Data Mining

The Search for Knowledge in Databases

Marcel Holsheimer, Arno Siebes
{marcel,arno}@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

Data mining is the search for relationships and global patterns that exist in large databases, but are 'hidden' among the vast amounts of data, such as a relationship between patient data and their medical diagnosis. These relationships represent valuable knowledge about the database and objects in the database and, if the database is a faithful mirror, of the real world registered by the database.

One of the main problems for data mining is that the number of possible relationships is very large, thus prohibiting the search for the correct ones by simple validating each of them. Hence, we need intelligent search strategies, as taken from the area of machine learning.

Another important problem is that information in data objects is often corrupted or missing. Hence, statistical techniques should be applied to estimate the reliability of the discovered relationships.

This report provides a survey of current data mining research, it presents the main underlying ideas, such as inductive learning, and search strategies and knowledge representations used in data mine systems. Furthermore, it describes the most important problems and their solutions, and provides an survey of research projects.

CR Subject Classification (1991): Database applications (H.2.8), Information search and retrieval (H.3.3), Learning (I.2.6) *concept learning, induction, knowledge acquisition*, Clustering (I.5.3)

Keywords & Phrases: database applications, machine learning, inductive learning, knowledge acquisition, data summarization

Report CS-R9406
ISSN 0169-118X
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Table of Contents

Chapter 1
Introduction

A database is a reliable store of information. One of the prime purposes of such a store is the efficient retrieval of information. This retrieved information is not necessarily a faithful copy of information stored in the database, rather, it is information that can be inferred from the database. From a logical perspective, two inference techniques can be distinguished:

Deduction is a technique to infer information that is a *logical consequence* of the information in the database. Most database management systems (DBMSs), such as relational DBMSs, offer simple operators for the deduction of information. For example, the join operator applied to two relational tables where the first administrates the relation between employees and departments and the second the relation between departments and managers, infers a relation between employees and managers.

Extending the deductive expressivity of query languages while remaining computationally tractable is pursued in the research area called deductive databases (see Ullman [60]).

Induction is a technique to infer information that is *generalized* from the information in the database. For example, from the employee-department and the department-manager tables from the example above, it might be inferred that each employee has a manager.

This is *higher-level* information, or *knowledge*: general statements about properties of objects. We search the database for *regularities*—combinations of values for certain attributes, shared by facts in the database. In a sense, this regularity is a high-level summary of information in the database. We can also formulate such a regularity as a *rule*, predicting the value of an attribute in terms of other attributes.

The most important difference between deduction and induction is that the former results in provably correct statements about the real world provided that the database is correct, while the latter only results in statements that are supported by the database, but not necessarily true in the real world. One of the most important aspects of the induction process is therefore the selection of the most plausible rules and regularities, supported by the database.

Inference of information from a database is beyond human capabilities, if only because of the ever growing size of databases. Hence, the inference-process should be supported by the DBMS. However, although all DBMSs support deduction of information, none supports induction. It is our goal to extend DBMS interfaces with inductive or *data mining* capabilities, thus revealing a source of valuable information.

There is a growing interest in data mining, both in research and application. Unfortunately, it is difficult to provide an overview of successes in the latter area since experimental results are often confidential because of their strategic importance. Only few examples can be found in literature. A credit card company has used a data mine tool to infer new, better, acceptance rules based on the credit-history of current clients [9]. IBM has used data mining techniques to predict defects during the assembly of disk drives [4]. Other applications can be found in [42].

1. Overview of the report

This report outlines data mining: it describes which forms of information can be derived, how regularities and rules can be discovered, and summarizes the major obstacles and techniques to overcome these. It also gives a survey of recent work in data mining. In the remainder of this chapter, we introduce the major topics.

1.1 Inductive learning

Humans and other intelligent creatures (which we from here on will refer to as *cognitive systems*) attempt to understand their environment by using a simplification of this environment—called a *model*. The creation of such a model is called *inductive learning*. During the learning phase, the cognitive system observes its environment and recognizes similarities among objects and events in this environment. It groups similar objects in classes and constructs rules that predict the behavior of the inhabitants of such a class.

Two learning techniques are of special interest. In *supervised learning*, an external teacher defines classes and provides the cognitive system with examples of each class. The system has to discover common properties in the examples for each class—the *class description*. This technique is also known as learning from examples. A class, together with its description forms a *classification rule* 'if ⟨*description*⟩ then ⟨*class*⟩' that can be used to predict the class of previously unseen objects.

In *unsupervised learning* the system has to discover the classes itself, based on common properties of objects. Hence, this technique is also known as learning from observation and discovery. Models, and the above forms of learning are discussed in Chapter 2.

1.2 Machine learning

The automation of inductive learning processes has been extensively researched in machine learning—an artificial intelligence research area. A machine learning system does not interact directly with its environment, but uses coded observations, often stored in a set—called the *training set*.

In supervised learning, the system searches for descriptions for the user defined classes, and in unsupervised learning, it constructs a summary of the training set as a set of newly discovered classes, together with their descriptions. In Chapter 3, we introduce training sets, classes and a representation for the descriptions.

As we will describe in Chapter 4, descriptions are constructed using an *iterative search strategy*, where the set of all constructible descriptions is searched for the best ones. First, an initial hypothesis (i.e. a description) is formulated, and verified by computing some *quality function*. This function, based on statistical techniques, computes the correctness of the hypothesis with respect to the training set. Next, the hypothesis is either accepted, rejected, or improved until a correct hypothesis is found, as shown in Figure 1.1. A hypothesis is improved by e.g. adding conditions on attributes, or generalizing conditions. When multiple alternatives exist for the new hypothesis, the choice is guided by *heuristics, constraints* and *user-interaction*.

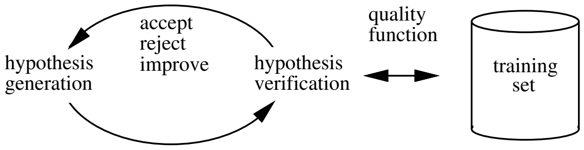


Figure 1.1: The iterative search for inductive information.

A machine learning system uses a small set of carefully selected laboratory data, and sometimes has the ability to interact with its environment, i.e. it can request new examples to investigate the behavior of the environment under particular conditions.

1.3 Data mining

The search for descriptions is called data mining when the training set is a database. A database is large, and contains data that has been generated and stored for purposes, other than learning processes. This data tends to be noisy and values for attributes are often missing. Moreover, the data represents only a small set of all possible behavior, and the system cannot manipulate its environment to generate interesting examples, as in machine learning.

Hence, it is harder to discover descriptions than in the ideal conditions found in machine learning. The size of the database makes verification of hypotheses a costly process. To reduce these costs, advanced database techniques, such as browsing optimization and caching are used. Statistical techniques are used to deal with noise and missing values. We discuss these problems and their solutions in Chapter 5.

It seems worthwhile at this point to explicate that although statistical techniques may seem ubiquitous in data mining, data mining should not be identified with statistics. An important difference is that a data mine system assists the user in the generation of hypotheses.

In this paper, propositional logic is the vehicle chosen to represent the (inferred) knowledge. Many other representations are, of course, possible. In Chapter 6 some alternative representations are introduced and their relative merits are discussed.

The seventh and eighth chapter give an extended overview of current data mining research placed in the framework developed in the first six chapters.

Chapter 2
Inductive learning

In the previous chapter we stated that inductive learning is the creation of a model of the environment. Such a model consists of classes, representing similar objects in the environment, and rules describing changes in the environment. These models are the topic of the first section of this chapter. Data mining, as we will discuss in following chapters, is a simple form of inductive learning, requiring relatively simple models, therefore we restrict our discussion to such simple models.

The second section discusses the creation of models by outlining some inductive learning techniques. The actual construction of inductive expressions is the topic of following chapters.

There are, of course, many possible models that can be created from a finite number of observations of the environment. Hence, criteria such as correctness and validity are needed to select an appropriate model. These criteria are the topic of the third and final section of this chapter.

1. Models

Models are used to predict changes in the environment, and to allow the cognitive system to interact more successfully with this environment. For an extensive discussion of models and inductive learning, the reader is referred to Holland *et al.* in [20].

1.1 Environment

The environment of a cognitive system depends on the context, it may be defined in very local terms (a chess board, or all customers of a sales company), or as the whole of the universe, including the system itself.

The status of the environment at a particular moment t is described by a *state* S_t. This state describes the objects in the environment together with their properties and mutual relationships. The state of the environment changes over time[1], so a subsequent state S_{t+1} may contain new objects and relationships, or objects may have disappeared, and properties of objects may have changed. So implicit to each environment is a *state transition function*

[1]For the sake of technical simplicity we will treat time as moving forward in discrete units.

$\mathcal{T}$ that specifies how the environment changes over time.

Definition 1. Environment The environment is a state transition system, i.e., a pair $(\mathcal{S}, \mathcal{T})$, where $\mathcal{S}$ is the set of all possible states and $\mathcal{T}$ is the transition function $\mathcal{T} : \mathcal{S} \rightarrow \mathcal{S}$. $\mathcal{T}$ defines the next state S_{t+1} for any state S_t. ∎

Example 1 Assume that the state consists of a single object, with properties 'color is red', and 'moving fast'. In the next state, the property 'color' of the object remains unchanged, but the 'moving fast' property has changed to 'motionless', obeying the law that all moving objects slow down. ∎

A straightforward way to create a model of the environment would be to make a faithful internal copy of this state transition system. That is, we simply store all states encountered so far and, moreover, we record all transitions. To predict the next state from the current state, we simply compare the current state with all recorded states. For example, one could learn to play chess, by simply observing games, and memorizing the appropriate next move for any of the possible configurations of pieces on the chess board. Unfortunately, this representation is only feasible for simple environments, consisting of a small variety of states. Besides the enormous amount of storage, needed to represent all these states, it is – for realistic environments – very unlikely that the current state will exactly match any of the previous states.

So, rather than making a faithful copy, we will use abstractions. We only use a small number of properties to characterize the objects in a state. Objects in the environment that satisfy the same subset of properties are mapped to the same internal representation.

At this point, we will make a very important restriction. Most data mine systems search for relationships within objects, i.e. relationships among properties in a single object, and not for relationships between different objects. Hence, in our perception of the environment, objects are completely unrelated, we assume that the effect of $\mathcal{T}$ on an object in S does not depend on other objects in S. Note that this assumption, although implicit to most data mine systems, is absurd, since objects in a database are seldomly unrelated (e.g. objects may have unique keys, and other, unknown, constraints may exist in a database).

1.2 Classes

Since a state is described using a limited set of properties, distinct objects in the environment may be represented internally as the same object. In other words, choosing a particular set of properties to be represented in the model induces *equivalence classes* of objects. To each class corresponds a unique pattern of values, the *class description*.

The set of all classes is denoted $\mathcal{C}$, and to each class C_i corresponds a description D_i of values for the selected properties. So, using these descriptions, we can construct a *classification function* $P : S \rightarrow \mathcal{C}$, that maps an object o in state S to class C_i when o satisfies D_i. P extends straightforwardly to a function $P : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{C})$, that maps each state to the set of classes corresponding with the objects in S.

Example 2 If we select the property 'speed' in the above example, we can distinguish two classes, 'fast moving' and 'motionless'. On the other hand, we could also select the property 'color', which – in the absence of other objects – induces a single class 'color is red'. ∎

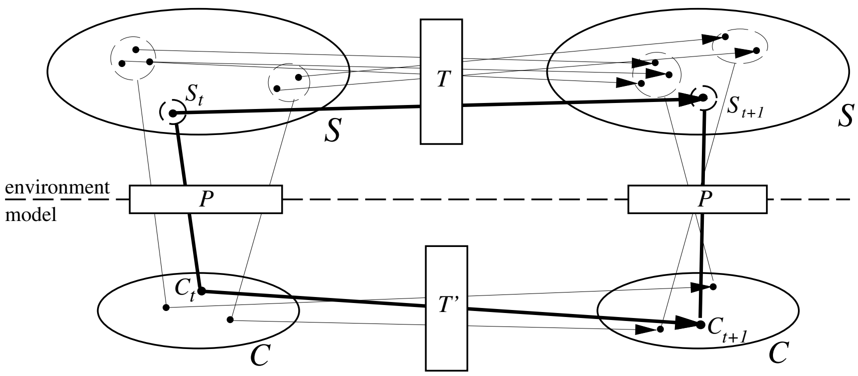


Figure 2.1: An internal model representing transitions in the environment.

1.3 Model transition function
With this internal representation, the system should construct a model transition function $\mathcal{T}'$, which is intended to mimic the transition function $\mathcal{T}$, operating in the real world. The function $\mathcal{T}'$ describes how the internal (class) representations $C_{t,i}$ of an environmental state S_t, leads to the internal representations $C_{t+1,j}$ of the subsequent state S_{t+1}.

EXAMPLE 3 In our example, we could construct a model transition function mapping the class 'fast moving' onto 'motionless', thus predicting that any fast moving object will slow down. ∎

We illustrated this model in Figure 2.1, where an environmental state S_t in the set of all states $\mathcal{S}$ is transformed to a state S_{t+1}. To simplify the picture, we assume that the internal representation of this state is just one class C_t and application of the model transition function $\mathcal{T}'$ to this class results in class C_{t+1}, which is the internal representation of state S_{t+1}.

1.4 Correctness of the model
A model is correct if it predicts the next state correctly, i.e. if, at any moment t, the internal representation of the next state is identical to the representation, predicted by the model. Since both the environment and the model are formalised as a state transition function, this means that P should be a homomorphism between state transition systems. In other words, the diagram must be commutative:

$$P(\mathcal{T}(S_t)) = \mathcal{T}'(P(S_t))$$

2. LEARNING
In the beginning, the cognitive system does not have any classes to represent environmental states internally, and it has no model transition function. Moreover, when the system already has a model to represent its environment, it can become inadequate because the environment is changing. For successful operation in such an environment, the system has to be adaptive, that is, it should *learn*.

Learning consists of finding both an appropriate internal representation, i.e. classes, and a model transition function, acting on this representation. There are various learning strategies, as discussed in [8]. These strategies differ in the amount of inferential skills, required by the system. In *learning by being told*, knowledge is acquired from a teacher or any other organized source, such as a textbook. Here, the only activity, performed by the system, is the translation of this knowledge to an internal format. Another form is *learning from analogy*, where the system generates new rules by transforming existing ones to make them applicable to new, but similar situations.

However, we are mainly interested in inductive learning strategies where the system infers knowledge itself, i.e. from observing its environment. There are two strategies that we will use:

2.1 Supervised learning

In supervised learning or learning from examples, the teacher supports the system in the model construction, by defining classes and supplying examples (i.e. pre-classified objects) of each class. The system has to find the description for each class. The teacher can either define a single class or multiple classes (see [14]):

Single class learning The teacher defines a single class C for which a so-called *characteristic description* has to be constructed: a description that singles out instances of C from any other example that is not an instance of C. One can distinguish two cases: all examples, provided by the teacher, are members of this class (the so-called *positive* examples), and the system has to construct a description for these.

Alternatively, when both positive and negative examples are provided, the negative examples can be seen as members of (infinitely many) other classes. Useful are the *near-misses*, examples that just fall outside the class and provide interesting information on the class boundaries.

Example 4 In an animal environment, we can define the class ‘bird’, and search for a characteristic description, where all birds serve as positive examples, and all other animals (e.g. fishes) as negative examples. An interesting example is the penguin—a bird that does not fly, but has wings, thus allowing us to induce that the characteristic property is wings, and not the ability to fly. ∎

Multiple class learning The teacher defines a finite number of classes $C_1, C_2, \ldots, C_m$, for which descriptions have to be found. The system can either search for *characteristic* descriptions, thus descriptions that distinguish instances of C_i from any other example, as in single class learning. Instances of C_i form the positive examples, and all objects not in C_i form the negative examples.

Alternatively, if each example belongs to at least one class, the system can search for *discriminating* descriptions: descriptions that together cover all objects, and separate an instance of a class from instances of all other classes.

Example 5 In the animal environment, we can construct discriminating descriptions if we assume that all animals belong to either the class ‘fish’ or ‘bird’. Possible descriptions are that birds have wings and fishes do not have wings. These descriptions are sufficient to distinguish fishes from birds. However, if the set of examples contains other animals, belonging to (unknown) classes, characteristic descriptions have to be constructed, such as, fishes have fins, and birds have wings, to distinguish birds and fishes from other animals. ∎

2.2 Unsupervised learning

In unsupervised learning, or learning from observation and discovery, the system has to find its own classes in a set of states, without any help of a teacher. Practically, the system has to find some clustering of the set of states $\mathcal{S}$. The data mine system is supplied objects, as in supervised learning, but now, no classes are defined. The system has to observe the examples, and recognize patterns (i.e. class descriptions) by itself. Hence, this learning form is also called *learning by observation and discovery* [29]. The result of an unsupervised learning process is a set of class descriptions, one for each discovered class, that together cover all objects in the environment. These descriptions form a high-level summary of the objects in the environment.

We believe that unsupervised learning is actually not different from supervised learning, when only positive examples of a single class are provided. Thus, we search a description that describes all objects in the environment. If different classes exist in the environment, this description will be composed of the descriptions of these newly discovered classes.

