={0,1}; the constraint function coefficients are also binary; and all constraint functions are greater than or equal to one. The SC problem is NP-hard, and thus only an approximate solution can be found. First, we transform the IP problem into the binary matrix (BIN) representation that is obtained by using the variables and constraint coefficients. BIN's columns correspond to variables (features/attributes) of the optimized function; its rows correspond to function constraints (examples), as illustrated in Figure 1. CLIP4 finds the solution of the SC problem in terms of selecting a minimal number of columns that have the smallest total number of 1's. This outcome is obtained by minimizing the number of 1's that overlap among the columns and within the same row. The solution consists of a binary vector composed of the selected columns. All rows for which there is a value of 1 in the matrix, in a particular column, are assumed to be "covered" by this column.

$$\begin{gathered} Minimize: \\ x_1 + x_2 + x_3 + x_4 + x_5 = Z \\ Subject\ \ to: \\ x_1 + x_3 + x_4 \geq 1 \\ x_2 + x_3 + x_5 \geq 1 \\ x_3 + x_4 + x_5 \geq 1 \\ x_1 + x_4 \geq 1 \\ Solution: \\ Z = 2, \quad when\ x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0 \end{gathered}$$

$$\begin{gathered} Minimize: \\ x_1 + x_2 + x_3 + x_4 + x_5 = Z \\ Subject\ \ to: \\ \begin{bmatrix} 1,0,1,1,0 \\ 0,1,1,0,1 \\ 0,0,1,1,1 \\ 1,0,0,1,0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \geq 1 \end{gathered}$$

Fig. 1. A simplified set-covering problem and its solution (on the left); in the BIN matrix form (on the right)

To obtain a solution we use our SC algorithm, which is summarized as follows.

Given: BINary matrix. *Initialize*: Remove all empty (inactive) rows from the BINary matrix; if the matrix has no 1's, then return error.

1. Select active rows that have the minimum number of 1's in rows – *min-rows*
2. Select columns that have the maximum number of 1's within the *min-rows* – *max-columns*
3. Within *max-columns* find columns that have the maximum number of 1's in all active rows – *max-max-columns*. If there is more than one *max-max-column*, go to Step 4., otherwise go to Step 5.
4. Within *max-max-columns* find the first column that has the lowest number of 1's in the inactive rows
5. Add the selected column to the solution
6. Mark the inactive rows. If all the rows are inactive then terminate; otherwise go to Step 1.

Result: Solution to the SC problem.

In the above pseudocode, an active row is a row not covered by a partial solution, and an inactive row is a row already covered by a partial solution. We illustrate how the SC algorithm works in Figure 2 using a slightly more complex BIN matrix that the one shown in Figure 1. The solution consists of the second and fourth columns, which have no overlapping 1's in the same rows.

Before we describe the CLIP4 algorithm in detail, let us first introduce a necessary notation. The set of all training examples is denoted by S. A subset of positive examples is denoted by S_P and the subset of negative examples by S_N. S_P and S_N are represented by matrices whose rows represent examples and whose columns correspond to attributes. The matrix of positive examples is denoted as POS and their number by N_{POS}. Similarly for the negative examples, we have matrix NEG and number N_{NEG}. The following properties are satisfied for the subsets:

$$S_P \cup S_N = S, \quad S_P \cap S_N = \varnothing, \quad S_N \neq \varnothing, \text{ and } \quad S_P \neq \varnothing$$