Example 6 For the animal database, we could find a description stating that the database consists of three classes: flying animals with wings, swimming animals with fins and a singleton class for the exception, a swimming penguin with wings. This description is correct, i.e. it covers all objects in the environment, and moreover, it is a simplification of the environment. ∎

3. Quality

The most important problem with inductive learning is that, given a set of examples, the system can construct multiple models that are correct with respect to these examples. That is, models that correctly predict the next state for all environmental states the system has encountered so far. However, if the model is to be used to predict the outcome of future situations, than it should not only be correct for states it has seen so far, but also for any unseen state that could occur.

Example 7 In the previous example, the system might have concluded that any animal that does not have legs is a fish. This conclusion is correct with respect to the examples. ∎

The problem – addressed by many philosophers – is that we must distinguish between really existing relationships among states and apparent relationships, that are not general valid, but occur only because of the limited number of examples.

Since, for most environments, the number of possible states is infinite, the correctness of a model cannot be verified by checking it for all possible situations. Therefore, we need to estimate the validity of a model: if we can construct multiple models, some of these will be simpler than others. We expect that simpler models are more likely to be correct. The rationale behind this rule– also known as Ockham's razor [58] – is that if there are multiple explanations for a particular phenomenon, it makes sense to choose the simplest, because it is more likely to capture the nature of the phenomenon.

Chapter 3
Data mining

The learning processes described in the previous chapter can also be performed by computers. The study and computer modeling of these processes is the subject of a research area called *machine learning.*

Generally, a machine learning system does not use single observations of its environment, as cognitive systems do, but an entire, finite, set – called the *training set* – at once. This set contains examples—observations coded in some machine readable form. When we use a database as a training set, the learning process is called *data mining.* In this chapter, we describe the training sets, patterns and rules as used in most data mine tools.

Given that the training set is finite, not all concepts can be learned exactly. In fact, even with infinite training sets some concepts cannot be learned exactly, since some problems are undecidable. However, some of these concepts can be *approximated.* Although computational learning theory, which studies these problems, is outside the scope of this survey, see e.g. [3], we outline the basic ideas in the last section of this chapter.

1. Induction from databases

As described in the previous chapter, learning is tantamount to the construction of rules, based on observations of environmental states and transitions. Automation of a learning process is called *machine learning. Data mining* is a special kind of machine learning where the environment is observed through a database.

1.1 Machine learning versus data mining

Figure 3.1 depicts the general framework for machine learning. The environment E represents the real world, the environment that is learned about. E represents a finite number of observations, or objects, that are encoded in some machine readable format by the encoder C. The set of encoded observations is the *training set* for the learning algorithm ML.

The general framework for data mining, Figure 3.2 is a variation of the machine learning framework. The encoder C is replaced by the database DB. In the terminology of the previous chapter, we can say that the database DB models the environment E. Each state from the database reflects a state from E, and each state transition of DB reflects a state transition

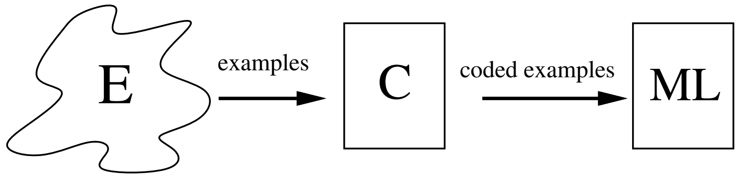


Figure 3.1: Machine learning diagram.

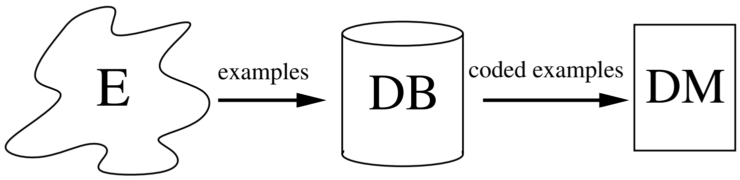


Figure 3.2: Data mining diagram.

in the environment E.

The learning algorithm in turn builds a model from DB. As far as the classes are concerned, this means that the algorithm has to infer the rules that govern the classification of database objects. The rules that govern the transitions between classes should be inferred from the transitions in the database. Note, however, that like most data mine tools we restrict ourselves to the learning of the classification rules.

Although the frameworks for data mining and machine learning in general may seem very similar, there are important distinctions. First and foremost is that the database is often designed for purposes different from data mining. That is, the representation of the real world objects in the database has been chosen to meet the needs of applications rather than the needs of data mining. Hence, properties or attributes that would simplify the learning task are not necessarily present. Moreover, these properties can not be requested from the real world.

The second important difference is that databases are invariably contaminated by errors. Whereas in machine learning the algorithm is often supplied with judiciously chosen laboratory examples, in data mining the algorithm has to cope with noisy and sometimes contradictory data.

For a discussion of the effect of these differences on the learning process, see Chapter 5.

1.2 The training set

The database, customarily called the *training set* S for the learning algorithms, represents information about the environment. In principle there are many possible representation formalisms, however, since most current DBMSs are relational or support at least an SQL-interface, we assume that our database is relational. In other words, objects in the environment are represented by tuples in the database.

The assumption made in the previous chapter implies that the tuples in the database only represent properties of the objects and not relationships between those objects. That is, each object in the environment is represented by a tuple and a tuple represents one or more objects in the environment. A tuple can only represent more than one object, if the distinction between those objects is deemed to be irrelevant.

The second assumption we make is the *Universal Relation Assumption.* That is, we assume that the database consists of a single table. Of course, values in this table may be *Nill* or unknown. This assumption may seem as severe as the above no relationships assumption. However, it is well-known that the universal relation assumption can be applied to all database schemata and queries (see [60]). In fact, the universal relation assumption has been defined as a less cumbersome user interface, and that is exactly the use we will make of it.

DEFINITION 1. TRAINING SET Let $\mathcal{A} = \{A_1, \cdots, A_n\}$ be a set of attributes with domains $Dom_1, \cdots, Dom_n$. A *training set* is a table over $\mathcal{A}$. An example, or fact, is a tuple in a training set. The *Universe* $\mathcal{U}$ is the full relation over $\mathcal{A}$, i.e., $\mathcal{U} = Dom_1 \times \cdots \times Dom_n$. Hence, each training set is a finite subset of $\mathcal{U}$. ∎

The final assumption that we make is that each domain, and hence, $\mathcal{U}$ is assumed to be finite.

EXAMPLE 1 We can represent the animal environment, described in the previous chapter, as a training set. Each fact describes an animal, by enumerating properties of this animal. Hence, the training set is:

species	wings/fins	transportation
hawk	wings	fly
swan	wings	fly
penguin	wings	swim
shark	fins	swim
trout	fins	swim

where the first attribute is the name of the species, the second denotes whether it has wings or fins and the third argument describes its means of transportation. The domain Dom_1 is the set {hawk, swan, ..., trout}, Dom_2 = { wings, fins} and Dom_3 = {swim, fins}. ∎

1.3 Classes

The database is the environment for the data mine tool, that is, the data mine tool has to infer a model from the database. In the supervised learning case, this requires that the user defines one or more classes, also known as *concepts* [29], in the database. Without loss of generality, we may assume that the database contains one or more attributes that denote the class of a tuple, these attributes are called the *predicted attributes.* The remaining attributes are called *predicting attributes.*

A combination of values for the predicted attributes defines a class, or, more general, a class is defined by condition on the attributes.

DEFINITION 2. CLASS A class C_i is a subset of the training set S, consisting of all objects that satisfy the class condition $cond_i$:

$$C_i = \{o \in S \mid cond_i(o)\}$$

Objects that satisfy the condition $cond_i$ are *positive* examples or *instances* of the class C_i. The examples outside this subset of the training set are *negative* examples. ∎

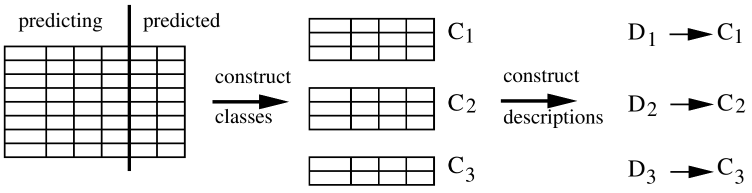


Figure 3.3: Learning classification rules from a database.

Learning classification rules means that the system has to find the rules that predict the class from the predicting attributes, see Figure 3.3. First, the user has to define the conditions for each class, and thus partition S into subsets $C_1, \ldots, C_n$. Then, the data mine system has to construct descriptions $D_1, \ldots, D_n$ for these classes. There are different techniques to define these classes in terms of the predicted attributes:

Conditions on attributes We can define a class in terms of the predicted – and possibly some predicting – attributes. For example, in a medical database, we can define a class 'flu' as a condition 'diagnosis = flu', where diagnosis is a predicted attribute. Hence, any example in the training set, for which the attribute diagnosis has value 'flu', belongs to class 'flu'.

Conditions can also include predicting attributes, e.g., we can define a class 'profitable' in a company's financial transaction database, as 'income > expenses', where 'expenses' is a predicting attribute and 'income' is a predicted attribute.

Classes can also be used incorporate information that is not explicitly represented in the database itself:

Multiple databases The training set can be composed from multiple databases. For example, we can search for differences between customers in Texas and in New York. All customers originally stored in the local database in Texas belong to the class 'Texas', and all customers from New York belong to class 'New York'.

Databases over time We can also look for *data evolution regularities*, that is, the discovery of global changes in a database over a period [16]. Thereto, we construct a training set, composed from two sets S_i and S_j, taken from the same database at different times. All examples originally from S_i belong to class 'at Time = i' and examples in S_j belong to class 'at Time = j'.

If both sets S_i and S_j represent the *same* objects (but at different moments), we can use S_i as the training set, and define a class in terms of S_i and S_j. For example, the class 'doubled profits', containing all companies which doubled their profits over the period, where profit is an attribute).

We can either define disjunctive or overlapping classes, or we can even define entire hierarchies of classes. In the latter case, an example can belong to multiple classes.

1.4 Clustering

If the user does not define classes, the learning is unsupervised. As we have seen in the previous chapter, the system has to discover its own classes, i.e. the system clusters the data

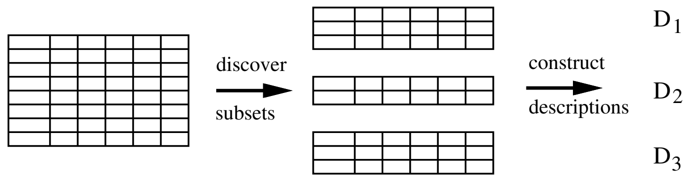


Figure 3.4: Discovering clusters and descriptions in a database.

in the database. This is illustrated in Figure 3.4. First, the system has to discover subsets of related objects in the training set, and then it has to find descriptions (e.g. D_1, D_2, and D_3) that describe each of these subsets.

To illustrate the complexity of unsupervised learning, we give the number of possible clusterings of a database with N tuples. If there are m clusters in the database, we partition the set of N tuples in m non-empty disjunct subsets. Let us denote by $P(N, m)$ the number of ways in which this can be done. In general the number of ways to partition N elements in m subsets is given by [15]:

$$P(N,m) = \frac{1}{m!}\sum_{j=0}^{m}\binom{m}{j}(-1)^j(m-j)^N$$

$P(N, m)$ is a function that grows exponentially fast in N, in fact, for large N we have $P(N,m) \approx m^N/m! \approx m^{N-m}e^m\sqrt{2\pi m}$. The total number of ways in which a database of N tuples can be clustered, denoted by $C(N)$ is:

$$C(N) = \sum_{m=1}^{N} P(N,m) = \sum_{m=1}^{N}\frac{1}{m!}\sum_{j=0}^{m}\binom{m}{j}(-1)^j(m-j)^N$$

For example, for $N = 8$, we have:

m	1	2	3	4	5	6	7	8
$P(8,m)$	1	127	966	1701	1050	266	28	1

Hence, $C(8) = \sum_{m=1}^{8} P(8,m) = 4140$. Clearly, the data mining process is only successful if the number of classes in the database is small. In other words, $C(N)$ is too large as an estimate. However, even if we assume that the number of classes we expect in the database is small compared to the size of the database, say 10, the number of possible clusterings, i.e., $\sum_{m=1}^{10} P(N,m)$, is still staggering.

1.5 Rules

When the classes are defined, the system should infer the rules that govern the classification. That is, the system should find the description of each class. These descriptions should, of course, only refer to the predicting attributes of the training set. Ideally, all of the positive examples should satisfy the description and none of the negative examples.

Since we only store attributes of objects in the training set, thus no relationships among objects, descriptions can only consist of conditions on these attributes. The descriptions commonly used in data mine tools are a subset of the selection conditions from the relational algebra.

DEFINITION 3. DESCRIPTION Let $\mathcal{A}$ be the set of predicting attributes. An *elementary description* is a formulae $A_1 = c_1 \wedge \cdots \wedge A_n = c_n$ such that:

1. $A_i \in \mathcal{A}$ and $i \neq j \rightarrow A_i \neq A_j$;
2. $c_i \in Dom(A_i)$.

A pattern or *description* is a (non-empty) disjunction of elementary descriptions. A condition $A_i = c_i$ is called an *attribute-value condition*. The set of all possible descriptions is called the *description space* and is denoted by $\mathcal{D}$. ∎

Some systems, e.g. LEX [36], only try to find elementary descriptions, while others allow the full set of descriptions. Since the domain are assumed to be finite, we can count the number of descriptions these systems can find.

For an elementary description ϕ, either $A_i = c_i$ for some $c_i \in Dom(A_i)$ is a conjunct of ϕ *or* A_i does not occur in ϕ. Hence, the number of elementary descriptions is given by:

$$\prod_{\mathcal{A}}(|Dom(A_i)| + 1)$$

Note that the number of descriptions depends on the size of the domains, and not directly on the size of the training set. So, the fraction of (non-empty) subsets of $\mathcal{U}$ that can be described with elementary descriptions is:

$$\frac{\prod_{\mathcal{A}}(|Dom(A_i)| + 1)}{2^{\prod_{\mathcal{A}}|Dom(A_i)|} - 1} \approx \frac{|\mathcal{U}|}{2^{|\mathcal{U}|}}$$

which vanishes quickly for growing $|\mathcal{U}|$. The total number of descriptions is simply given by:

$$2^{\prod_{\mathcal{A}}(|Dom(A_i)|+1)} - 1$$

That is, there are more descriptions than possible classes. However, not all of these descriptions are distinct, since the fact that A_i does not occur in ϕ is equivalent with allowing A_i to take any value in $dom(A_i)$. Thus, as to be expected, the total number of distinct descriptions is $2^{\prod_{\mathcal{A}}|Dom(A_i)|} - 1$. That is, there is a description for each non-empty subset of $\mathcal{U}$.

All examples that satisfy a description D, are said to be *covered* by D. In other words, the examples in $\sigma_D(S)$, i.e. the (relational algebra) selection D from set S, are the examples covered by D.

For notational convenience, we allow the value of an attribute c_i in a description in a description to be a set of symbols S_i, rather than a single symbol. This set S_i must be a subset of $Dom(A_i)$. This is nothing but a (purely syntactical) shorthand notation for the disjunction of descriptions[1]:

$$\begin{gathered} A_1 \in \{c_{1,1}, \cdots, c_{1,m_1}\} \wedge \cdots \wedge A_n \in \{c_{n,1}, \cdots, c_{n,m_n}\} = \\ (A_1 = c_{1,1} \wedge \cdots \wedge A_n = c_{n,1}) \vee \cdots \vee (A_1 = c_{1,m_1} \wedge \cdots \wedge A_n = c_{n,m_n}) \end{gathered}$$

[1]Actually, this is the (UN)NEST operation in relational algebra.

There are systems (e.g. CN2, see Chapter 7), that only try to find descriptions of this form, which we will dub *set-descriptions* in this paper. That is, set-descriptions are of the form $A_1 \in \{c_{1,1}, \cdots, c_{1,m_1}\} \wedge \cdots \wedge A_n \in \{c_{n,1}, \cdots, c_{n,m_n}\}$. Not every descriptions can be written as a set-description, i.e. set-descriptions form a subset of the set of all constructible descriptions. The number of set-descriptions is given by:

$$\prod_{\mathcal{A}} (2^{Dom(A_i)} - 1) \approx 2^{\sum_{\mathcal{A}} Dom(A_i)}$$

In other words, the fraction of classes that can be described by set-descriptions is approximated by:

$$\frac{2^{\sum_{\mathcal{A}} Dom(A_i)}}{2^{\prod_{\mathcal{A}} |Dom(A_i)|}}$$

Since the domains are finite, we will freely use expressions like $A_i \leq c_i$ and $A_j \neq c_j$, which can be rewritten to standard descriptions in the obvious way. Moreover, if the domain is linear, $[a, b]$ denotes the set of all symbols, between and including a and b.

Definition 4. Rule A classification rule consists of a *description* D and a *class symbol* C:

$$\forall o \in \mathcal{U} : o \in \sigma_D(S) \;\rightarrow\; o \in C$$

or simply 'if D then C, stating that any object that satisfies D, belongs to class C. ∎

Informally, a rule is correct with respect to the training set if its description covers each positive example, and none of the negative examples of the class. That is, if $\sigma_D(S) = C$.