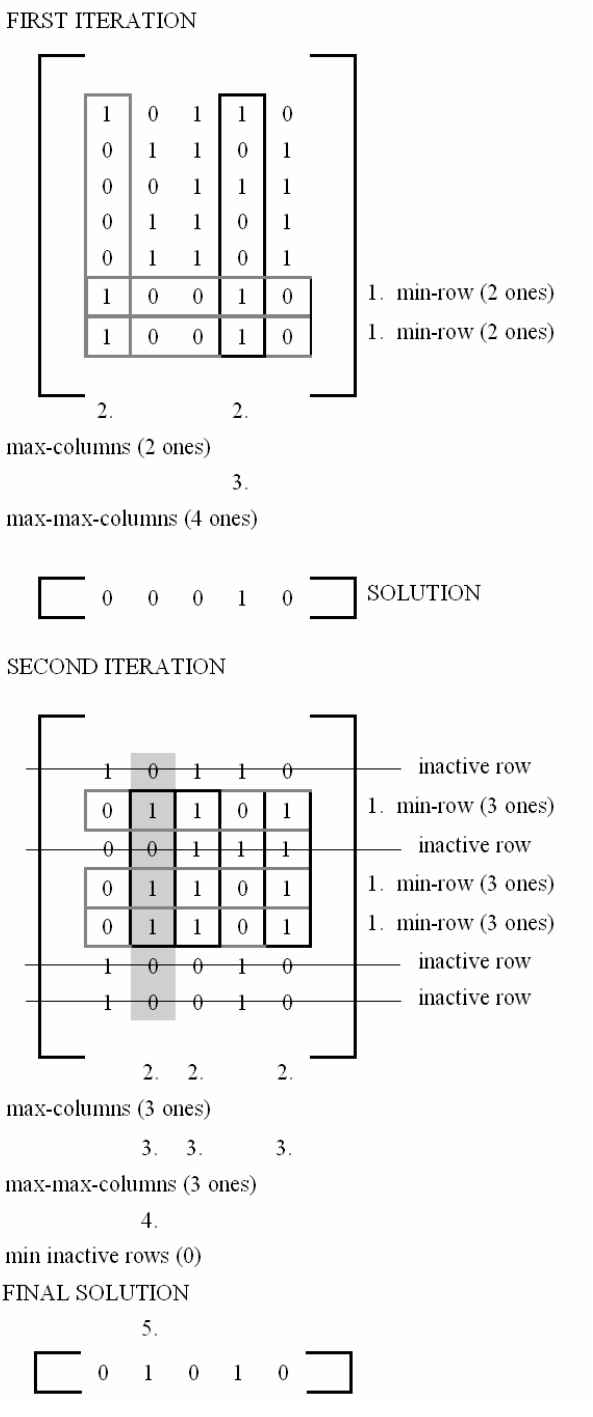

Fig. 2. Solution of the SC problem using the SC algorithm

The examples are described by a set of K attribute-value pairs:

$$e = \wedge_{j=1}^{K}[a_j \# v_j]$$

where a_j denotes the j^{th} attribute with value $v_j \in d_j$, and # is a relation ($\neq$, $=$, $<$, $\approx$, $\leq$, etc.), where K is the number of attributes. An example e consists of a set of selectors $s_j = [a_j \neq v_j]$.

The CLIP4 algorithm generates rules in the form:

$$\text{IF } (s_1 \wedge \ldots \wedge s_m) \qquad \text{THEN class} = \text{class}_i$$

where all selectors are only in the form $s_i = [a_j \neq v_j]$, that is they use only inequalities.

The positive examples from the matrix POS are described by a set of values $\text{pos}_i[j]$, where $j=1,\ldots,K$ is the column number and i is the example number (the row number in the POS matrix). The negative examples are described similarly by a set of $\text{neg}_i[j]$ values. CLIP4 uses binary matrices (BIN) that are composed of K columns, filled with either 1 or 0 as values. Each element of the BIN matrix is denoted by $bin_i[j]$, where i is a row number and j is a column number. These matrices are the result of operations performed by CLIP4 which are modeled and solved using the SC algorithm described above. The pseudocode of the CLIP4 algorithm is provided in Figure 3.

The algorithm consists of three key phases that are explained below:

Phase I: The POS data is partitioned into subsets of similar data in a decision-tree-like manner. Each node represents a data subset. Each level of the tree is built using one negative example to find selectors that distinguish between all positive examples and the negative example. The selectors are used to create new branches of the tree. During the tree growth, we use pruning to eliminate noise from the data (described later) and to avoid excessive growth, which reduces execution time.

The tree is grown in a top-down manner. At the i^{th} tree level, N_i subsets, represented by matrices $\text{POS}_{i,j}$ ($j=1,\ldots,N_i$), are generated using N_{i-1} subsets from the previous tree level and the single negative example neg_i. Each subset, represented by the matrix $\text{POS}_{i,j}$ (examples that constitute this matrix's elements are denoted as $\text{pos}^{i,j}$), is transformed into a BIN matrix of the same size as the $\text{POS}_{i,j}$ matrix, using the neg_i example, and then modeled and solved using the SC method. The solution is used to generate subsets for the next tree level, represented by the $\text{POS}_{i+1,j}$ matrices.