Example 2 We will continue the animals example from the previous chapter, and define two classes, 'fish' and 'bird'. To each class corresponds a subset of the training set, in particular, 'bird' is a set, containing the hawk, swan and penguin examples, and 'fish' is a set containing the shark and trout. When the system searches descriptions for both classes, the following classification rules could be constructed:

if $A_2 = wings$ then *bird*
if $A_2 = fins$ then *fish*

stating that the unique pattern for birds is that their second attribute is wings, and that the second attribute of all fishes is fins. ∎

Note that the description for a class is not uniquely defined. Many different descriptions can be constructed that are correct with respect to the training set. Unfortunately, not all of these descriptions will correctly classify unseen examples. The problem, as discussed in the previous chapter, is that not all knowledge derived from observations is valid. In the next chapter, we discuss a rule of thumb that can be used to select those descriptions that are most likely to be valid.

2. Computational learning theory

Until now we discussed learning algorithms, but we tacitly avoided the question whether such algorithms exist and if they exist if such algorithms have a reasonable complexity in, e.g., the size of the training set. These issues are topics in the area of computational learning theory, of which a survey is outside the scope of this paper. However, in this section we briefly discuss some of its main results in as far as they apply to data mining. The interested reader is referred to, e.g., [3].

2.1 Learning by enumeration, or exhaustive search

Clearly, there exist concepts that cannot be learned by any algorithm. For example, one can create a training set that consists of algorithms and inputs for these algorithms and classify these on whether the algorithm terminates on that input or not. However, since termination is an undecidable property, we cannot hope that our learning algorithm learns to classify all algorithm input pairs correctly.

However, our goals for data mining are more modest than the halting problem. Let us forget for the moment that databases are polluted by noise. Then we can learn a description for a class by exhaustive search, also known as learning by enumeration. That is, in the supervised case, we simply try all descriptions and take that one that fits the class best. In the unsupervised case, we simply try all clusterings, and for each clustering we try all descriptions for all classes in this clustering. Finally, we pick the clustering and the set of descriptions that is deemed to be the best.

There are, however, two caveats with this "solution". The first is its staggering complexity, the second is that we implicitly assume that all possible examples of a class are actually present in the database. The complexity issue is discussed in this subsection, while the second problem is discussed in the next subsection.

EXAMPLE 3 Let S be a database with n attributes and N tuples. Assume that the user has defined classes, i.e. supervised learning. The complexity of exhaustive search clearly depends on the kind of description we are looking for. For example, there are more set-descriptions than elementary descriptions. Hence, the chance of finding the best description grows if we have more possible descriptions. The next table summarises this information from the previous section:

Descriptions	no. of trials	$P(success)$
elementary	$\prod_{\mathcal{A}}(\lvert Dom(A_i)\rvert + 1)$	$\lvert\mathcal{U}\rvert/2^{\lvert\mathcal{U}\rvert}$
set	$2^{\sum_{\mathcal{A}} Dom(A_i)}$	$2^{\sum_{\mathcal{A}} Dom(A_i)}/2^{\prod_{\mathcal{A}}\lvert Dom(A_i)\rvert}$
all	$2^{\prod_{\mathcal{A}}\lvert Dom(A_i)\rvert} - 1$	1

In the unsupervised case, matters even get worse. For we not only have to find the descriptions but also the classes. That is, for each possible class in each partitioning, we have to try each possible description. That is, if we denote the number of descriptions by $D(\mathcal{A}, N, i)$ and the number of hypotheses (i.e. a clustering together with the descriptions for its classes) by $H(\mathcal{A}, N, i)$, were i denotes the type of descriptions chosen, we have:

$$H(\mathcal{A}, N, i) = \sum_{m=1}^{N} D(\mathcal{A}, N, i) \times m \times P(N, m)$$

Summarizing from the previous section, we get the following table:

Descriptions	no. of trials
elementary	$\sum_{m=1}^{N}\prod_{\mathcal{A}}(\lvert Dom(A_i)\rvert + 1) \times m^{N+1}/m!$
set	$\sum_{m=1}^{N} 2^{\sum_{\mathcal{A}} Dom(A_i)} \times m^{N+1}/m!$
all	$\sum_{m=1}^{N} 2^{\prod_{\mathcal{A}}\lvert Dom(A_i)\rvert} \times m^{N+1}/m!$

For each of the entries, the chance of success is the same as that in the previous table. ∎

2.2 Probably approximate correct learning

In our description of learning by enumeration, we assumed that all positive examples of class C are contained in the training set S. However, since the training set generally represents only part of the universe $\mathcal{U}$, this is assumption is very unlikely to be correct. All we can hope for is that some positive examples of C are present in the training set.

The implication of the fact that not all possible examples of C are in S, is that we cannot know for certain that the description we find for $S \cap C$ is the description that describes the class C correctly, i.e. covers all its *possible* instances. However, we would like to have some certainty about the correctness of our description. More precisely, we want to minimise the chance that our rule 'if D then C' mis-classifies an (unseen) example.

This notion of certainty is formalised through the notion of *Probably Approximately Correct learning* or PAC-learning [62]. Before we formulate this notion, we first recall some basic concepts from probability theory.

A probability distribution on $\mathcal{U}$ is a function[2] $\mu : \mathcal{P}(\mathcal{U}) \rightarrow [0,1]$ such that:

1. $\mu(\emptyset) = 0$;
2. $\mu(\mathcal{U}) = 1$;
3. for pairwise disjoint sets $S_1, \ldots, S_n \in \mathcal{P}(\mathcal{U})$, $\mu(\bigcup_{i=1}^{n} S_i) = \sum_{i=1}^{n} \mu(S_i)$.

For a set $E \subseteq \mathcal{U}$, $\mu(E)$ denotes the change that a randomly chosen $x \in U$ belongs to E. If we use C as a predicate, with $C(x) = \mathit{true}$ iff $x \in C$, then we want to minimise:

$$Er_{\mu}(D,C) = \mu\{x \in \mathcal{U} \mid D(x) \neq C(x)\}$$

For $Er_{\mu}(D,C)$ is the probability that our rule misclassifies an example. Clearly, the description D we discover depends on the database we use. So, if we denote by $L(C,S)$ the description our learning algorithm L finds for class C in a database S, we want to minimise:

$$er_{\mu}(L,C,S) = \mu\{x \in \mathcal{U} \mid L(C,S)(x) \neq C(x)\}$$

Since we cannot choose a particular database state, but are given one at random, we want to minimise $er(L,C,S)$ regardless of the particular database state S. To formalise this, we need the distributions $\mu^n : \mathcal{P}(\mathcal{U}^n) \rightarrow [0,1]$, where $\mu^n(Y)$ denotes the chance that a random sample of n elements belongs to Y. Then:

DEFINITION 5. A n algorithm L learns $\mathcal{C}$ *probably approximately correct* or *PAC* if

$$\begin{array}{l} \forall 0 < \epsilon < 1,\ \forall 0 < \delta < 1,\ \exists m_0,\ \forall c \in \mathcal{C},\ \forall \mu,\ \forall m \geq m_0 : \\ \mu^m\{S \mid m = |S| \wedge er_{\mu}(L,C,S) < \epsilon\} \ > \ 1 - \delta \end{array}$$

∎

In other words, if we choose a small ϵ and δ, then for almost all databases whose size exceeds m_0 result in a rule that is almost always correct.

A learning algorithm is called *consistent* if it is correct on its training set. An important result in computational learning theory implies that all consistent algorithms for data mining are PAC, under the assumption that $\mathcal{U}$ is finite.

[2] Remember, $\mathcal{U}$ is assumed to be finite.

The importance of this result for data mining in practice, is however, far from clear for two reasons. The first reason is that databases invariably suffer from missing information. Hence, we cannot demand that our learning algorithm is consistent on a given database. The second, and perhaps even more important, reason is that the proof of the theorem depends on the assumption that the examples in a database are independent. However, real life databases are invariably subject to all kinds of, perhaps even unknown, constraints. Through these constraints, the examples in the database are no longer independent. In other words, the theorem no longer applies.

Chapter 4
Search algorithms

Given a training set, and possibly some user-defined classes, a data mine system can construct many descriptions. Some of these descriptions are more correct than others, i.e. some descriptions are more likely to classify unseen examples correctly and thus describe some of the – unknown – relationships that underly the data. So, once we have defined a measurement for the quality of a description, the construction of a description is nothing but a *search problem*: finding the best description in the set of all constructible descriptions $\mathcal{D}$.

As we have seen in the previous chapter, this set is generally too large to explore using an exhaustive search. Hence, we need a more efficient search algorithm. Most data mine systems choose an initial description, and iteratively modify it, thereby improving its quality. These modifications are operations on the description. The set of descriptions, together with these operations, and the quality function, is called the *search space* (see [41]), as we will discuss in the first section.

The second section is devoted to search strategies: the choice of an initial description, and the order in which different alternatives are explored, e.g. back-tracking or irrevocable search. Often, a description can be modified using multiple operations. To find a correct solution as fast as possible, we have to select the most promising operation. In the third section, we will discuss how heuristics and domain specific knowledge can be used to guide the search process.

In the fourth, and final section, we review alternatives search strategies, such as genetic algorithms and simulated annealing.

1. Search space

The search space $\langle \mathcal{D}, f, \mathcal{O} \rangle$ consists of a set of descriptions $\mathcal{D}$, a set of operations on these descriptions $\mathcal{O}$ and a quality function f.

1.1 Description space

The description space $\mathcal{D}$ is the set of all constructible descriptions, in a particular representation. In this chapter, we adopt the set-descriptions, as introduced in the previous chapter. To each description D in $\mathcal{D}$ corresponds a subset of the training set S, called the *cover* $\sigma_D(S)$.

1.2 Operations

We can divide the set of all operations $\mathcal{O}$ in *generalization* operations, that 'weaken' the description, i.e. make it cover more objects, and *specialization* operations, that 'strengthen' the description, and reduce its coverage.

Generalization Applying a generalization operation to a description D, results in a new description D' that covers more objects i.e. $\sigma_D(S) \subseteq \sigma_{D'}(S)$. Hence, if an object is covered by the description D, it is also covered by description D', but the reverse does not hold. Hence, the generalization operation is not truth preserving, i.e. if a rule classifies the objects correctly, then a generalization of this rule need not be correct as well. However, generalization rules are falsity preserving: if an object is incorrectly classified by the rule (i.e. the object is covered by D, but is not an example of class C, so the object falsifies the rule), then it will also falsify any of the generalizations of this rule.

DEFINITION 1. GENERALIZATION A set-description $D = (A_1 \in S_1 \wedge \cdots \wedge A_i \in S_i \wedge \cdots \wedge A_n \in S_n)$ is generalized by extending the set of values for a particular attribute A_i to S_i' where $S_i \subset S_i' \subseteq Dom_i$. ∎

A special case of this operations is the *dropping condition* operation, where the set S_i is extended to the entire domain Dom_i. Another special case, used in e.g. the DBLearn system, is the *climbing generalization tree* operation. Often, we want to make generalizations that provide a more compact representation of the class. Replacing sets of permissible values with larger sets is obviously not an operation that results in a simpler description. Therefore, it would be convenient if we could replace entire sets of symbols with single symbols.

We define a partial order on the symbols in the domain of attribute A_i, that is, the values in Dom_i form a hierarchy. A subset W of S_i is replaced with a single value s, that is larger than all values in W. These generalization hierarchies are a form of domain specific knowledge, as we will discuss in the following section.

Specialization The specialization operation is the inverse of the generalization operations, where the set S_i is replaced with S_i', and $S_i \supset S_i'$. We will not discuss this operation into detail, since its definition is straightforward.

Now we have defined descriptions, and operations on descriptions, we can represent the search space as a directed graph, where descriptions form the nodes and the operations form the arcs.

1.3 Domain of the attributes

The generalization and specialization operations need some information about the structure of the domain of the attributes. For each of the domains Dom_i, the user has to specify the structure of the domain, which can be one of the following basic types:

Nominal (categorical) The domain consists of independent symbols or names, i.e. the values are unordered. For example, blood-types or first names.

Linear The domain is totally ordered, e.g. a numerical domain. This allows us to use intervals instead of sets of values. Linear domain are either *ordinal*, such as {low, medium, high}, where an ordering is defined, but other operations, say 'low + low = medium', make no sense. Alternatively, the domain can be an *interval* domain, where addition is defined, but multiplication is not defined, e.g., the temperature $20^o C$ is not twice as warm as $10^o C$. Finally, the domain can be a *ratio* domain, such as numbers, where both addition and multiplication have useful interpretations.

Partially ordered The domain is partially ordered, thus forming a hierarchy, where a parent node denotes a more general concept than its children. By definition, any symbol is smaller than the top-symbol, which denotes the entire domain.

For example, the user could define that 'Holland' is a parent node for 'Amsterdam', 'Rotterdam' and 'The Hague'.

1.4 Quality function

The quality function assigns a value, e.g. in the domain $[0, 1]$, to each description, indicating its quality. There are two aspects to the quality of a description. A description should be general *valid*, that is, it should classify any unseen object correctly. Furthermore, the description should be *correct* with respect to classes defined by the user. We can combine these criteria, by assigning a value for each criterion, and use a function to compute the overall quality. We first discuss validity, then correctness, which differs for supervised and unsupervised learning, and finally we discuss how these criteria can be combined.

Validity In general, the validity of a rule can never be proven, because its correctness cannot be verified for all possible situations. Hence, we need some indication for the likelihood that a description is valid. Most data mine systems rely on Ockham's razor: the simpler a description, the more likely it is that it describes some really existing relationship in the database. This complexity can be measured e.g. in the size of the description. The validity f_v is higher for simpler descriptions.

Correctness, supervised learning A description D for a class C is correct if it covers all positive, and none of the negative examples, i.e. if $\sigma_D(S) = C$.

During the iterative process of constructing a correct description, the system will encounter many descriptions that are not correct, but nevertheless useful, because the serve as components for new, and hopefully better, descriptions. To be able to select the most promising description out of a set of incorrect descriptions, we need to extend our notion of correctness, by allowing it to be a continuum of values, rather than a boolean. With each description D for a class C, we can associate the following values.

Definition 2. Classification accuracy We define the classification accuracy as the probability that the rule classifies correct, thus the probability that an object covered by the description actually belongs to the class. The classification accuracy is the relative portion of $\sigma_D(S)$ that is also covered by C:

$$\textit{classification accuracy} = \frac{|\sigma_D(S) \cap C|}{|\sigma_D(S)|}$$

∎

Definition 3. Coverage We define the *coverage* of a description as the probability that an arbitrary object, belonging to the class C, is covered by the description D:

$$coverage = \frac{|\sigma_D(S) \cap C|}{|C|}$$

∎

Based on these values, we can distinguish the following kinds of rules:

Definition 4. Complete rules If the coverage equals 1, the rule is complete, that is, any object belonging to the class is covered by the description for this class, i.e. $C \subseteq \sigma_D(S)$. In other words, the description is a necessary condition for the class [7, p. 222]. ∎

Definition 5. Deterministic rules If the classification accuracy is 1, the rule is deterministic, i.e. always classifies correct. Any object covered by the description belongs to class, $C \supseteq \sigma_D(S)$. Hence the description is a sufficient condition for the class. ∎

Definition 6. Correct rules If both the classification accuracy and the coverage are 1, the rule is correct as we defined above. The description is both a necessary and sufficient condition. ∎

The correctness-criterion f_c is assigned a value 1 if the description is correct, the value for any incorrect rules is smaller than 1. In [43], Piatetsky-Shapiro proposes principles for the construction of a function which assigns a numerical value to any description in $\mathcal{D}$, indicating its correctness. This correctness depends on the size of the set $\sigma_D(S)$, covered by the description, the size of the class C and the size of their overlapping region $\sigma_D(S) \cap C$, such that:

1. $f_c = 0$ if D and C are statistically independent. If the probability that a particular example covered by description D belongs to class C as well is equal to the probability that an arbitrary example in the set of examples S belongs to C, i.e.

$$\frac{|\sigma_D(S) \cap C|}{|\sigma_D(S)|} = \frac{|C|}{|S|}$$

   then D and C are statistically independent, and the description is not interesting.

2. f_c monotonically increases with $|\sigma_D(S) \cap C|$: the number of tuples in the overlapping region, when $\sigma_D(S)$ and C remain the same.

3. f_c monotonically decreases with $|\sigma_D(S)|$ (or $|C|$) when $\sigma_D(S) \cap C$ remains the same.

Example 1 The simplest function satisfying these principles is:

$$|\sigma_D(S) \cap C| - \frac{|\sigma_D(S)||C|}{|S|}$$

This function can intuitively be understood as the difference between the actual number of examples for which the rule classifies correct and the number expected if C were independent of D. ∎

Correctness, unsupervised learning When the training set contains only positive examples, or when no classes are defined by the user (i.e. unsupervised learning), application of the above correctness criterion will result in over-generalization. Any simple description that covers the entire training set will be assigned a high quality. Especially, the description *true*, covering all objects in the training set, will rate very good, because it is extremely simple, and assigns all examples to their correct (and only) class.