CLIP4 grows a virtual "tree", since any iteration consists of only one level: the most recent bottom tree level. Moreover, this tree is pruned so that even the one level that is kept has at most a few nodes. This results in a high memory efficiency. The data splits are made by generating not just one "best" division, based on some "best" feature (say in terms of the highest information gain, as in decision trees), but a set of divisions based on any feature that distinguishes between the positive and the one negative example. This mechanism of partitioning data assures generation of the (possibly) most general rules.

```
N0=1; Create POS0,1 consisting of entire SP; Create NEG consisting of entire SN                    // Initialize
1  for negi, i=1 to NNEG do {                                                                      // PHASE I STARTS
2     for j=1 to Ni-1 do {                                                                         //for each POSi-1,j matrix
3        for k=1 to K do {                                                                         //create new BINj matrix
4           for l=1 to number of POSi-1,j rows do {
5              if posl^i-1,j[k] = negi[k] then binj[k] = 0;
6              if posl^i-1,j[k] ≠ negi[k] then binj[k] = 1;
7              if posl^i-1,j[k] = '*' then binj[k] = 0;                                           // missing value encountered
8              if negi[k] = '*' then binl[k] = 0; }}                                              // missing value encountered
9        SOLj = SolveSCProblem(BINj);}
10    PruneMatrices(POSi-1,j , SOLj , j=1,…,Ni-1);
11    ApplyGeneticOperators(POSi-1,j , SOLj , j=1,…,Ni-1);
12    Ni=1;                                                                                        //counter for POSi+1 matrices
13    for j=1 to Ni-1 do {                                                                         //for each POSi,j matrix
14       if POSi-1,j was not pruned or redundant then {
15          for k=1 to K do {                                                                     //through entire solution vector
16             if SOLj[k]=1 then {                                                                //then create new POSi,Ni matrix
17                for l=1 to number of POSi-1,j rows do {
18                   if posl^i-1,j[k]≠negi[k] then add posl^i-1,j to POSi,Ni matrix; }}
19       Ni=Ni+1; }}}
20    for j=1 to Ni do {                          //for each POSi matrix check if it large enough to be not considered as a noise
21       if number of rows of POSi,j < NoiseThreshold then { remove POSi,j from the tree; Ni=Ni-1;}}
22    EliminateRedundantMatrices(POSi,j j=1,…,Ni); }
23 Create BIN matrix that consist of N_NNEG columns and NPOS rows, and fill with zeros // PHASE II STARTS
24 for i=1 to N_NNEG do {                                                                          // for all tree leaves
25    for j=1 to NPOS do {
26       for k=1 to number of rows of POS_NNEG,i do { if posj^0,0 = pos_k^NNEG,i then binj[i]=1;}}}
27 SOL = SolveSCProblem (BIN);                                                                     // select best leaf node subsets
28 for i=1 to N_NNEG do {                                                                         // through entire solution vector
29    create BINi=NEG;
30    if SOL[i]=1 then {                                                                          // back-project POSNneg,i matrix
31       for j=1 to K do {
32          for k=1 to NNEG do {
33             if negk[j]='*' then bink^i[j]=0; else {
34                for l=1 to number of rows of POS_NNEG,i do { if pos_l^NNEG,i[j]=negk[j] then bink^i[j]=0; }}}}}
35    for j=1 to K do {                                                                           // convert BINi values to binary
36       for k=1 to NNEG do { if bink^i[j]≠0 then bink^i[j]=0;}}
37    SOLi = SolveSCProblem (BINi);
38    for j=1 to K do {                                                                             // start generation of i-th rule
39       if SOLi[j]=1 then {                                                                       // add selectors to the rule
40          for k=1 to NNEG do { if bin^i_k[j]=1 then add "aj≠neg k[j]" selector to the Rulei; }}}}
                                                                                                    // PHASE III STARTS
41 best#covered=0; previous_best#covered=0;                                                       // holds # ex. covered by the rule
42 while best#covered≥0 do {                                                                      //until best rules are accepted
43    for i=1 to N_NNEG do {                                                                      // through all generated rules
44       coversi=NPOS;                                                                  // # examples covered by Rulei
45       for j=1 to NPOS do {                                                                     // for all examples in POS
46          for k=1 to K do {
47             for l=1 to number of selectors in Rulei do {
48                if al=k and vk=posj^0,1[k] then { coversi=coversi –1; j=j+1; }}}}}              // example not covered by Rulei
49       if best#covered<coversi then best#covered=coversi; best_rule=Rulei; }
50    NPOS = NPOS – best#covered;
51    if NPOS<StopThreshold or NPOS=0 then STOP;
52    POS0,1 = POS0,1 – examples covered by best_rule;
53    if (best#covered<previous_ best#covered/2 and best#covered<NPOS/2) then best#covered=-1;      // multiple rules
54    previous_best#covered_examples=best#covered_examples; }
```

Fig. 3. Pseudocode of the CLIP4 algorithm

Phase II: A set of terminal subsets (tree leaves) is selected using two criteria. First, large subsets are preferred over small ones (according to Occam's razor which hopefully will result in the rules that are "strong" and more general), while the

accepted subsets (between them) must cover the entire POS data. Second, we want to use the completeness criterion. To that end, we first perform a back-projection of one of the selected positive data subsets using the entire NEG data, and then convert the resulting matrix into a BIN matrix and solve it using the SC method. The solution is used to generate a rule, and the process is repeated for every selected positive data subset. Explanation of lines 24-41: $N_{N_{NEG}}$ is the number of tree leaves. The binary BIN matrix is used to select the minimal set of leaves that covers the entire POS data. Back-projection results in a matrix BIN_i that has 1's for the selectors that can distinguish between positive and negative examples. The back-projection generates one matrix for every terminal subset ($POS_{N_{NEG},i}$). It is computed from the NEG matrix, by setting a value from NEG to zero if the same value appears in the corresponding column in the $POS_{N_{NEG},i}$ matrix; otherwise it is left unchanged. The i^{th} rule is generated by solving the SC problem for BIN_i and adding a selector for every 1 that is in any column indicated by the solution.

The rule induction generates a rule directly from two sets of data (NEG data and the selected subset of positive data). It does not require the use of logic or storing and traversing the entire tree, as in decision trees, but only a comparison of values between the two matrices. Thus, the only two data structures required to generate the rules are lists (vectors) and matrices.

Phase III: A set of best rules is selected from the generated rules. Rules that cover the most POS examples are chosen as being possibly the most general. If there is a tie between the "best rules", the rule that uses the minimal number of selectors is chosen.

To find the number of examples covered by a rule, we first find examples not covered by the rule, and subtract these from the total number of examples. We do this because the rules consist of selectors involving inequalities. The variable called $covers_i$ records the number of positive examples covered by the i^{th} rule. After a rule is accepted, the positive examples covered by it are removed from the POS matrix (line 53). In this phase, more than one rule can be generated. The rules are accepted in the order from the strongest to the weakest. We use a heuristic for accepting multiple rules, which states that the next rule is accepted when it covers at least half the number of examples covered by the previously accepted rule and at least half the number of positive examples not covered by any rule so far. CLIP4 generates multiple rules in a single sweep through the data. The thresholds used in the pseudocode in Figure 3 are described later. We illustrate the high-level view of the process used by CLIP4 in Figure 4.

The basic characteristics of the CLIP4 algorithm are its completeness (the generated set of rules describes all positive examples), consistency (rules do not describe any of the negative examples) and use of a minimal number of selectors in the generated rules. CLIP4 is memory-efficient (because it stores only the bottom level of the tree) and is robust to noisy and missing-value data, as shown later in the chapter.

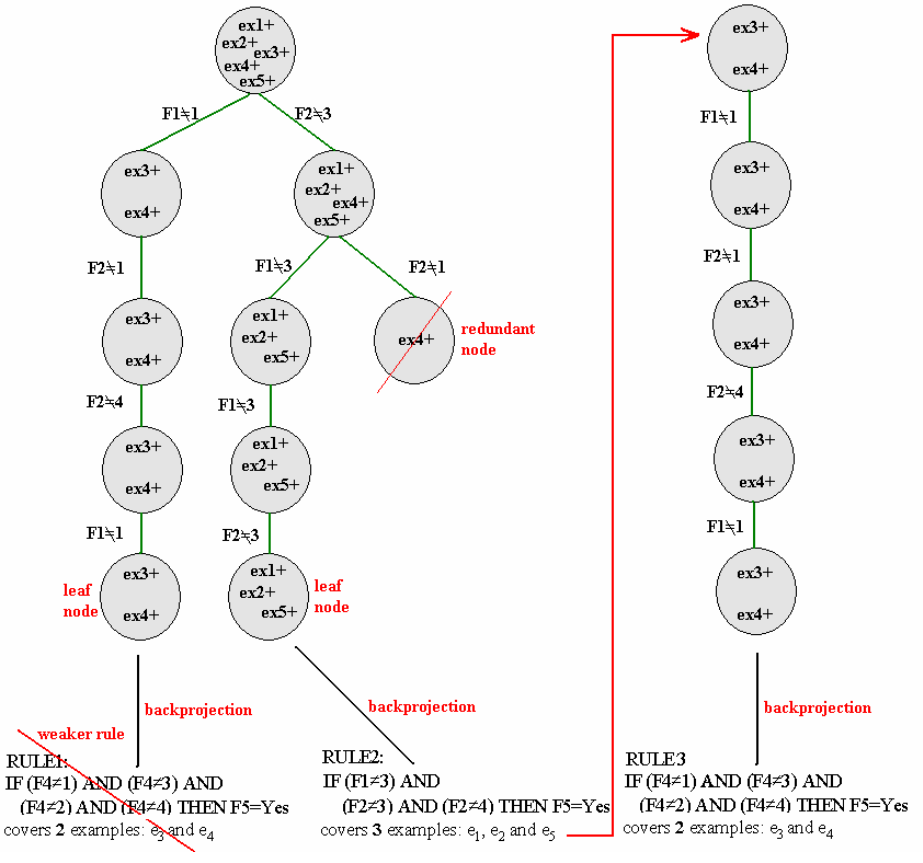

Fig. 4. A different view of the process used by the CLIP4 algorithm

4.2 Handling of Missing Values

Data with the missing values of the attributes are often encountered in real-world situations. If the number of examples with missing values is small, i.e., only a small fraction of the data is affected, we can simply discard them in a preprocessing step. However, in many situations – for example, in medical data - the number of examples (records) with missing values can exceed 50% of the data (Kurgan et al., 2005). In such situations, we need to use all examples and all attributes with values and omit from our computations only these attributes with missing values. The CLIP4 algorithm does not discard any examples with missing values nor does it fill them with any computed values. It simply takes advantage of the available values in the examples and ignores the missing values.

Attributes with missing values influence the first two phases of the algorithm. They are removed from processing by filling in the corresponding cell in the BIN matrix cell with a 0 when the new branch of the tree in Phase I is generated, and when the back-projection in Phase II is performed. The mechanism used in CLIP4 to deal with missing attribute values assures that they are not used in generated rules and that all existing

attribute values, even in examples with missing attributes, are used during the rule generation process. The advantage of this mechanism is that the user simply supplies data to the algorithm and obtains his or her without needing to use some statistical methods to calculate missing values. This approach works well under the assumption that the remaining (complete) portion of the data is sufficient to infer the correct model.

Data shown in Table 2 is used to illustrate how CLIP4 deals with missing values. The missing values are denoted by * and class1 constitutes the positive class.

Table 2. Data with missing values

S	F_1	F_2	F_3	F_4	Class
e_1	1	2	3	*	1
e_2	1	3	1	2	1
e_3	*	3	2	5	1
e_4	3	3	2	2	1
e_5	1	1	1	3	1
e_6	3	1	2	5	2
e_7	1	2	2	4	2
e_8	2	1	*	3	2

CLIP4 generates two rules:

IF $F_1 \neq 3$ AND $F_1 \neq 2$ AND $F_3 \neq 2$ THEN class1
IF $F_2 \neq 2$ AND $F_2 \neq 1$ THEN class1

The first rule covers examples 1, 2, and 5, while the second covers examples 2, 3, and 4. Between them they cover all positive examples, including those with missing values, and none of the negative examples. The rules overlap since both cover the second example.

4.3 Classification

CLIP4 classifies examples by using sets of rules generated for all classes. Two classification outcomes are possible: an example is assigned to a particular class, or it is left unclassified. To classify an example, two principles are used:

- All rules that cover the example are found. If no rules cover the example then it remains unclassified. Such a situation may occur if the example has missing values for all attributes used by the rules
- For every class, goodness of the rules describing a particular class and covering the example is summed up. The example is assigned to a class that has the highest summed value. If there is a tie, then the example is left unclassified. For each generated rule, a goodness value, equal to the percentage of the positive training examples that it covers, is calculated. For instance, if an example is covered by two rules from class 1 with corresponding goodness values of 15% and 20%, but the example is also covered by two rules from class 2 with goodness values 50% and 10%, then it would be classified to Class 2 (the sum of goodness values for Class 1 is 35% vs. 60% for Class 2).