In [34], Michalski and Stepp suggest that a quality function should not only dependent on the simplicity of the description, but also on the *fit*: how close does the description approximate the set of examples. Thus, for a description, we should compute the ratio of $|\sigma_D(S)|$, the number of examples in the training set, covered by the description, to $|\sigma_D(\mathcal{U})|$, the total number of examples in the universe, covered by this description.

This ratio is exactly one if the description covers only examples in S, and none of the examples outside S. Thus, we search for the simplest description with the best fit.

Combining criteria The quality of a description depends on its validity f_v and correctness f_c. The quality function could also take some other factors into account, such as the cost of evaluating the description, or the cost of measuring attributes, used in the description.

We have to combine these criteria to compute the overall quality. We can either assign a *weight* to each criterion that denote its relative importance. The overall quality is the weighted sum of the qualities for these criteria. Finding the optimal weights is a form of fine-tuning the system. An alternative to this weighted sum is the *lexicographic evaluation functional* (LEF) [29]. The criteria $f_1, f_2, \ldots$ are ordered and the overall quality is computed as the LEF of these criteria:

$$\mathrm{LEF} = \langle (f_1, t_1), (f_2, t_2), \ldots \rangle$$

where $t_1, t_2, \ldots$ are tolerances. Given a set of descriptions, the LEF determines the most preferable ones in the following way: all descriptions are ordered, based on their score for the first criterion. Only the best, or within the range defined by the threshold t_1 from the best are retained. These are ordered according to the next criterion, and again only the best are retained. After evaluating the last criterion, the best description is returned.

Application of the quality function to the set of all constructible rules results in a landscape where peaks denote rules of high quality and basins denote rules of low quality.

2. Search algorithm

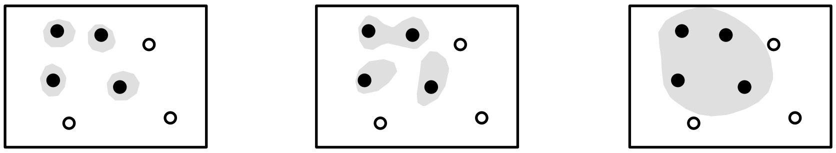
Different algorithms are used to traverse the search space. The basic idea is to start with an initial description D_1, and iteratively modify this description until its quality exceeds some user-defined threshold.

2.1 Initial description

There are two approaches, differing in the choice of the initial description, and in the operations that are applied to improve the quality of this description:

Bottom-up or data-driven. The initial description is simply the set[1], formed of all examples of the target class. This description is of course correct, but it is too complex. To reduce the complexity, the description is modified by repeated generalizations. This results in a more general rule, that classifies the examples correctly (or correct within a certain tolerance).

[1]Note that bottom-up strategies require that descriptions can cover an arbitrary subset of S, e.g. we cannot use set-descriptions.

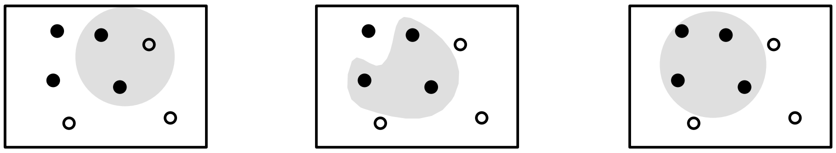


(a) (b) (c)

Figure 4.1: A bottom-up learning process.

An example of this technique is the DBLearn system, as we will discuss in Chapter 7. The bottom-up technique is illustrated in Figure 4.1. The learning task consists of finding a simple description that covers all positive examples (filled circles) and none of the negative examples (empty circles). The initial description (the shaded area) covers only the four positive examples, as denoted in *(a)*. The second description is more general, i.e. covers a larger area, as shown in *(b)*. The final description *(c)* covers all positive examples and none of the negative examples.

Top-down or model-driven, where we search the description space $\mathcal{D}$, to find the 'best' rule. We choose an initial description, e.g. the description covering the entire universe. We transform this description by applying a sequence of generalization and specialization operations, until its quality exceeds the threshold. Examples of a top-down approach are the META-DENDRAL system, the BACON system and the ID3-system, as we will discuss in Chapter 7.



(a) (b) (c)

Figure 4.2: A top-down learning process.

This top-down technique is illustrated in Figure 4.2. In *(a)*, the initial relationship covers two positive and one negative example. This gradually improves until in *(c)* a relationship is found that covers all positive and none of the negative examples.

2.2 Search graph
To the initial description D_1, we can apply some of the operations in $\mathcal{O}$, say $o_1, o_2, \ldots, o_n$ and application of any of these operations o_i results in a new description D_{2i}. To this description, we can again apply any of the operations, resulting in n new descriptions $D_{31}, \ldots, D_{3n}$. Thus, by repeatedly applying all operations, we can construct a *search graph*, where descriptions form the nodes, and operations form the arcs, as denoted in Figure 4.3. Mind that this graph need not be a tree, since different sequences of operations can result in the same description.

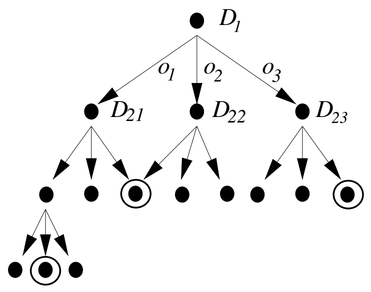


Figure 4.3: A search graph.

The aim is to find a description of sufficient quality, say one of the marked *goal nodes* in the figure, as fast as possible. The construction of a rule can thus be characterized as a search process, where operations are tried until some sequence of them is found that produces a rule of sufficient quality. The way in which these sequences are checked is called the *search strategy.* Basically, there are two kinds of search strategies, as discussed by Nilsson in [41]:

Irrevocable search In an irrevocable search strategy, an operation is selected and applied irrevocably without provision for reconsideration later. Only a single sequence of operations, i.e. a single path in the graph is evaluated.

Tentative search In a tentative search strategy, an applicable operation is selected and applied, but provision is made to return later to this point in the computation to apply some other operation.

We distinguish two different types of tentative strategies. In *backtracking*, a backtracking point is established when an operation is selected. Should subsequent computation encounter difficulty in producing a solution, the state of the computation reverts to the previous backtracking point, where another operation is applied instead, and the process continues. At any moment during computation, the system does not only store the current description, but also the path to the initial description, and for each node on this path it has to keep track of all operations that have been tried so far.

In a *graph search* strategy, the system stores the entire section of the search graph that has been explored so far. This allows for a greater flexibility, new descriptions can be generated at any place in this graph, but it requires more storage.

These search strategies can be *uninformed*—no heuristic information is used to guide the search: the operations are chosen arbitrarily. When the irrevocable strategy is used, there should either be only a single applicable operation, or the order in which operations are applied should be irrelevant. In tentative search, nodes can be examined in a *depth-first* manner, where a single path is examined, and backtracking is performed upon failure. If graph search is used, nodes can also be examined using *breadth-first* search, where nodes on equal distance from the starting node are examined.

Uninformed strategies are expensive, because they generate many descriptions, and for each of these descriptions, the quality has to be computed, which is a very expensive (database) operation. Thus, the performance of the system would benefit from a reduction of the number of generated nodes, by choosing the operations that are most likely to be on the shortest path to a goal node.

3. Heuristic search

The idea of heuristic search is to reduce the search effort by carefully selecting operations such that a description of sufficient quality is found as soon as possible. In other words, the system needs information that can be used to select the nodes on the shortest path to a goal node. Since this path is unknown in advance, the strategy requires information about the search domain, so-called *heuristic* or *domain* knowledge.

3.1 Hill climber

The hill climber strategy chooses always the operation that results in the greatest quality improvement. Hence, the system has to compute (or estimate, if computation is to expensive) the quality of all possible extensions. We define an *expectation function*, that returns the quality of the description that would be generated by applying a particular operation.

Definition 7. Expectation function The expectation function $F : \mathcal{D} \times \mathcal{O} \rightarrow [0, 1]$ could thus be defined as:

$$F(D, o) = f(o(D))$$

where f is the quality function, as defined above, and $o(D)$ is the description, resulting from application of operation o on D. We can generalize the expectation function, by looking n levels ahead, instead of a single level:

$$F^n(D, o) = \max_{o_i \in \mathcal{O}} F^{(n-1)}(o(D), o_i)$$

where $F^{(1)}$ is the simple expectation function F, as defined above. Thus, the expectation of operation o is the maximal quality of the descriptions that can be constructed within n steps, starting with the application of o. ∎

Instead of computing the quality of all possible extensions, one can also *estimate* the probability that applying a particular operation will result in a description of sufficient quality. For example, the ID3 system uses an information theoretic measurement, based on information about the training set, to estimated the likelihood that application of a particular operation will result in a simple, but correct, knowledge structure.

3.2 Limitations on the operations

Domain knowledge can be used to guide the search process. This information is specific to a particular domain, and has to be supplied by the user. Michalski distinguishes the following forms of domain knowledge (see [29]):

Irrelevant attributes Not all attributes in the examples are relevant. For example, first name is not considered to be a relevant attribute for medical diagnoses. For each attribute, the user could define whether it could be relevant for the classification or not, and thus reduce the number of constructible descriptions.

Sometimes, the relevance of an attribute depends on the value of other attributes. For example, the attribute *pregnant* is only meaningful if the person under consideration is female. So for particular classes, this attribute can be discarded.

This domain knowledge can be incorporated in the search algorithm by e.g. modifying the expectation function, such that it returns a high value for operations removing a condition on irrelevant attributes, and a low value for operations adding such a condition.

Interrelationships between attributes For some applications, there may exist known relationships between attributes that constrain their set of possible values. For example, the length of an object can be defined to be always greater or equal to its width, or the area of a rectangle always equals the product of its width and its length.

Hence, the quality of a description will not improve when adding a condition, that is already implied by conditions in this description. The expectation of such operations is therefore low.

3.3 User interaction

Another source of heuristic knowledge is the user, an expert in the domain. The system presents multiple descriptions, possibly together with their expected or estimated quality. The user selects one or more of these descriptions for further investigation by the system. Any information that is known to the user, but not coded as domain knowledge, can thus be incorporated in the search process.

3.4 Previously discovered rules and classes

Previously discovered rules, classes and their descriptions can be used to guide the search process. Classes can form a hierarchy, and can be used to construct new classes, or they can be refined during the search process. This is especially important when the set of examples is updated, and previously generated descriptions have to be refined, to be consistent with the new set of examples (see also incremental learning, Chapter 5).

4. Alternative search algorithms

So far we discussed the traditional symbolic computation approach: a systematic exploration of the search space, guided by heuristics. This approach has some shortcomings: in the absence of good heuristics, the strategy performs poorly, it takes unacceptable long times to find a description of sufficient quality. Another problem is that the search process is sometimes not able to escape a local maximum. To overcome these problems, other strategies have been proposed.

4.1 Genetic algorithms

Genetic Algorithms (GAs) originated from the studies of cellular automata, conducted by Holland *et al.* [18, 19]. A GA is a search procedure modelled on the mechanics of natural selection rather than a simulated reasoning process. The basic idea is to use a set of candidate descriptions—called a *population*, and to gradually improve the quality of this population by constructing new descriptions, assembled from parts of the best descriptions in the current population. These newly generated descriptions form the second *generation* G_1, and again, the best descriptions are recombined to form the next generation, until descriptions of sufficient quality are found or no further improvement occurs.

The candidate descriptions, also called *organisms*, are strings of symbols from a particular alphabet, generally the binary alphabet $\{0,1\}$. Each set-description is coded as a fixed length bit string. This string contains a substring for each attribute A_i, and in this substring, each position represents a value in the domain Dom_i for this attribute. If a value belongs to the set S_i, the corresponding bit-value is 1, otherwise it is 0.

Example 2 Assume a training set with two attributes, 'color' and 'shape', where the respective domains are {red, blue} and {rectangle,circle}. The set-description 'color $\in$ {red, blue} $\wedge$ shape $\in$ {circle}' can thus be coded as the string 11 01. ∎

Genetic algorithms do not use the generalization and specialization operations, as discussed earlier, but use two other operations instead: *crossover*, where part of a string is replaced by part of another string, i.e. two strings (the parents) are combined into a single new string. The other operation is *mutation*, a random change of a bit value in a string.

There are two approaches for learning classification rules, named after their university. These approaches differ in the representation of the rules, the crossover operations and the quality function. An overview of learning classification rules can be found in [22, 59]:

The Michigan approach developed by Holland. An organism represents a set-description. Consequently, all organisms are of the same length. The entire population together forms a classification rule, that is, a rule whose description consists of the disjunction of all descriptions, represented by the organisms.

The crossover operation consists of cutting both parent strings at the same position, and creating two new strings by combining the lower part of one parent string with the upper part of the other string, and vice versa.

The quality of an organism in this population dependents on its classification accuracy, i.e. the ratio of positive and negative examples covered, and its generality, i.e. the number of examples covered by the organism, relative to the total number of examples, covered by the entire population.

The Pittsburgh approach An organism is not a single description, as in the Michigan approach, but a disjunction of set-descriptions. So each organism represents a complex description, and the population consists of a set of competing descriptions. The length of each organism is thus a multiple of the length of a single set-description.

In the crossover operation, both parents are cut at the same place, modulo the set-description length. This guarantees that the children are (complex) descriptions as well.

The quality of an organism can be computed using the quality function, i.e. a function based on the correctness, classification accuracy and coverage.

Genetic Algorithms outperform traditional learning techniques, especially when the descriptions that have to learned are complex, i.e. include multiple conjunctions and disjunctions, or when no domain knowledge is available. However, GAs suffer from two important drawbacks: firstly, they outperform traditional techniques only when almost no domain knowledge is available. Although domain knowledge can be incorporated in the GA by either modifying the genetic operators, choosing a particular initial population, or modifying the quality function, the use of domain knowledge is limited, compared to traditional techniques.

A second drawback is the number of evaluations. A GA typically requires 500-1000 samples of the search space, i.e. about 10 generations of 50-100 organisms each, before it finds qualitative good solutions. Moreover, in a comparison of a genetic and a decision tree classifier, Quinlan concludes that an incremental genetic classifier *Boole* needed a much larger training set to achieve a similar accuracy as decision trees [50].

Hence, GAs are suited for learning tasks on small databases, where no domain knowledge, such as heuristics, is available. It may be possible to use GA in combination with traditional approaches, where the GA is used for a rough exploration of the search space, and traditional techniques are used to improve the result.

4.2 Simulated annealing

So far we used the *hill-climber* search strategy, i.e. at any moment during the search process, we apply the operation that results in the highest quality increase. However, this search process may result in a local maximum, that is, a non-optimal solution that has no neighbours of a higher quality. If we visualise the search space as a landscape, we could say that in an attempt to climb the highest mountain, we ended up on top of a small hill, where all routes – including the one to the mountain – lead down.

A solution for this problem is *simulated annealing*, a technique where we traverse the search space with a certain amount of arbitrariness. That is, we do not necessarily apply the operation that results in the highest quality increase, but use a selection process where each applicable operation has a probability of being chosen. This probability depends on:

Quality increase/decrease Operations that increase the quality are assigned a higher probability than operations that decrease the quality of the description. But the extend to which the quality influences this probability is controlled by another factor:

Temperature The randomness of the selection process is controlled by a global parameter *temperature*. If the temperature is high, then all operations are chosen with more or less equal probability. In particular, if the temperature is maximal, then all operations are assigned equal probability. On the other hand, if the temperature is low, operations that increase the quality are preferred to operations that decrease the quality. At the extreme, when the temperature is zero, the selection process reduces to a hill-climber strategy, where the best operation is always chosen.

The idea is to start the search process with a high temperature, such that the search process is able to escape from local maxima. The temperature is slowly decreased during the search process, and the process stabilizes in the global maximum.

The SAMIA system combines a bottom up strategy with simulated annealing (see [6]). Here, the next operation is randomly chosen, if application of this operation results in a positive change in quality Δf. Otherwise, if Δf is negative, the operation is applied with probability $e^{\Delta f/T}$, where T is the temperature.

Chapter 5
Problems

In previous chapters we discussed the discovery of descriptions in data. We implicitly assumed that these descriptions exist, e.g. in supervised learning, we assumed that there exists a deterministic classification rule. Although this may be a valid assumption for some artificial data sets, used in machine learning, it is certainly not correct when databases are used. Using a database as a training set causes several problems.

In this chapter, we discuss these problems and possible solutions. There are three categories of problems: first of all, the information supplied to the system is limited, so not all information, essential for the determination of the object's class, is available. Secondly, the available information can be corrupted or even partly be missing. Finally, the size and the dynamic behavior of databases introduce problems.

1. Limited information

In supervised learning, we attempt to find a relationship between the predicting variables and a class. We can envisage this as some process $\mathcal{T}$, where the input consists of the predicting variables, and the output is the predicted class, as depicted in Figure 5.1.

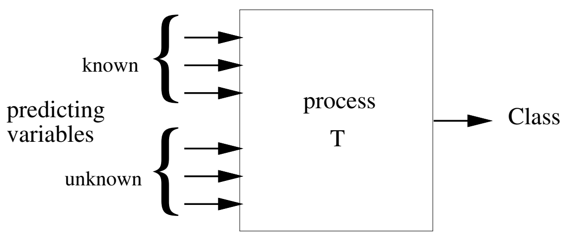


Figure 5.1: The environmental process $\mathcal{T}$.