4.4 Thresholds

CLIP4 uses three thresholds to perform tree pruning and to remove "noisy" examples:

Noise Threshold determines which nodes (possibly containing noisy positive examples) are pruned from the tree grown in Phase 1. The threshold prunes every node that contains fewer examples than its value.

Pruning Threshold is used to prune nodes from the generated tree. It uses a goodness value, identical to the fitness function described later, to perform selection of the nodes. The threshold selects the first few nodes with the highest fitness value and removes the remaining nodes from the tree.

Stop Threshold stops the algorithm when fewer than the threshold number of positive examples remains uncovered. CLIP4 generates rules by partitioning the data into subsets containing similar examples, and removes examples covered by already generated rules. This approach has the advantage of eliminating small subsets of positive examples (which contain examples different than the majority already covered) from the subsequent rule generation process. If the user were to know the amount of noise in the data then the threshold could be set to this value (e.g., 5%). The noise and stop thresholds are specified as a percentage of the size of positive data and thus are easily scalable.

4.5 Use of Genetic Operators to Improve Accuracy on Small Training Data

CLIP4 uses genetic algorithms to improve the accuracy of generated rules. Its genetic module works by exploiting a single loop through a number of evolving populations. The loop consists of establishing the initial population of individuals and then selecting the new population from the old population, altering and evaluating the new population, and replacing the old one with the new. These operations are performed until a termination criterion is satisfied. CLIP4 uses the GA in Phase I to enhance the partitioning of the data and to obtain more "general" leaf node subsets. The components of the genetic module are as follows:

- *Population* and *individual*
  An individual/chromosome is defined as a node in the tree and consists of the $POS_{i,j}$ matrix (the j^{th} matrix at the i^{th} tree level) and $SOL_{i,j}$ (the solution to the SC problem obtained from the $POS_{i,j}$ matrix). A population is defined as a set of nodes at the same level of the tree.
- *Encoding* and *decoding scheme*
  There is no need for encoding using the individuals defined above since GA operators are used on the binary $SOL_{i,j}$ vector.
- *Selection* of the new population
  The initial population is the first tree level that consists of at least two nodes. CLIP4 uses the following fitness function to select the most suitable individuals for the next generation:

$$fitness_{i,j} = \frac{number\ of\ examples\ that\ constitute\ POS_{i,j}}{number\ of\ subsets\ that\ will\ be\ generated\ from\ POS_{i,j}\ at\ (i+1)^{th}\ tree\ level}$$

The fitness value is calculated as the number of rows of the $POS_{i,j}$ matrix divided by the number of 1's from the $SOL_{i,j}$ vector. The fitness function has high values for the tree nodes that consist of a large number of examples with a low branching factor. These two properties influence the generalization ability of the rules and the speed of the algorithm.

The mechanism for selecting individuals for the next population is as follows:

- All individuals are ranked using the fitness function
- Half of the individuals with the highest fitness are automatically selected for the next population (they will branch to create nodes for the next tree level)
- The second half of the next population is generated by matching the best with the worst individuals (the best with the worst, the second best with the second worst, etc.) and applying GA operators to obtain new individuals (new nodes in the tree). This mechanism promotes the generation of new tree branches that contain large number of examples.

An illustration of the selection mechanism used in the CLIP4 algorithm is shown in Figure 5.

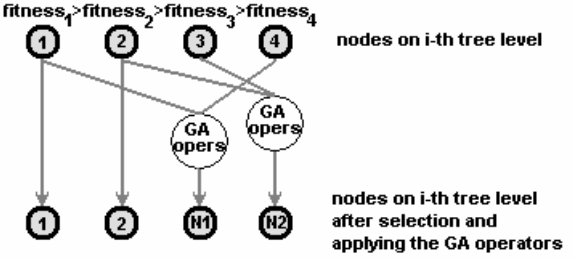

Fig. 5. Selection mechanism performed by the GA module

The GA module uses the crossover and mutation operators. Both are applied only to the $SOL_{i,j}$ vectors. The resulting $ChildSOL_{i,j}$ vector, together with the $POS_{i,j}$ matrix of the parent with the higher fitness value, constitutes the new individual. The selection of the $SOL_{i,j}$ matrix assures that the resulting individual is consistent with CLIP4's way of partitioning data. The crossover operator is defined as

$$ChildSOL_i = \max (Parent1SOL_i, Parent2SOL_i)$$

where $Parent1SOL_i$ and $Parent2SOL_i$ are the i^{th} values of $SOL_{i,j}$ vectors of the two parent nodes.

CLIP4 uses the mutation with 10% probability to flip a value in the ChildSOL vector to 1, regardless of the existing value. For particular data, the probability of mutation can be established experimentally. Each 1 in the ChildSOL generates a new branch, except for 1's taken from the $SOL_{i,j}$ of the parent with higher fitness value, which are discarded because they would generate branches redundant with the branches generated by the parent.

The termination criterion checks whether the bottom level of the tree has been reached. The entire evolution process of the GA module is shown in Figure 6.

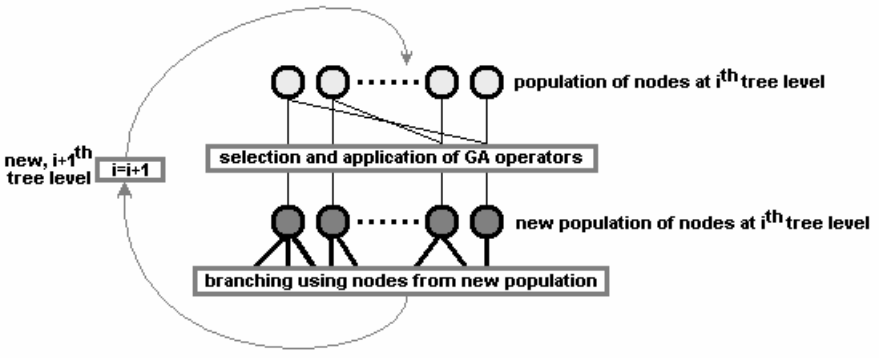

Fig. 6. The evolution process performed by the GA module

The GA module of the CLIP4 algorithm is used when a small data set covering a small portion of the search space is available; the reason is that then CLIP4 will generate rules that potentially cover a not yet described portion of the state space.

4.6 Pruning

CLIP4 uses prepruning to prune the tree during its process of tree generation. The prepruning stops the learning process even if some positive examples have still not been covered and while some negative examples are still covered. More precisely, CLIP4 prunes the tree grown in Phase 1 as follows:

- First, it selects a number (defined by the pruning threshold) of best nodes on the i^{th} tree level. The selection is performed based on the goodness criterion identical to the fitness function described later. Only the selected nodes are used to branch into new nodes, and are passed to the $(i+1)^{th}$ tree level.
- Second, all redundant nodes that resulted from the branching process are removed. Two nodes are redundant if one mode contains positive examples that are identical to each other , or forms a subset of positive examples of the other node. The node with the smaller number of examples is pruned first.
- Third, after the redundant nodes are removed, each new node is evaluated using the noise threshold. If the node contains fewer examples than the number specified by the noise threshold, then the node is pruned.

The prepruning method used in the CLIP4 algorithm avoids some disadvantages of classical prepruning. Namely, it never allows negative examples to be covered by the rules, and it ensures the completeness condition of the rule generation process. This type of prepruning increases the accuracy of the generated rules and lowers the complexity of the algorithm.

4.7 Feature and Selector Ranking

The CLIP4 algorithm not only generates a data model in terms of rules but also ranks attributes and selectors. This aspect of the algorithm can be used for feature selection, which is defined as a process of finding a subset of original features that is optimal

according to some defined criterion; in the case of classification, the criterion is to obtain the highest predictive accuracy. CLIP4 ranks attributes and selectors by assigning to them a goodness value that quantifies their relevance to a particular learning task. These rankings provide additional insight into the information hidden in the data.

The goodness of each attribute and selector is computed by means of a set of generated rules for a given class. All attributes with goodness value greater than zero are strongly relevant to the classification task. Strong relevancy means that the attribute cannot be removed from the attribute set without decreasing the accuracy of classification. The other attributes can be removed from the data. The attribute and selector goodness values are computed in the following manner:

- Each generated rule has a goodness value equal to the percentage value of the positive training examples it covers. Recall that each rule consists of one or more (attribute, value of the selector) pairs.
- Each selector has a goodness value equal to the goodness of the rule from which it comes. The goodness of the same selectors from different rules is summed, and then scaled to the (0,100) range; with 100 being the highest. Scaling of the goodness values is necessary because otherwise the summed goodness for a particular selector can grow to over 100, while only the ratio to the goodness's of other selectors is important.
- For each attribute, the sum of the scaled goodness of all attribute selectors is computed and then divided by the number of attribute values to obtain the goodness of the attribute.

The feature and selector ranking performed by CLIP4 algorithm can be used to:

- select only relevant attributes (features) and discard irrelevant ones; the user can discard all attributes that have a goodness of 0 and still have an equally accurate model of the data, and
- provide additional insight into data properties. The selector ranking can help in analyzing the data in terms of the relevance of the selectors to the classification task.

5 DataSqueezer Algorithm

The DataSqueezer algorithm was first published in (Kurgan 2003; Kurgan et al., 2006) and later it was incorporated into a larger data mining system called MetaSqueezer (Kurgan and Cios, 2004). The latter system was recently successfully used in analysis of clinical data concerning patients with cystic fibrosis (Kurgan et al., 2005); it was able to discover clinical information that is implicated in stabilization or improvement of patient's health. The DataSqueezer induces a set of production rules from a supervised training data by utilizing a very simple data reduction procedure via prototypical concept learning, inspired by the Find S algorithm (Mitchell, 1997). The main advantages of this method are its log-linear complexity resulting in very fast generation of the rule-based classification models and robustness to missing values. DataSqueezer is also relatively simple to implement.