1.1 Incomplete information

So far, we assumed that this process is deterministic, i.e. the class is uniquely determined by the predicting variables. However, often not all these variables are known: we can assume a set of unknown predicting variables, that are relevant for the classification, but not recorded in the database. Hence, it may not always be possible to construct a rule that classifies each example correctly in terms of the known predicting variables.

Basically, there are two approaches towards unknown variables: we can either restrict ourselves to *deterministic rules*, that is, we only construct rules when all variables relevant to the classification are known. This is the discovery of strong rules, as discussed in [43]. A disadvantage is that much valuable information that is hidden in the database cannot be found.

Alternatively, we can look for rules that do not necessarily classify examples correctly, but merely indicate the probability that an object, covered by the description, belongs to a class. These *probabilistic* rules provide very important information about relationships in the environment. For example, the (causal) relationship between smoking cigarettes and cancer is not a correct relationship, i.e. smoking is not a sufficient nor necessary condition for cancer, still, this relationship is considered very important.

If none of the known predicting variables is relevant for the classification, the variables and the class are *unrelated*, and even a probabilistic rules cannot be found. For example, finding a rule, predicting somebody's lastname in terms of medical data is impossible.

1.2 Sparse data

When the data mine system constructs a classification rule, it has to discover the class boundaries. The exact position of these boundaries can only be investigated if the database contains examples located just within or outside the class—the so-called *near misses* and *near hits*. In other words, the examples should represent a large variety of behavior, i.e. should be located throughout the universe $\mathcal{U}$ (the *entropy* should be high).

Unfortunately, facts in a database generally represent only a small subset of all possible behavior. Hence, class boundaries cannot be determined exactly, and will be either vague or incorrect. Previously unseen objects that are located outside the training set, are likely to be incorrectly classified.

As a solution, the system could interact with its environment, that is, the environment acts as an *oracle*: the system generates an interesting example, supplies it to the environment, which determines the corresponding class. An example of such a system is MARVIN [56]. Unfortunately, a data mine system is not able to manipulate its environment, since it uses an already existing database. A solution might be to search this database specifically for interesting examples, i.e. browse for additional information.

1.3 Samples

The training set S can be seen as a sample of the set of all constructible objects—the *Universe* $\mathcal{U}$. A data mine system searches for rules in this training set. A rule consists of a description D and a class C, and this description covers a subset $\sigma_D(S)$. This subset can again be seen as a sample.

Now assume that an arbitrary object in S belongs to class C with probability p, and that an object in set $\sigma_D(S)$ belongs to this class with probability p', and $p \neq p'$. The main question is: is the difference between p and p' statistically significant? In other words, did we locate a probabilistic relationship, or is the observed difference just due to chance? Data mine systems have to rely on statistical techniques to check the validity of discovered relationships.

1.4 Test set
The correctness of the discovered descriptions can be tested by splitting the database in two sets: a *training set*, used to construct descriptions, and a *test set*, used to test the accuracy of these descriptions. A data mine system is correct if the *actual probability* of each rule does not significantly differ from the *predicted probability* of this rule.

For adaptive systems – systems that adjust their rules over time – we can use a *sliding window*, i.e. compute the actual probability of a rule over the last n classifications (as used in [59]).

2. Data corruption
So far, the information supplied in the set of examples has been assumed to be entirely correct. Sadly, learning systems using real-world data are unlikely to find this assumption to be tenable. The examples may include attributes based on measurement or subjective judgements, both of which may give rise to errors in the value of attributes. Some of the examples may even be misclassified. Non-systematic errors of this kind in either the values of attributes or class information are usually referred to as *noise*.

2.1 Noise
Noise causes two problems: first of all when generating descriptions using a noisy training set, and secondly when noisy objects are classified using these descriptions.

Constructing descriptions If the system has to construct descriptions in a noisy environment, it should be prevented from constructing descriptions that attempt to cover corrupted examples as well. Therefore, it must be able to decide that a particular example – or an attribute in this example – is corrupted, and thus should be ignored.

In [48], Quinlan uses statistical techniques to decide if an attribute should be added to a classification tree. The basic idea is that a small amount of exceptional data is considered to be caused by noise, and can therefore be neglected.

Noise can also affect the class information of an example, which has a dramatic impact on the classification accuracy of the generated rules. Therefore, an attempt should be made to eliminate noise that affects the class information of the objects in the training set.

Classifying examples Once descriptions have been constructed from the training set, they can be used in classification rules to determine the class of previously unseen examples. In experiments with some systems, adding noise to the data resulted in *graceful degradation*: adding even substantial noise resulted in low levels of misclassification of unseen examples.

An interesting phenomenon, discussed in [47, 48], is that rules, learned from a corrupted training set, perform superior on classifying noisy data when compared to rules that are learned on the same, but noise free training set. In [47], Quinlan concludes that "it is not worthwhile expending effort to eliminate noise from the attribute values of objects in the training set if there is going to be a significant amount of noise when the induced classification rule is used in practice".

2.2 Missing attribute values
Another problem that also arises in using a database, is that attribute values may be missing. However, we would like a data mine system to construct descriptions even when some of the

attribute values are missing, and to be able to apply classification rules, containing these descriptions, to data where some attribute values are missing as well.

Constructing descriptions Examples with missing attributes can be simply discarded, or an attempt can be made to replace the missing value with some 'most likely' value. In [48], Quinlan suggests to construct rules that predict the value of a missing attribute, based on the value of other attributes in the example, and the class information. These rules are then used to 'fill in' the missing attribute values, the resulting set is used to construct the descriptions.

Another approach is to treat unknown values as a separate value, i.e. add the value 'unknown' to the domain of each attribute, and use this value in the descriptions.

Classifying examples The constructed rules can be used to classify unseen examples from which attribute values are missing. When a rule contains conditions on some of these attributes, it cannot be applied.

A technique to overcome this deficiency is to compute the *probability* that a certain rule applies. This probability is the product of the probabilities that each missing attribute value is the value, required in the condition of the rule. The probability that an attribute has a particular value can be estimated by analysing the relative frequency of values for this attribute in examples in the training set.

So, given a set of classification rules, we compute for each rule the probability that the example is consistent with this rule. Next, we sum the probabilities for each class, the result of the classification is the class with the highest value. Straightforward though it may be, this technique has been found to give a very graceful degradation as the number of unknown values increases.

3. Databases

As we stated in Chapter 3, data mining is machine learning using a large database as a training set. These databases differ in some aspects from the training sets used in machine learning.

3.1 Size

Machine learning systems use small training sets, e.g. thousand objects is considered to be large. Databases are generally very large, both in the amount of information per object, as in the number of objects in the database:

Information per object Most databases contain many attributes. For example, in Chapter 7, we describe the RADIX/RX project, where the ARAMIS (American Rheumatism Association Medical Information System) database is used. This database contains information about clinical visits of patients. Each visit is stored in a record with 400 attributes, and for each patient 50 to 100 visits are stored. So for each patient, in total 20 to 40,000 attributes are recorded.

First of all, we should notice that it is an advantage that much information is provided, because the more information (per object) is provided, the more likely it is that relationships actually exist. However, more information also increases the number of constructible descriptions, i.e. the size of the description space.

The number of descriptions depends on the size of the domains, e.g. using set-descriptions, the number of constructible descriptions is roughly 2^l, where l is the sum of the domain-sizes of the attributes. Clearly, for any realistic database (where $l \gg 1000$), the description space becomes very large. We can use constraints and heuristics to search this space for near optimal solutions, as we discussed in Chapter 4.

Number of objects During the search process, the quality of each generated description has to be verified. As discussed before, we need statistical tests to check if the description actually describes some regularity in the data. This test needs information about the set of examples, such as the number of examples covered by the description, or the distribution of values in this set. In other words, computing the quality of a rule requires database access.

The databases used in data mining are large (typically $> 100,000$ objects), hence querying the entire database is expensive. To overcome this problem, we can use the following techniques:

1. *Multiple descriptions* can be constructed in a single iteration of the search process, and their quality can be computed simultaneously, that is, by a single (but complex) database access. We run down the entire database and for each object update the information for each description.

2. *Windows.* We can compute the quality of a description using a representative sample of the database, called a *window*. A small subset – containing a few thousand objects – is used to construct descriptions. The best descriptions are then tested on the entire database, i.e. the database is browsed for additional proof.

   As long as the actual probability of the rules is not equal to the predicted probability, we add some of the incorrectly classified examples to the window, and modify the rules using this window. This modification process is called *incremental learning*, as we will discuss in the next section.

3.2 Updates

Databases are frequently updated: information is added, modified or removed. Any knowledge that was previously extracted from this database can therefore become inconsistent. It is obvious that a learning system should *adapt* to such changes. Moreover, the reliability of rules increases when the sample size increases, so if rules are learned on a small database, and this database is extended over time, it makes sense to keep rules consistent.

It is essential to keep rules consistent with the most *recent* information, because characteristics of examples can change over time, due to trends and processes in the environment. For this reason, the most recent information should be valuated higher then older information, i.e. we should use some *learning bias*, assigning higher weights to recent experiences in the learning process.

Cognitive systems adjust their rules when too many incorrect predictions are made (see Holland *et al.* [20]). This could also trigger rule adjustment in data mine systems: when the actual probability of a rule is out of pace with its predicted probability over a certain period, the rule is adjusted.

We can either reconstruct the rule from scratch, but it would be more convenient to allow some kind of *incremental learning*, where previously generated knowledge is used in the reconstruction process. There are two forms of incremental learning: *learning with full memory*, where the system remembers all examples that were seen so far. By this method, as opposed to *learning with partial memory*, new rules are guaranteed to be correct with respect to all (old and new) training examples. In Chapter 7, we describe some systems that support incremental learning.

Chapter 6
Knowledge representation

In previous chapters we represented knowledge using (relational algebra) selection conditions, either as the condition in classification rules (supervised learning) or to describe the entire database (unsupervised learning). We will briefly review some other representations, starting with logic formulae in propositional form.

As we will see, propositional representations suffer from some disadvantages, which has been a reason to move to a more powerful representation, such as First Order Logic (FOL). We shortly outline the advantages of FOL, and discuss some representations with equivalent expressive power that offer a better structuring of the knowledge, such as semantic networks and frames. For a more extensive discussion of these representations, the reader is referred to [8, 11, 54].

The last part discusses neural networks, a non-symbolic representation, together with a highly parallel learning algorithm. We compare this representation with symbolic representations.

1. Propositional-like representations

Propositional representations use a logic formulae, consisting of attribute value conditions. For example, '(color = red $\vee$ color = green) $\wedge$ shape = circle' is a formula in Conjunctive Normal Form (CNF), i.e. a conjunction of clauses, where clauses are disjunctions of attribute value conditions. In previous chapters, we represented knowledge as set-descriptions, which is actually a formula in CNF. For example, the above formula could be represented as the set-description 'color $\in$ {red, green} $\wedge$ shape $\in$ {circle}'.

An alternative is the Disjunctive Normal Form (DNF), a disjunction of terms, where terms are conjunctions of attribute value conditions. However, as shown in [37], the CNF performs surprisingly well, that is, the generated descriptions are smaller than the DNF representations for the same learning tasks.

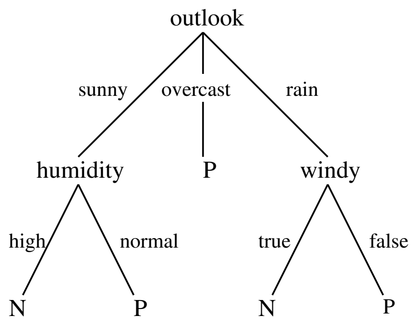
Actually, both representations are not really propositional, because they involve a variable—the object itself. For example, when the above formulae is used in a production rule, it is a condition on an object X, stating '$P(X)$ = (X.color = red $\vee$ X.color = green) $\wedge$ X.shape = circle'. For this reason, we refer to these representations as 'propositional-like'.

1.1 Decision trees

A decision tree is a simple knowledge representation that has successfully been used in supervised machine learning systems, e.g. in Quinlan's ID3 system, as we will discuss in Chapter 7.

A decision tree classifies examples to a finite number of classes. Nodes in the tree are labeled with attribute names, the edges are labeled with possible values for this attribute, and the leaves are labeled with the different classes. An object is classified by following a path down the tree, by taking the edges, corresponding to the values of the attributes in the object.

Example 1 An example, taken from [48], consists of a small training set, storing objects that describe the weather at a particular moment. Objects contain information on the outlook, which is either sunny, overcast or rain, the humidity, which is either high or normal, and some other properties. Some objects are positive examples of a class P, other are negative examples. The classification task consists of constructing a simple tree that classifies all objects in the training set correctly. The following tree could be constructed:



∎

1.2 Production rules

A disadvantage of decision trees is that they tend to grow very large for realistic applications, and are thus difficult to interpret by humans. Hence, there has been some research in transforming decision trees into other representations. In [49], Quinlan describes a technique to generate propositional-like production rules from decision trees. This transformation is useful because:

1. production rules are widely used for representing knowledge, e.g. in expert systems,

2. they can easily be interpreted by human experts, because of their extreme modularity, i.e. a single rule can be understood without reference to other rules,

3. the classification accuracy of a decision tree can be improved by transforming it into a set of production rules, thereby eliminating tests that are attributable to peculiarities in the training set.

Here we consider simple, propositional-like production rules, i.e. if-then rules, where the conditional part consists of a propositional expression, e.g. an expression in CNF, or simply a term. The consequence is the class. Michalski's AQ15 system (see Chapter 7) uses production rules as a knowledge representation.

EXAMPLE 2 Each path in a decision tree corresponds to a term: a conjunction of conditions on attributes. The above tree is equivalent to the following set of production rules:

if outlook = sunny and humidity = high then class = N
if outlook = rain and windy = true then class = N
default class = P ∎

1.3 Decision lists

Another propositional-like representation, proposed in [55], is the decision list representation. This representation strictly generalizes both decision trees, DNF and CNF representations, i.e. any knowledge structure in these representations can be transformed to a decision list. A decision list is a list of pairs

$$(\phi_1, C_1), (\phi_2, C_2), \ldots, (\phi_r, C_r)$$

where each ϕ_i is an elementary description, and each C_i is a class, and the last description ϕ_r is the constant true. The class of an object o is C_j if j is the least index of a description ϕ_j that covers object o. Such an index always exists, since the last term is always true, i.e. the default class.

We may think of a decision list as an extended 'if ϕ_1 then C_1 elseif ϕ_2 ... else C_r' rule. Or we may think of a decision list as defining a class by giving the general pattern with exceptions. The exceptions correspond to the early terms in the decision list, whereas the more general patterns correspond to the later terms. Decision lists are used in the CN2 system, which we will discuss in Chapter 7.

1.4 Ripple-down rule sets

From a knowledge representation point of view, one of the main features of rules is that they tend to have exceptions. More realistic rules are of the form 'if ϕ_i then C_i unless ϕ_j'. We can represent this in a decision list by placing an exception rule 'if ϕ_j then C_j' into the list before the actual rule 'if ϕ_i then C_i'. However, the main disadvantage of using decision lists is that this exception is *global*, i.e. any object for which ϕ_j is true will be assigned to class C_j, whereas we wanted the exception to be *local* to the rule 'if ϕ_i then C_i'.

This lack of locality makes decision lists difficult to understand, since the rules in the list are only meaningful in the context given by all the preceding rules. Hence, there is a need for representing exceptions in a more localized manner, e.g. by using ripple-down rule sets [13]. These rules consist of conditions and exceptions to these conditions that are local to the rule. In fact, these rules are nested if-then statements, e.g.

if ϕ_i then
    if ϕ_j then C_j
        else C_i
    else C_j

is a ripple-down rule set of depth 2. Note that by giving all the exceptions locally, we have eliminated the need for a global ordering of the rules, as in decision lists. In [24], an efficient algorithm for the construction of ripple-down rule sets is presented.

2. First order logic

The simple propositional-like representation has been successfully used in many machine learning systems. Along with the successes of this technology, the following limitations are becoming apparent [39]:

Restricted representation Propositional-like representations strongly limit the form of relationships and patterns that can be represented. Patterns that are defined in terms of relationships among objects or attributes cannot be represented.

For example, consider a class consisting of all persons with 'identical first and lastname'. Using a propositional-like representation, we would end up with an *over-fitted* pattern, enumerating all persons in the database with identical first and last name. Using a more powerful representation allows us to state that any person where 'firstname = lastname' belongs to the class, thereby capturing the true nature of this class.

Inability to make use of domain knowledge It is difficult to incorporate domain knowledge in the learning process. A commonly used technique is to regard domain knowledge as constraints on the descriptions, generated by the system. Since domain knowledge is rarely complete and consistent, this constraint is generally believed to be over-restrictive.

Strong bias to vocabulary Present inductive systems construct descriptions within the limits of a fixed vocabulary of propositional attributes. For many applications, it might be useful to increase the set of patterns that can be found and to improve the comprehensibility of the representation by inventing auxiliary predicates.

The above shortcomings can be overcome by moving towards a more powerful representation. A growing number of machine learning systems employ some kind of First Order Logic (FOL) to represent learned knowledge (for an overview see Muggleton [39]). In this area, called *Inductive Logic Programming* (ILP) the aim is to construct a FOL-program that, together with the domain knowledge, has the training set as its logical consequence. If we regard the program as a logical theory, then the training set is a model of this theory. An example of an ILP system is Quinlan's FOIL system (see [51]).