Let us denote a training dataset by D, consisting of s examples and k attributes. The training dataset is divided into two disjoint subsets, D_P that consists of positive examples and D_N that includes the negative examples. Figure 7 shows the pseudocode of the algorithm. Vectors and matrices are denoted by capital letters, while elements of vectors/matrices are denoted with the same name but use small letters. The matrix of positive/negative examples is denoted as POS/NEG and their number as N_{POS}/N_{NEG}. The examples are represented by rows, and features/attributes by columns. Positive examples, POS, and negative examples, NEG, are denoted in the DataReduction procedure by $d_i[j]$ values, where j=1,…,k is the column number, and i is the example number (row number in matrix D, which denotes either POS or NEG). The algorithm also uses matrices that store intermediate results (G_{POS} for the POS, and G_{NEG} for the NEG), which have k columns. Each cell of the G_{POS} matrix is denoted by $gpos_i[j]$, where i is a row number and j is a column number; and similarly for the G_{NEG} matrix each cell is denoted by $gneg_i[j]$. The G_{POS} stores a reduced subset of the data from the POS matrix, and the G_{NEG} stores a reduced subset of the data from the NEG matrix. The G_{NEG} and G_{POS} matrices have an additional $(k+1)^{th}$ column that stores the number of examples from the NEG and POS matrices, described by a particular row in the G_{NEG} and G_{POS}, respectively. For example, $gpos_2[k+1]$ stores the number of examples from the POS that are described by the second row in the G_{POS} matrix.

DataSqueezer works in two steps. In step 1 it performs data reduction to generalize information stored in the original data. Data reduction is performed via the use of a prototypical concept learning procedure. This is performed for both positive and negative data and it results in generation of the G_{POS} and G_{NEG} matrices. This reduction

```
Given:   POS, NEG, k (number of attributes), s (number of examples)
Step1.
1.1      G_POS = DataReduction(POS, k);
1.2      G_NEG = DataReduction(NEG k);

Step2.
2.1      Initialize RULES = []; i=1;                    // where rules_i denotes i^th rule stored in RULES
2.2      create LIST = list of all columns in G_POS
2.3      within every G_POS column that is on LIST, for every non missing value a from selected column j compute sum,
         s_aj, of values of gpos_i[k+1] for every row i, in which a appears and multiply s_aj, by the number of values the
         attribute j has
2.4      select maximal s_aj, remove j from LIST, add "j = a" selector to rules_i
2.5.1    if rules_i does not describe any rows in G_NEG
2.5.2        then remove all rows described by rules_i from G_POS, i=i+1;
2.5.3          if G_POS is not empty go to 2.2, else terminate
2.5.4      else go to 2.3
Output:  RULES describing POS

         DataReduction (D, k)                           // data reduction procedure for D=POS or D=NEG
DR.1     Initialize G = []; i=1; tmp = d_1; g_1 = d_1; g_1[k+1]=1;
DR.2.1   for j=1 to N_D                                 // for positive/negative data; N_D is N_POS or N_NEG
DR.2.2     for kk = 1 to k                              // for all attributes
DR.2.3       if (d_j[kk] ≠ tmp[kk] or d_j[kk] = '*')
DR.2.4         then tmp[kk] = '*';                      // '*' denotes missing" do not care" value
DR.2.5     if (number of non missing values in tmp ≥ 2)
DR.2.6         then g_i = tmp; g_i[k+1]++;
DR.2.7       else i++; g_i = d_j; g_i[k+1]=1; tmp = d_j;
DR.2.8   return G;
```

Fig. 7. Pseudocode of the DataSqueezer algorithm

is related to the least generalization in the FindS method. The main difference is that FindS performs the least generalization multiple times for the entire positive set through a beam-search strategy, while the DataSqueezer performs it only once in a linear fashion by generalizing consecutive examples. It also generalizes the negative data. In step 2 the DataSqueezer generates rules by a greedy hill-climbing search on the reduced data. A rule is generated by using the search procedure starting with an empty rule, and adding selectors until the termination criterion is satisfied. The rule that is being generated consists of selectors generated using the G_{POS}, and is checked against the G_{NEG}. If the rule covers any data in the G_{NEG}, a new selector is added to make it more specific, and thus better able to distinguish between positive and negative data. The maximum depth of the search is equal to the number of features. Next, the examples covered by the generated rule are removed, and the process is repeated.

When applied to dataset given in Table 1 the DataSqueezer algorithm generates two rules:

$$\text{If } F_1 = 1 \text{ THEN Class} = 1$$
$$\text{If } F_3 = 3 \text{ THEN Class} = 2$$

Note that each rule was generated separately, i.e., the first rule was generated by using class 1 as positive and class 2 as negative, while the second rule was generated by using class 2 as positive and class 1 as negative. This means that each of these rules can be used independently to classify the data shown in Table 1. When applied to the Table 1 data, each of these rules perfectly separates the two classes.

The DataSqueezer algorithm handles data with missing values very well. Similarly to CLIP4, it uses all available information while ignoring missing values during the induction process, i.e., the missing values are handled as "do not care" values. The algorithm is robust to a very large number of missing values and was shown to successfully generate rules even from data that have more than half of missing values (Kurgan et al., 2005). DataSqueezer, like the other algorithms covered in this chapter handles only discrete-valued numerical and nominal attributes (the latter are automatically encoded into numerical values). We note that continuous attributes can be converted into discrete-values attributes using a discretization algorithm (Kurgan and Cios, 2004). The generated rules are independent of the encoding scheme because during the rule induction process no distances are calculated between the feature values.