There are some extensions to FOL that may prove to be very useful for representing knowledge in a data mine system. *Constraint logic programming* languages (see Jaffar and Lassez in [21]) allow complex constraints between numerical variables. These constraints can be very useful for representing numerical relationships, as we discuss in Chapter 8.

The language LIFE (Logic Inheritance Functions Equations), developed by Aït-Kaci (see [1, 2]), is a synthesis of three different programming paradigms: logic programming, functional programming and object-based programming. In LIFE one can define hierarchies of values, that are useful for representing domain knowledge. Also, LIFE allows functions to be used in a logic language.

2.1 Expressive power versus computational complexity

Using a powerful representation such as FOL allows us to find simple descriptions for classes, that would have a very complex description (or could not even be described) in a less powerful representation. So on one hand, the computational complexity decreases, since the system can find a simple description for the class. Thus enlarging the set of possible descriptions may make learning easier, rather than harder. The reason is that it may be easier for the learning algorithm to produce a nearly correct answer from a rich set of alternatives than from a small set of possibilities, as discussed in Section 2 in Chapter 3, and in [55].

On the other hand, when using a more expressive formalism, more different descriptions can be constructed, i.e. the description space extends, and finding the best description becomes harder. A solution is to search for particular descriptions only. For example, when using a FOL representation we could only consider single clause programs. Actually, this is the same problem as in database query languages, where, by restricting queries to a small subset of FOL (say SQL), queries remains computational. An alternative solution is to develop good heuristics, needed to guide the search in the increased search space.

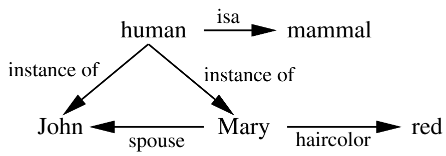
3. Structured representations

The representation of knowledge can be improved by using more expressive formalisms, as we stated above, but systems can also benefit from using a more comprehensible *structured* representation. We discuss two such representations, semantic nets and frames. Although they are not more powerful than FOL (they can easily be expressed as a FOL program), they provide a more comprehensible representation, e.g. by explicitly stating subtype relationships among objects, or allowing for an exception based representation.

3.1 Semantic nets

A semantic network is a graph, where the nodes denote concepts, or meanings, and the arcs denote relationships between these concepts (see Minsky [35]).

Example 3 The semantic network below states that John is the spouse of Mary, and Mary has red hair. Furthermore, John and Mary are both instances of the sort humans, which are again mammals.



■

As we can see from this example, there are actually two forms of relationships: relationships between concepts (spouse, and hair color) and relationships between concepts and classes (instance_of, and isa). Because storing both relationships in the same network is confusing, the latter relationships – *subtype relationships* – are sometimes represented in a separate network.

A semantic network can easily be mapped to FOL, where each arc is a binary predicate, with its nodes as terms. This representation can also be used for the subtype relations, but a better representation might be:

spouse(mary, john).
haircolor(mary, red).
human(john).
human(mary).
$\forall X.human(X) \rightarrow mammal(X).$

The main advantage of semantic nets is that all information related to a particular object can easily be found by following the links from this object. When semantic nets are used for data mining, each example is a semantic net. Operations on the examples consist of graph-manipulations, in order to find patterns, i.e. subgraphs, shared by all examples of the same class.

3.2 Frames and schemata

A frame, also called schema, is a structured object, consisting of a name, and named attributes—called *slots*, filled with values for particular instances.

EXAMPLE 4 The information about Mary in the previous example can be stored in a frame with four slots:

framename	person
slot 1	isa: mammal
slot 2	name: Mary
slot 3	spouse: John
slot 4	haircolor: red

∎

As can be seen from the example, we can incorporate subtype information in frames as well, by using 'isa' slots. An isa slot refers to another frame, in this case the mammal frame. This frame stores information about mammals, e.g. that all mammals are viviparous. Slots and their values, defined at higher levels in the hierarchy are inherited by the lower levels. However, these values may eventually be overridden, i.e. at a high level, default values are defined for slots (all mammals are viviparous), but these can be overridden for exceptional cases at lower levels (a platypus is a mammal, but not viviparous).

Although frames can be represented in FOL, they provide a better insight in the structure of knowledge than a set of – logically equivalent but unstructured – first order predicates. The EURISKO system uses frames for the representation of the acquired knowledge [27].

4. Neural networks

Artificial neural networks, also known as connectionist models, are densely interconnected networks of simple computational elements (for an introduction, see [28, 40]). In Figure 6.1 such an element, called *neuron* is shown. The input consists of N values $x_0, x_1, \ldots, x_{N-1}$ and a single output y, all having continuous values in a particular domain, e.g. $[0, 1]$.

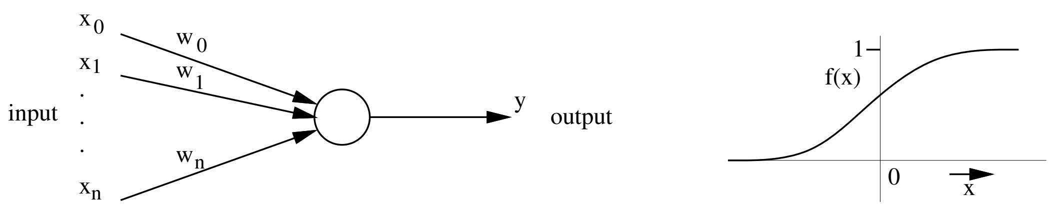


Figure 6.1: A neuron and a sigmoid function.

The neuron computes the weighted sum of its inputs, subtracts a threshold θ, and passes

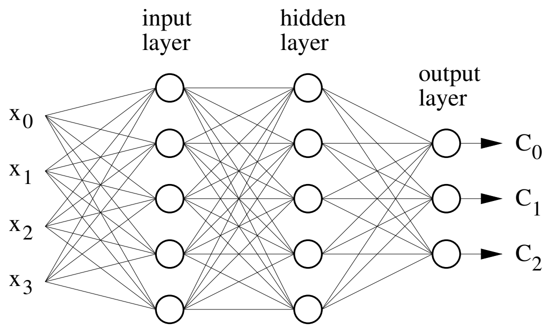


Figure 6.2: A multi-layer perceptron.

the result to a non-linear function f, e.g. the sigmoid, shown in Figure 6.1. Hence, each neuron in a network computes a function

$$y = f\left(\sum_{i=0}^{N-1} w_i x_i - \theta\right) \text{ where } f(x) = \frac{1}{1+e^{-x}}$$

and w_i are the weights.

Neural networks are constructed by connecting the output of a neuron to the input of one or more other neurons, and by assigning unconnected inputs of some nodes as the input of the network, and particular nodes as the output nodes of the network. There exist many different network topologies, such as the Kohonen or Hopfield network. Here, we will discuss a network called the *multi-layer perceptron*, as depicted in Figure 6.2. This network consists of multiple layers, such that the output of each neuron in a layer is connected to the input of nodes in the next layer. The inputs of the first layer—the *input layer*, form the input of the network, where the outputs of the highest layer—the *output layer*, form the output of the network.

4.1 Representation

A multi-layer perceptron is especially useful for implementing a classification function that assigns an object, denoted by an input vector $[x_0, x_1, \dots, x_n]$, to one or more classes $C_1, C_2, \dots,$ C_m. If the object belongs to class C_i then the network output C_i has a high value (typically 1) and a low value (typically 0) if it is a negative example of class C_i.

So by selecting appropriate weights and thresholds for all nodes, the network can represent a wide range of classification functions. Choosing correct weights can be done by supervised learning, where the network is provided with examples. An example is an input vector together with the desired output vector, that is, a vector where C_i has value 1 if the example belongs to class C_i and 0 otherwise.

4.2 Learning

In learning algorithms, examples are provided one at a time. For each of these examples, the actual output vector is computed and compared to the desired output. Then weights and thresholds are adjusted, where weights that contributed to a correct output remain unchanged, and weights that contributed to an incorrect output are decreased if the actual output value is higher than the desired value, and increased if the actual value is lower than the desired value. The algorithm terminates when all weights stabilize.

An algorithm that is often used is the *back-propagation* algorithm, discussed in [17, 28]. This algorithm uses an iterative method to propagate error terms (i.e. the difference between the actual and the desired output) required to adapt weights back from nodes in the output layer to nodes in lower layers.

4.3 Neural nets versus symbolic learning methods

The error rates of neural nets are equivalent to error rates of rules produced by symbolic learning methods, although neural nets perform slightly better when dealing with noisy data.

However, there are some disadvantages in using neural nets for data mining. First of all, learning processes in neural nets are very slow, compared to symbolic learning systems. In benchmarks described by Quinlan in [45], the ID3 system outperforms back propagation neural nets by a factor 500 to $100,000$.

Another disadvantage is that knowledge, generated by neural nets, is not explicitly represented in the form of rules or conceptual patterns, but implicitly in the network itself, as a vast number of weights. One of the objects of data mining is to generate knowledge in a form suitable for verification or interpretation by humans. There has been some research on transforming this knowledge to a format better suited for human reading, as in [23, 57], but this mainly concerns single layer networks, that model simple, linear functions.

As with genetic algorithms, it is difficult to incorporate any domain knowledge or user interaction in the learning process. Hence, neural nets perform best in areas where no additional information is available, which is generally not the case with data mining.

Chapter 7
Overview of data mine systems

In Chapters 3 and 4, we presented a framework for data mine systems. As an illustration of this framework, and to make the reader familiar with techniques used in this area, we will discuss a few data mine systems. Several aspects of these systems will be discussed: the representation for the training examples and for the generated knowledge, the operations on knowledge structures, and the quality function. We also discuss the search strategy, and the heuristics, used to guide the search.

We do not intend to provide a complete overview of the research, neither to compare systems. Moreover, one should keep in mind that most of these systems are actually machine learning systems, i.e. they assume small sets of relatively noise-free training examples, and attempt to construct deterministic rules. All systems that we discuss are designed for supervised learning.

1. ID3

The system that had the greatest impact on machine learning research in the last years is ID3, developed in the first half of the eighties by Quinlan (see [46, 48]). ID3 stands for *Induction of Decision Trees*, and is a supervised learning system that constructs decision trees from a set of examples. These examples are tuples, where the domain of each attribute in these tuples is limited to a small number of values, either symbolic or numerical.

Early versions of the ID3 system generated descriptions for two classes, i.e. positive and negative examples, but this restriction has been removed in later systems. Classes have to be mutually disjunct: there are no inconsistent examples. ID3 generates descriptions that classify each object in the training set correctly, i.e. it generates strong classification rules.

1.1 Search space

Knowledge is represented as a decision tree, as we described in Section 1.1 in Chapter 6. The search space for a particular problem consists of all trees that can be constructed with attributes and values in the test set. To traverse this space, a transformation operation is defined: extend the tree by replacing a leaf with a new subtree (of depth one).

Among all trees in the search space, the system needs to find the 'best' tree. The quality function, i.e. the quality of a tree, depends on both the classification accuracy, and the size of the tree. Trees that classify all objects in the test set correctly, and are simple as well, are preferred. The rationale behind the latter is that a decision tree captures some meaningful relationship between an object's class and the values of its attributes. Given a choice between two decision trees, each of which is correct over the training set, it seems sensible to prefer the simpler one on grounds that it is more likely to capture structure inherent to the problem (i.e. Ockham's razor). The simpler would therefore be expected to classify more 'unseen' objects correctly.

1.2 Search algorithm
The ID3 system uses a top-down irrevocable strategy that searches only part of the search space, guaranteeing that a simple – but not necessarily simplest – tree is found. A tree is constructed as follows:

1. an attribute is selected as the root of the tree, and branches are made for all different values this attribute can have;
2. the tree is used to classify the training set. If all examples at a particular leaf belong to the same class, this leaf is labeled with this class. If all leaves are labeled with a class, the algorithm terminates;
3. otherwise, the node is labeled with an attribute that does not occur on the path to the root, and branches are created for all possible values. The algorithm continues with step 2.

The above algorithm always creates a tree that classifies all data objects in the set of examples correct, but this tree is not necessarily simple. A simple tree can be generated by a suitable selection of attributes. In ID3, an *information-based heuristic* is used to select these attributes. The heuristic selects the attribute providing the highest information gain, i.e. the attribute which minimizes the information needed in the resulting subtrees to classify the elements.

We define two classes: a class P containing all positive examples, and a class N for the negative examples. Let the set of examples S contain p elements of class P and n elements of class N. In information theory, a measure is defined for the amount of information, needed to decide if an arbitrary example in S belongs to P or N:

$$I(p,n) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$$

Note that $I(p,n)$ depends on p and n only, and that $I(p,n) = 0$ for $p = 0$ or $n = 0$, and $I(p,n) > 0$ otherwise. Assume that using attribute A as the root in the tree will partition S in sets $\{S_1, S_2, \ldots, S_v\}$. If S_i contains p_i examples of P and n_i examples of N, the information, needed to decide if an element in S_i belongs to P or N is $I(p_i, n_i)$. So the information – needed to classify an element of S using a tree with attribute A as root – is the weighted average of the information, needed to classify objects in all subtrees S_i:

$$E(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I(p_i, n_i)$$

The attribute A is selected such that the *information gain* is maximal, that is, $E(A)$ is minimal. However, a known problem with this criterion is that it tends to favor attributes with many values. Different solutions to this problem are discussed in [48].

1.3 Numerical attributes

A condition on an attribute, i.e. an internal node of the decision tree, is a test on the value of an attribute, with branches for all possible values. Although this is convenient for symbolic attributes, it would be useful if one could test on ranges of numerical attributes. An extension of the ID3 algorithm, called C4.5 (see [52]) allows tests on the inequality of numerical attributes, such as $A_i \leq N$ and $A_i > N$, with two possible outcomes (branches).

The information gain of such a test is computed as follows: the examples are first sorted on the values of the attribute being considered. There are only a finite number of these attributes, say $\{v_1, v_2, \ldots, v_m\}$. In this set, there are $m-1$ possible splits on the attribute, all of which are examined. It may seem expensive to examine all $m-1$ splits, but, when the examples have been sorted as above, this can be carried out in one pass, updating the class distributions to the left and right of the threshold on the fly. For each possible threshold, the information gain is computed, and used in the process of selecting the next test, as described above.

1.4 Grouping attribute values

Another test implemented in the C4.5 system is a test whether the value of an attribute belongs to a particular set of values, such as $A_i \in \{v_1, v_2, \ldots, v_n\}$. The node is labeled with the attribute, and the branches are labeled with sets of values, rather than singleton values. For some situations, this can reduce the complexity of the resulting tree, e.g. when values are related in some sense.

However, for m different values, there are $2^{m-1}-1$ different binary partitions, which excludes the possibility of an exhaustive search for the best partitioning. C4.5 uses a irrevocable bottom up search, based on iteratively merging of groups. The initial groups are just the individual values of the attribute under consideration and, at each cycle, C4.5 evaluates the consequences of merging every pair of groups. (Any division of values into groups is reflected in the split of examples by that attribute, and so in the corresponding split information and gain.) The process continues until just two value groups remain, or until the gain cannot be further improved by merging.

1.5 Noise

Noise can affect both attribute values and class information (see [47]). A corrupted set of examples causes two problems:

1. It is no longer possible to generate a tree that classifies all examples correctly, i.e. the consistency condition is violated. When classes overlap, at step 3 in the algorithm there will occur a set S_i of objects that do not belong to the same class, and there are no attributes that can be used to further branch at this leaf.

   The solution consists of labeling this leaf with the class that dominates S_i, or with all classes in S_i, together with their probability.

2. Corrupted examples can cause the tree to grow to accommodate these examples. For example, generally the algorithm will not branch on an attribute A which hardly reduces the information needed after branching. However, if values for this attribute are corrupted for some examples, branching on A might give an apparent information gain, even though values in A are random, and form no indication for the classification.

   To overcome this problem, we can require the information gain to exceed some threshold. However, experiments suggest that a threshold which is high enough to rule out irrelevant attributes, will also rule out relevant ones.

An alternative is the χ^2 test: if attribute A is *irrelevant* to the class of an object in S, i.e. attribute A and the class are statistically independent, the expected value p'_i of p_i (the number of elements of class P in S_i), would be:

$$p' = p\frac{p_i + n_i}{p + n} = p\frac{|S_i|}{|S|}$$

If n'_i is the corresponding expected value of n_i, the statistic

$$\sum_{i=1}^{v} \frac{(p_i - p'_i)^2}{p'_i} + \frac{(n_i - n'_i)^2}{n'_i}$$

is approximately χ^2 with $v - 1$ degrees of freedom. The tree construction procedure is modified such that only attributes whose irrelevance can be rejected with a very high confidence level will be used. Application of this test results in a pruned tree.

1.6 Missing attribute values

Another problem concerns missing attribute values, resulting in problems in the construction of a decision tree, and in problems arising when we attempt to classify an object with missing values.

1. For the construction of a tree, several methods have been proposed to deal with missing attribute values. We can either treat missing values as the value most appearing in their class, or simply discard examples with missing values, or we can treat missing values as a special value *unknown*. However, the latter technique increases the expected information gain for an attribute if some values are unknown, which is not a desirable property. Therefore, a better technique is proposed, where the distribution of unknown values is assumed to be in proportion to the relative frequency of these values in S.

2. During classification, objects with missing attribute values cannot be classified, if branching on one of these attributes is required. All branches for this attribute are then explored, and for each resulting path, ID3 estimates the probability that this is the correct choice of values for the attributes. This probability is the product of the relative frequencies of the chosen values for the unknown attributes. Then, for all classes, these probabilities are summed , and the result of the classification is the class with the highest probability. This procedure offers a very graceful degradation when the incidence of unknown values increases.

1.7 Windows

At each cycle in the algorithm, the training set has to be queried to determine the information gain for attributes. (Note that we only have to query the subset of examples that have not yet been classified.) Instead of the entire training set, a randomly chosen subset—called *window* can be used. Using this window, a tree is generated, and all examples in the training set are classified using this tree. As long as not all examples are correctly classified, some of the incorrectly classified examples are added to the window and the process continues: a new tree is generated for these examples, however, this tree is generated from scratch (see below).

Experimental results show that correct decision trees are found within a few iterations, and that this method is usually faster than forming the tree using the entire database.

1.8 Incremental learning

The ID3 algorithm performs well when the entire training set is supplied at once. However, if the examples are supplied one at a time, the ID3 algorithm can still be used, but it would construct a new decision tree from scratch, every time a new example is observed. For such *serial learning tasks*, one would prefer an incremental algorithm, on the assumption that it is more efficient to revise an existing tree than it is to generate a new tree every time.

An adaptation of the ID3 algorithm, called ID5R (see [61]), does not create a new tree for each new example, but instead restructures the tree to make it consistent with the current and all previous examples. These previous examples are retained, but not reprocessed, they are used for restructuring purposes only. ID5R constructs exactly the same tree as the ID3 system on the same data set.

Even for non-serial training tasks, incremental algorithms can perform superior. Experiments show that selecting training instances one at a time, based on inconsistency with the current tree, leads to a smaller tree than selecting many instances at a time, as done in ID3. The latter will find the identical tree if the initial window contains just one example, and the window is grown by one instance at a time. But as we saw above, this approach is prohibitively expensive.

1.9 Conclusion

The system performs very well on a wide range of application domains, such as medical domains, artificial domains, and e.g. the analysis of chess end games. The classification accuracy is high. However, the system does not make any use of domain knowledge. Furthermore, trees are not easy to understand, however, they can be transformed to decision rules, as described in [49].

2. AQ15

Michalski's AQ15 system (see [32, 33]) is an inductive learning system that generates decision rules, where the conditional part is a logical formulae. A special feature of this system is *constructive induction*, i.e. the use of domain knowledge to generate new attributes, that are not present in the input data.

As ID3, and many other machine learning systems, AQ15 is primarily designed for the construction of strong rules, i.e. for each class, a decision rule is produced that covers all positive examples and no negative ones. The system handles incomplete and inconsistent examples by pre and post processing. Number and size of the discovered rules are drastically reduced by applying a post processing technique, called *rule truncating*, that does not affect the classification accuracy.

Examples in the training set are vectors of attribute values. Attributes can be of three types: nominal, linear or (hierarchically) structured.

2.1 Search space

Rules are represented in the VL_1 (Variable-valued Logic system 1) notation, a multiple-valued logic attributional calculus with typed variables. A *selector* is an attribute value condition that relates an attribute to a value or to a disjunction of values, e.g. 'color = red $\vee$ green'. A conjunction of selectors is called a *complex* (i.e. a set-description). The conditional part of a decision rule is formed of a disjunction of complexes, called a *cover*.

When building a decision rule, AQ performs a heuristic search through the space of all logical expressions to determine those that account for all positive and no negative examples.

Because there are (in machine learning problems) usually many such strong rules, the goal of AQ is to find the most preferred one. This *preference criterion* is defined by the user, to reflect the needs of the application domain.

The operations on the knowledge structure are the addition of a complex to the cover of a rule, and the *intersection* of a set of complexes with a set of selectors. The intersection of two sets A and B is the set of all combinations of conjunctions of elements from both sets, i.e. $\{x \wedge y \mid x \in A, y \in B\}$.

2.2 Search algorithm

The system generates a decision rule for each class in turn, using a top down irrevocable search algorithm (actually a beam search: multiple hill climbers in parallel). At each step in the algorithm, the best complex is added to the cover. Each step starts with focussing attention on one selected positive example—called the *seed*. The algorithm generates a set of all complexes (a *star*) that cover the seed and do not cover any negative examples, and then selects the best complex from the star according to the user defined criteria. This complex is added to the cover. The basic covering algorithm is:

While partial cover does not cover all positive examples
do

1. select a seed, i.e. an uncovered positive example,
2. generate a star, i.e. determine maximally general complexes covering the seed and no negative examples (see below),
3. select the best complex, as defined by quality function,
4. add the complex to the cover.

The algorithm starts with an initial cover that is either empty, previously learned, or supplied by the user. Step 2, generating a star, consists of:

While partial star covers negative examples
do

1. select a covered negative example,
2. generate a *partial star*: all maximal general selectors that cover the seed and exclude the negative example,
3. generate a new partial star by intersecting the current partial star with the partial star constructed so far,
4. trim the partial star, i.e. retain only the *maxstar* best complexes.

If the star generating procedure were to work exhaustively, the search space for covers might grow very rapidly. Therefor, the user can define a parameter *maxstar*, that controls how many disjoint complexes may be kept in a partial star. Only the best complexes are retained, according to user specified criteria such as 'maximize the number of positive examples covered and negative examples excluded', or 'minimize the number of selectors'.

So, for a given partial star, the above procedure computes the quality of the complexes in the intersection, retains *maxstar* of these (the beam width), and intersects these again, until none of the negative complexes is covered.

2.3 Inconsistent data and noise

The training set can be inconsistent, that is, some positive and negative examples are identical, so a correct rule can never be constructed. AQ provides three options, it treats these inconsistent examples either as positive examples, negative examples or simply neglects them. If statistical information about the probability of inconsistent examples is available, they are preclassified according to the maximum likelihood.

When the training set contains incorrect examples, it may be advantageous to apply the constructed rules in a probabilistic manner. So, apart from the *strict* matching, where one tests if an instance satisfies the description, one can distinguish another method of testing class membership: *flexible* matching, where the degree of similarity (or conceptual closeness) determines the class.

If flexible matching is used, it is possible to simplify a description by removing one or more complexes. This technique, called *truncation*, removes the complex that covers the least examples. If the training set is noisy, these complexes may be indicative of errors in the data. The resulting rule is no longer correct, but probabilistic, since similarities for different classes are compared. But these rules are simpler to understand, and the size of the knowledge base reduces.

At each truncation step, the classification accuracy is measured. Each step produces a different trade off between the complexity of the description and the accuracy of the rule. At some step, the best overall result may be achieved. Several experiments show that rules can be truncated without affecting the classification accuracy.

2.4 Constructive induction

The AQ system uses domain knowledge, expressed in the form of rules, to generate new attributes. These rules are of two types: logic rules that define values of new variables, and arithmetic rules that introduce new variables as functions of numerical attributes. The system attempts to use these new variables to produce better decision rules.

The logic rules represent background class definitions, constraints among classes, generalization hierarchies, causal dependencies, etc. Classes and their descriptions, learned by the program, are added to stock of rules.

2.5 Incremental learning

AQ15 has an incremental learning facility. The user may supply decision hypotheses as initial rules. The system implements the method of *learning with full memory*, where it remembers all examples that were seen so far, as well as the rules it formed. By this method, as opposed to *learning with partial memory*, new decision rules are guaranteed to be correct with respect to all (old and new) learning examples.

2.6 Conclusion

AQ15 focusses on the discovery of strong rules, and inconsistent and noisy data are handled by preprocessing (inconsistent examples), and postprocessing (rule truncation).

The system has been tested for several medical domains, such as diagnosis in lymphography, prognosis of breast cancer recurrence, and the location of a primary tumor. It discovered decision rules that performed at the level of accuracy of human experts. The training sets are small, typically a few hundred examples.

A potentially significant result of these tests is that application of the proposed method of rule truncation and flexible matching (i.e. constructing some kind of probabilistic rules) drastically decreases rule complexity without affecting classification accuracy.

3. CN2

The CN2 system, by Clark and Niblett (see [12]), is an adaptation of the AQ system. As we mentioned above, a disadvantage of the AQ algorithm is that the algorithm handles noise not by itself, but uses pre- and postprocessing (e.g. rule truncation). As Clark and Niblett point out, it is difficult to adapt the algorithm itself, because it strongly depends on specific training examples during search (the seed). The objective of CN2 is to remove this dependency, and incorporate noise handling technique (e.g. search space pruning) in the algorithm itself. CN2 combines the best features of both ID3 and AQ, where it uses pruning techniques similar to the techniques used in ID3, and the conditional rules used in AQ.

3.1 Search space

The output of the CN2 system is a decision list, i.e. an ordered set of if-then rules (see Section 1.3 in Chapter 6). The rules in this list resemble the ones used in the AQ system, that is, 'if ⟨complex⟩ then ⟨class⟩', with the exception that the conditional part is a complex and not a disjunction of complexes – a cover – as with the AQ rules.

During the search process, complexes are specialized by adding a conjunctive selector or by removing a disjunctive value in one of the selectors. The CN2 algorithm generates *all* possible specializations of a set of complexes (a star) by intersecting this set with the set of all possible selectors.

The quality criterion for complexes consists of two tests, a test to determine if the complex is *accurate* (has a high accuracy on the training set when predicting the majority class covered), and *significant* (the high accuracy on the training set is not due to chance).

Computing the accuracy of a complex involves first finding the set E of examples that are covered by the complex, and the probability distribution $P = (p_1, p_2, \ldots, p_n)$ of examples in E among the n classes. CN2 uses the information theoretic entropy measure

$$Entropy = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

to evaluate the quality (the lower the entropy, the better the quality). This function thus prefers complexes covering a large number of examples of a single class and few examples of other classes. These complexes score well on the training set when used to predict the majority class covered.

The complex should be significant as well, that is, the complex should locate a regularity unlikely to have occurred by chance, and thus reflect a genuine correlation between attribute values and classes. CN2 compares the observed distribution of examples among classes with the expected distribution that would result if the complex selected examples randomly. The system uses the likelihood ratio statistic, given by

$$2\sum_{i=1}^{n} f_i \log(f_i/e_i)$$

where the distribution $F = (f_1, \ldots, f_n)$ is the observed frequency distribution, and $E = (e_1, \ldots, e_n)$ is the expected frequency distribution. The statistic provides an information theoretic measure of distance between the two distributions. Under suitable assumptions, this statistic is distributed approximately as χ^2 with $n-1$ degrees of freedom. The lower the score, the more likely that the apparent regularity is due to chance. Only complexes that pass a user defined minimum threshold of significance are taken into consideration.

3.2 Search algorithm

The CN2 algorithm generates a decision list in an iterative fashion, constructing a rule at each iteration. A rule is constructed by searching for a complex that covers a large number of examples of an arbitrary class C_i and few of other classes (the construction of such a complex is described below). Having found a good complex, the algorithm removes those examples it covers from the training set and adds the rule 'if $\langle$complex$\rangle$ then predict C_i' to the end of the decision list. For the remaining set, a new rule is constructed, until no further complexes of sufficient quality are found.

The best complex in a training set is found using a beam search, i.e. the construction of a search graph, where at each iteration, the graph is extended at at most *maxstar* leaves. The beam width *maxstar* is defined by the user. At any moment during the search, the active leaves (i.e. complexes) are stored in a size limited set, called *star*. The system also keeps track of the best complex found so far.

Initially, the star contains only the root of the search graph, which is the empty complex, covering the entire training set. All specializations of all complexes in the star are examined by computing the intersection of the star and the set of all possible selectors. Complexes whose significance does not exceed the threshold are discarded. From the remaining complexes, the most interesting ones – as defined by the quality criterion – form the new star.

The process of searching the best complex terminates when no further complexes that pass the significance test can be found. The best complex, found in the search graph is returned, and forms the conditional part of a new rule. If no complex is found at all, a new rule cannot be constructed and the default rule is added to the end of the decision list.

3.3 Conclusion

CN2 constructs simple, comprehensible production rules in domains where noise may be present. It construct probabilistic rules, i.e. the conditional parts cover examples of a single class, but possibly a few examples of other classes as well. The result is not dependent on the order in which particular examples are chosen, as with the AQ system, and it searches a larger area of the search space, since it considers *all* possible extensions of a star. But the principal advantage of CN2 over AQ15 is that since the former supports a cut off mechanism, it does not restrict its search to only those rules that are consistent with the set of examples.

The performance of ID3, AQ, and CN2 has been compared on medical and artificial domains. The discovered knowledge structures are equivalent in terms of accuracy and complexity.

4. DBLearn

The DBLearn system, designed by Cai, Han and Cercone (see [7, 16]), uses domain knowledge to generate descriptions for predefined subsets in a relational database. Special features of this system are its bottom up search strategy, its use of domain knowledge in the form of hierarchies of attribute values, and its use of relational algebra. The set of examples is a relational table, that is, a set of n-ary tuples.

4.1 Search space

The system uses relational tables as knowledge structures: for each class, it constructs a relational table, whose attribute (column) names are a subset of the attribute names in the set of examples. A tuple can be seen as a logical formula, formed by the conjunction of its attribute-value pairs. A table, a set of tuples, can thus be seen as a disjunction of these conjunctions. Hence, the starting node in the search space is the set of examples, and the aim is to generalize this table to a class description: a much smaller table, covering all examples belonging to this class. The maximum number of tuples in the resulting table is specified by a user defined threshold.

There is a trade off in the size of this threshold. A small threshold leads to a simple rule with fewer disjuncts, but it can result in overgeneralization and the loss of some valuable information. However, a large threshold can preserve some useful information, but this can result in a relatively complex rule with many disjuncts and some inadequately generalized results.

The domains of some attributes are partially ordered, and form semi upper lattices, i.e. all values are smaller than the greatest value *'ANY'*. The values form an *IS_A*-hierarchy, where values denote generalizations of the values below them.

Two generalization operations are used to generate a class description from a set of examples:

Dropping conditions A tuple is a logical conjunction. Dropping all attribute-value pairs for a particular attribute in the table eliminates a conjunct, and thus generalizes the rule. This is a *projection* in relational algebra.

Climbing generalization tree Values in a particular attribute are replaced with a larger value in the hierarchy, thus generalizing the relation.

Applying the above operations to a relational table may result in identical tuples. These tuples are removed, thus reducing the size of the table.

4.2 Search algorithm: complete rules

The system constructs rules that are complete but not necessarily consistent, that is, the class description covers all examples for this class, and possibly some examples of other classes as well. The algorithm uses only positive examples and consists of four steps:

1. Task-relevant data is selected from the database, i.e. a single table, containing all positive examples is retrieved.

2. The attribute-oriented induction consists of applying the generalization operations on the table. A simple heuristic is used: if there are many distinct values in an attribute, and a higher level value is provided, the attribute is generalized by substituting each value with its higher level value. On the other hand, if there are many distinct values for an attribute, but no higher level value is provided, the attribute is simply dropped in all tuples in the table.

   Duplicated tuples are removed from the table, and the generalization process continues until the number of tuples is no more than the specified threshold.

3. If possible, the resulting table is simplified, i.e. transformed into a table, covering the same set. For example, if some tuples have identical values for all, except one, attributes, and the values for this attribute form *all* descendants of a symbol in the hierarchy, these tuples can be replaced with a single tuple, where the varying attribute has the symbol as value.

4. The table is transformed into a logic formula.

4.3 Search algorithm: consistent rules

The other algorithm, designed by Cai *et.al.* is the learning of consistent rules that are not necessarily complete, i.e. all examples covered by the description belong to the class, but not necessarily all positive examples are covered. This algorithm uses both positive and negative examples.

The algorithm is similar to the algorithm for the learning of characteristic rules, only the second step is slightly different. Instead of using a single table for positive examples, two tables, one containing positive and the other containing negative examples are used. However, both tables can share tuples. These tuples – called the *overlapping* tuples – are marked, and these marks are inherited when the tuples are generalized. Because generalization can produce new overlaps, a check should be performed after every generalization operation. Finally, the unmarked tuples in each table form the class descriptions for both classes.

4.4 Noise

Both algorithms are extended to handle noise and exceptions. A small number of unusual examples is regarded as noisy and exceptional data, and should be discarded. The algorithm is extended by adding quantitative information to the generalization process, and the elimination of tuples from the table is based on this information.

A special attribute *votes* is added to each tuple in the table. This attribute registers the number of tuples in the initial table that are generalized to this tuple in the current table. Based on the votes, two weights are defined for each tuple in the generalized table:

1. The *t-weight* is a measure for the *generality* of the description represented by this tuple. This weight is the ratio of the number of tuples covered by tuple q_j to the number of tuples covered by the entire table $\{q_1, \ldots, q_n\}$:

$$t\_weight(q_j) = \frac{votes(q_j)}{\sum_{i=1}^{n} votes(q_i)}$$

   A high t-weight implies that the tuple is induced from the majority of the data, and a low t-weight implies that the tuple is induced from some rare, exceptional cases. Tuples with a low t-weight are therefore removed from the table.