DataSqueezer uses two thresholds that can be set by the user. By default these thresholds are set to zero. The pruning threshold is used to prune very specific rules, i.e., rules that cover only a few examples. The rule generation process is terminated if the first selector added to $rule_i$ has s_{aj} value equal to or smaller than the threshold's value. The algorithm induces rules by selecting maximal s_{aj} values (selectors that cover most of the POS examples) and removes examples that have already been covered. This approach has the benefit of leaving small subsets of positive examples that store different examples from the majority already covered. These potential outliers can be filtered out using this threshold. The generalization threshold is used to allow for rules that cover a small amount of negative data. It relaxes the requirement from line 2.5.1 (Figure 7) and allows for acceptance of rules that cover negative examples; the number is equal to or smaller than this threshold. It is a useful mechanism in the case of data with overlapping classes, or in the case of inconsistent examples present in the training dataset (examples that should have been but were not eliminated during preprocessing). Both thresholds should be set to small values preferably expressed as percentages of the POS data size.

6 Conclusions

The works of Michalski undoubtedly provided solid foundations and a strong motivation for many of the subsequent works on rule-based inductive ML algorithms. Some of our own methods, including the CLIP family of algorithms and DataSqueezer, were inspired by his work. We believe that ours and future generations of computer scientists will continues this line of research, which is fueled by an increasing demand for scalable methodologies that can generate human-readable models for current and future datasets.

References

Cios, K.J., Liu, N.: An algorithm which learns multiple covers via integer linear programming. Part I - The CLILP2 Algorithm. Kybernetes 24, 29–50 (1995)

Cios, K.J., Liu, N.: An algorithm which learns multiple covers via integer linear programming. Part I - The CLILP2 Algorithm. Kybernetes 24, 29–50 (1995); (The Norbert Wiener 1997 Outstanding Paper Award, http://www.mcb.co.uk/literati/outst97.htm#k)

Cios, K.J., Wedding, D.K., Liu, N.: CLIP3: cover learning using integer programming. Kybernetes 26(4-5), 513–536 (1997)

Cios, K.J., Pedrycz, W., Swiniarski, R., Kurgan, L.: Data mining: a knowledge discovery approach. Springer, Heidelberg (2007)

Cios, K.J., Pedrycz, W., Swiniarski, R.: Data mining methods for knowledge discovery. Kluwer, Dordrecht (1998)

Cios, K.J., Kurgan, L.: CLIP4: Hybrid inductive machine learning algorithm that generates inequality rules. Information Sciences 163(1-3), 37–83 (2004)

Farhangfar, A., Kurgan, L., Pedrycz, W.: A novel framework for imputation of missing values in databases. IEEE Transactions on Systems, Man, and Cybernetics, Part A 37(5), 692–709 (2007)

Farhangfar, A., Kurgan, L., Dy, J.: Impact of imputation of missing values on classification error for discrete data. Pattern Recognition 41(12), 3692–3705 (2008)

Kurgan, L.: Meta mining system for supervised learning, Ph.D dissertation, the University of Colorado at Boulder, Department of Computer Science (2003)

Kurgan, L., Cios, K.J.: Meta mining architecture for supervised learning. In: 7th International Workshop on High Performance and Distributed Mining, Proc. 4th International SIAM Conference on Data Mining, Lake Buena Vista, FL, pp. 18–26 (2004)

Kurgan, L., Cios, K.J.: CAIM discretization algorithm. IEEE Transactions on Data and Knowledge Engineering 16(2), 145–153 (2004)

Kurgan, L., Cios, K.J., Sontag, M., Accurso, F.: Mining the cystic fibrosis data. In: Zurada, J., Kantardzic, M. (eds.) Next generation of data-mining applications, pp. 415–444. IEEE Press - Wiley (2005)

Kurgan, L., Cios, K.J., Dick, S.: Highly scalable and robust rule learner: performance evaluation and comparison. IEEE Transactions on Systems, Man, and Cybernetics, Part B 36(1), 32–53 (2006)

Mitchell, T.M.: Machine learning. McGraw-Hill, New York (1997)

Kodratoff, Y.: Introduction to machine learning. Morgan Kaufmann, San Francisco (1988)

Langley, P.: Elements of machine learning. Morgan Kaufmann, San Francisco (1996)

Łukasiewicz, J.: O logice trójwartościowej (in Polish). Ruch Filozoficzny 5, 170–171 (1920); English translation: On three-valued logic. In: Borkowski, L. (ed.) Selected works by Jan Łukasiewicz, pp. 87–88. North–Holland, Amsterdam (1970)

Michalski, R.S.: On the quasi minimal solution of the general covering problem. In: Proc. 5th International Symposium on Information Processing (FCIP 1969), Bled, Yugoslavia, vol. A3, pp. 25–128 (1969)

Michalski, R.S.: Variable valued logic: system VLl. In: Proc. 1974 International Symposium on Multiple Valued Logic and Pattern Recognition, West Virginia University, Morgantown, pp. 323–346 (1974)

Michalski, R.S.: Knowledge acquisition through conceptual clustering: a theoretical framework and algorithm for partitioning data into conjunctive concepts. International Journal of Policy Analysis and Information Systems 4, 219–243 (1980)

Michalski, R.S., Mozetic, I., Hong, J., Lavrac, N.: The multipurpose incremental learning system AQ15 and its testing application to three medical domains. In: Proc. 5th National Conference on Artificial Intelligence, pp. 1041–1045. Morgan-Kaufmann, San Francisco (1986)

Mitchell, T.M.: Machine learning. McGraw-Hill, New York (1997)

Quinlan, J.R.: C4.5 programs for machine learning. Morgan-Kaufmann, San Francisco (1993)

Pawlak, Z.: Rough sets - theoretical aspects of reasoning about data. Kluwer, Dordrecht (1991)