2. The *d-weight* is a measure for the *discriminating ability* of the description represented by the tuple. This weight is the ratio of the number of tuples that belong to the class and are covered by this tuple to all tuples that are covered by this tuple.

   The d-weight can only be constructed if the number of classes is two or more, i.e. only for consistent rules. A high d-weight indicates that the tuple is mainly generalized from the original tuples in the target class, a low d-weight indicates that the tuple is mainly derived from the contrasting class and from only a few (probably exceptional) cases in the target class. Hence, only tuples with a high d-weight are included in the classification rule.

4.5 Incremental learning

Incremental learning is implemented by using the votes information. If a tuple is added to the database, it is generalized to the same class level as the tuples in the generalized table. If the tuple is already in the table, the votes information for this tuple is incremented. Otherwise, it is inserted as a new tuple in the generalized table, and if the number of tuples in this table exceeds the threshold, further generalization of the table is performed.

4.6 Conclusion

The DBLearn system is a relatively simple system, using two generalization operations to construct descriptions. It is an instructive example of the use of domain knowledge in the generalization process.

The system uses an attribute-oriented generalization process. Hence, a description is a disjunction of conjunctions of attribute value conditions. Each of these conjunctions consists of conditions on the *same* attributes. This limits the set of descriptions that can be constructed (compared to the set of descriptions in e.g. AQ). The performance of the system is good, the time-complexity of the algorithm is $O(N \log N)$, where N is the number of tuples in the initial relation.

In incremental learning, only the votes information is updated, and possibly new tuples are added to the description. It is unclear whether this kind of incremental learning is correct, i.e. the resulting description need not be the same as the description that would be constructed from scratch using the updated database. In the latter case, another sequence of generalization operations could be performed (because the database has been updated), resulting in a different description, that cannot be constructed by updating votes information in the initial description.

The DBLearn approach can very well be integrated with database operations, since generalization operations are set-oriented, and both data and knowledge are represented as relational tables. In fact, both generalization operations are identical: dropping a condition is equivalent to replacing the values with a single top-value ANY, thus climbing a (very flat) generalization tree. A useful extension would be not to generalize all values for a particular attribute, but to allow partial generalization, i.e. replace only some of the values with a higher level value.

The system has been extended to deal with noise, and constructs probabilistic rules (using the votes information). Techniques to deal with noise are primitive, application of statistical techniques, such as the χ^2 test, could improve results. There are no techniques to deal with missing attribute values.

5. Meta-Dendral

The Meta-Dendral system is a special purpose machine learning system (see [14, 63]), designed for the automated discovery of rules of mass spectroscopy. We discuss this system, because it applies machine learning techniques to an entirely different data representation.

Mass spectroscopy is a technique to analyze the three dimensional structure of a molecule. A spectrometer shoots high energy electrons at the molecule, breaking the molecule into fragments. The spectrometer measures the spectrogram: the relative abundance of the fragments of each mass. Whether a molecule breaks at a particular bond depends on the structure around this bond, e.g. a carbon-oxygen ($C = O$) double bond will rarely break, while carbon-carbon ($C - C$) single bonds break more easily. Thus, a mass spectrogram provides information on the structure. The aim of the Meta-Dendral system is the discovery of rules that define where a molecule will break, given its structure.

5.1 Data structure
The input of the system consists of a set of related molecule structures and their mass spectrograms. Meta-Dendral generates a set of rules defining the relationship between substructures, found in these molecules, and broken bonds.

5.2 Search space
Knowledge is represented as rules of the form $R_1 \star R_2$, where both R_1 and R_2 denote fragments of arbitrary form and $\star$ is the broken bond. A rule states that any molecule containing this structure will break at this bond.

To each rule corresponds a subset of elements in the training set whose broken bonds are explained (i.e. covered) by this rule. Rules are refined using a specialization operation, where fragments R_1 or R_2 are replaced by molecule structures, specifying more information, such as $(R_3 - C) \star (C - R_4)$, specifying that the bond between two carbon atoms is broken. The rules that can be constructed form a hierarchy, where the general rules are predecessors of more specific rules, and the coverage of a rule subsumes the coverage of the rule below it. An example of such a hierarchy is shown in Figure 7.1.

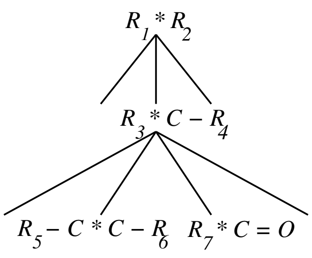


Figure 7.1: A hierarchy of Meta-Dendral rules.

The quality function is based on the generality of the rule, the classification accuracy, and the number of predicted fragmentations.

5.3 Search algorithm
The algorithm consists of three stages:

Preprocessing The *Intsum* subsystem uses the mass spectra to define which bonds in each molecule broke. It then generates possible fragmentations for each molecule and tests if these fragments fit into the spectrogram. The output consists of a list of proposed breaks, i.e. sets of fragments.

Inductive learning The *Rulegen* subsystem uses these fragments to recognize common constellations around broken bonds. The subsystem attempts to find a set of correct rules, i.e. rules that correctly predict the breakpoint in the molecules covered by this rule. Hereto, the system employs a top down search algorithm, starting with the most general rule $R_1 \star R_2$ and stepwise refining it, using all possible applications of the generalization operation. The refinement process continues iteratively, specifying more complex structures on both sides of the broken bond, until the quality of none of the generated children is better than its parents. The output of the induction stage consists of a list of generated rules.

Post-optimization The *Rulemod* subsystem modifies these rules to a smaller set of rules, that better explain the examples. Often, two or more rules can explain observed peaks in the spectrogram. Rulemod removes the rules that give too many false predictions, or rules that are more specific without having a higher classification accuracy. It also generalizes too specific rules, as long as this does not introduce false predictions. Typically, post-optimization decimates the number of rules, while increasing the quality.

The generated rules can be used by human analysts, or by an expert system—called *Dendral*. This system generates all possible structures for an unknown chemical structure, and uses rules to compute the fragments in which these structures would break. It then compares these fragments with the observed mass spectrum to find the best-fit.

5.4 Conclusion

The Meta-Dendral system was successful in finding previously unknown fragmentation rules for particular molecules. However, the search strategy is inefficient, since a large area of the search space is explored and many rules are constructed, of which many are discarded in the optimization stage. It is very difficult to find heuristics that guide the search. This limits the size of the molecules that could be explored using Meta-Dendral. There are no explicit techniques to handle noise and exceptional data in the rule construction algorithm, but post optimization is used to remove those rules that are due to noisy data.

6. RADIX/RX

The RX system is used for the discovery of relationships in a clinical database (see [5, 63, 64]). A very important difference with other systems is that the notion of *time* is included: the data objects in the set of examples store information on patients at different times and the generated knowledge consists of *causal* relationships. Furthermore, the system employs a two stage discovery process: first, it generates hypotheses, and afterwards it uses advanced statistical techniques to validate these.

6.1 Data representation

The training set consists of objects, where each object represents the information about a single patient. For each attribute in this object, values are recorded at different moments. We could thus represent the set of examples as a three dimensional matrix $P \times A \times T$: one dimension for P patients, one for A attributes for each patient and one for T different moments at which the value of each attribute is recorded, as shown in Figure 7.2.

6.2 Knowledge representation

The discovered knowledge is represented as a *causal model*: a directed labeled graph, where nodes represent properties and arcs represent causal relationships between properties. These properties can be either attributes directly stored in the database, such as *temperature*, or they can be derived from other attributes in the database, such as a diagnosis, using domain knowledge.

The arcs represent causal relationships between properties, thus A causes B means that:

1. **Correlation:** A is correlated with B, if an example has property A, it also has property B, and vice versa. However, causal relationships also denote some order in which the events take place:

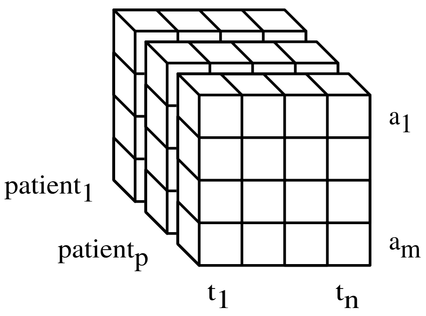


Figure 7.2: A 3D matrix storing patient data.

2. **Time precedence:** A is generally followed by B, that is, A is the *cause* and B is the *effect*;

3. **Nonspuriousness:** A and B are correlated if there is no known third property C responsible for the correlation, that is, there is no causal relationship between C and A, and C and B. C is called a *confounding* property.

Each arc is labeled to denote some extra information about the relation, such as:

intensity: the expected change in the effect given a change in the cause;

frequency: the distribution of the effect across patients;

direction: does B increase or decrease when A increases;

validity: to distinguish tentative associations from widely confirmed causal relationships.

The aim of the RX system is to find new causal relationships in a model, when provided part of this model and a training set. Finding these relationships is a two stage process: firstly, all possible relationships are generated and their quality is computed. Secondly, the validity of the most plausible of these relationships is tested. These stages are implemented in different modules. Both modules use a knowledge base, storing information about known relationships, statistical techniques, functional definitions of derived properties, etc.

6.3 Discovery module

The discovery module generates relationships and tests their probability by computing their covariation and checking their time-precedence. It produces a list of hypothesis ranked according to their probability. The main problem is that, given a set of N properties, a maximum of $N(N-1)$ binary relationships can be generated. In RX, the number of generated relationships is reduced by using techniques such as marking particular properties to be causes only (sources in the graph) and mark others as effects only (sinks in the graph).

Each generated relationship has to be tested. The correlation between the two properties is computed for different time lags, so that effects that were not immediate can also be observed, and the time precedence can be verified. A score for time precedence is computed based on correlations obtained when the cause is shifted forward and backward in time.

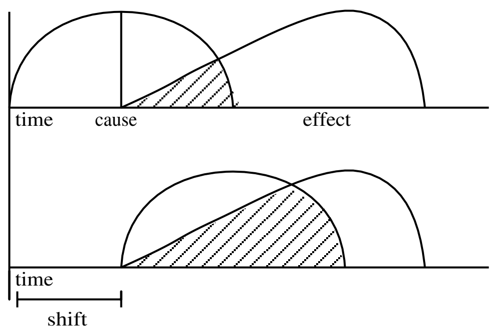


Figure 7.3: The principle underlying lagged correlation.

In Figure 7.3, this technique is illustrated. If the cause is shifted forward in time, the correlation (the shaded area) increases. Hence, the cause takes place before the effect, and both are correlated. However, the nonspuriousness of the relationship is not checked at this stage, but this is done by the study module.

6.4 Study module
The study module designs statistical models for the relationships with the highest probability. Information from the knowledge base is used to identify confounding properties, i.e. properties that influence both dependent and independent properties, and that may give false results if not controlled. If such properties exist, a method is chosen to control the effects, such as excluding the patients who have the confounding property, or using statistical techniques to control the influence of the property.

A *statistical package* is invoked by the study module to test the statistical model. It chooses an appropriate statistical method, generally multiple regression applied to individual patient records, and retrieves the relevant data from the database. The results of the evaluation are returned to the study module for interpretation.

If the relationship is statistically significant and important in the domain, it is incorporated in the model, i.e. stored in the knowledge base, labeled with some information such as validity, evidential basis, and so on.

6.5 Conclusion
The RX system used the ARAMIS database of rheumatology. One of the major results of the RX system was in conforming and quantitating the hypothesis that the drug prednisone increases cholesterol in the blood. The main drawback of the RX system is that it does not use domain knowledge to guide the search for interesting relationships. A new version of the system, called RADIX, uses medical knowledge to focus discovery.

Chapter 8
Numerical and hybrid learning systems

Most supervised learning systems construct knowledge structures that classify objects to a finite number of classes. For some applications however, we would like to predict the value of an unknown attribute in an object. A problem arises if the domain of this attribute is infinite, typically numerical. We can construct rules that predict the unknown value, but generally, we would end up with as many rules as there are different values.

A solution is to discretize these values, by mapping them to a finite number of symbols, e.g. $\{$*high, medium, low*$\}$ or $\{[0, 999], [1000, 1999], \ldots\}$. Although this may be convenient for some situations, discretization involves a loss of information. Therefore, we would like the result of the data mining process to be a knowledge structure that predicts numerical values, i.e. a function that returns a numerical value.

In this chapter we review two numerical learning systems, i.e. systems where both the predicting variables and the predicted variable are numerical.

1. Bacon

The Bacon system uses a data analysis algorithm to discover mathematical relationships in numerical data [26, 63]. It rediscover relationships such as Ohm's law for electric circuits and Archimedes' law of displacement.

The training set consists of numerical data, possibly generated at some previous experiment. Each example – a tuple – consists of the values that are measured for various variables in this experiment.

1.1 Search space

Bacon attempts to find a relationship between these variables that holds for all examples. A relationship is a numerical expression over these variables—a *term*. If this term has a constant value for all examples, then it describes a numerical relationship.

The search space consists of all terms that can be constructed from the variables, using some elementary numerical operations such as multiplication, addition and quotient. Operations on these terms, i.e. operations that transform terms in other terms (and thus allow the search algorithm to traverse the search space) include:

1. numerical operations such as multiplication or division of the term with a variable,
2. specialization of the term, by restricting the set of examples to a particular subset of the data,
3. creating terms for the slope and intercept of linear relations between two variables,
4. creating terms for constant modulo-n relations.

The preference criterion, i.e. the extend to which a term correctly describes a relationship in the training set, is simple: a term is correct if it has a constant value for all examples.

1.2 Search algorithm

The Bacon system employs a top down search, by generating a term, and checking the correctness of this term for the set of examples. As long as the preference criterion is not met, the term is modified by employing one of the above operations on the term.

The choice for these operations is guided by heuristics; the value of the term is computed for all objects, and these values are compared with the values for previously constructed terms. Examples of these heuristics are: If the values for the term and a previously constructed term increase monotonously, the ratio of the term and the variable is taken, or: if two terms vary inversely, the product is taken.

EXAMPLE 1 As an example, we will demonstrate the use of this algorithm to rediscover Kepler's third law—the law stating that the ratio of a planet's distance to the sun d, and the period of its revolution p are related as $d^3/p^2 = k$, where k is a constant.

p	d	d/p	d^2/p	d^3/p^2
1	1	1.0	1.0	1.0
8	4	0.5	2.0	1.0
27	9	0.33	3.0	1.0

Table 8.1: Planet distances and periods of revolution.

A set of examples, i.e. the values of p and d for some (strictly hypothetical) planets, are depicted in the first two columns of Table 8.1. Based on the observation that d and p increase monotonically, the term d/p is constructed. This term is not constant, as we can see in the third column, so the term is modified. Since d and d/p vary inversely, a new term $(d/p)d = d^2/p$ is constructed. This term varies inversely with d/p, so their product d^3/p^2 is computed. Computation of this term results in a constant for all examples and the algorithm halts. ∎

1.3 Intrinsic properties

Sometimes the set of examples consists of tests for different entities, instead of tests on a single entity. Each example is then labeled with its entity. If Bacon cannot construct a constant term for the entire set, but only a term which is constant for each entity, it assumes that these entities have some intrinsic property. For example, assume a test where we measure the current I and the resistance R of different batteries. Bacon would come up with a term IR, and notice that this term is constant for all tests for a particular battery, but differs for all batteries. It then proposes that the value of IR is an internal property of the batteries, which is of course correct, since $IR = V$, the voltage of the battery.

1.4 Conclusion

The Bacon system successfully rediscovered some empirical laws in chemistry and other areas. However, there are some drawbacks to the system: first, it assumes noise free, complete data, since it has no facilities to deal with inexact and incomplete data. An enhancement would be to use a more sophisticated preference criterion, such as the smallest squares method, to define the matching between a term and a set of numerical data.

Furthermore, the system assumes that *all* variables are relevant, and attempts to find a relationship between all these variables. Of course, some variables may be unrelated, or even have random values, thus prohibiting the construction of a correct term. A third problem is the efficiency of the algorithm. The rule, used in the selection of the appropriate operation is non-deterministic, so multiple paths in the search graph have to be explored. This results in an increasing number of terms that can be used to construct new terms. Hence, the performance of the system is very bad, even for small data sets. A solution may be to use more sophisticated heuristics.

2. KEDS

The Bacon system, and related systems such as Fahrenheit [65] and Fortyniner [66], construct a global function, i.e. a relationship that holds for all examples. This can only be successful when relationships in the data are actually homogeneous, that is, when the same relationship among variables holds for the entire training set.

However, many real-life engineering phenomena are multi-dimensional and nonhomogeneous: different relationships hold between variables in different subsets of the training set. Even if one is able to construct a single homogeneous function that accurately models the data, this function can be overly complicated and incomprehensible for humans.

The KEDS system (Knowledge based Equation Discovery System, see [53]) uses a divide-and-conquer strategy, where it breaks the problem space into smaller regions, and constructs functions for each of these regions. These functions are comprehensible and since they form a more expressive knowledge representation they are more likely to describe the relationships among the data correctly.

2.1 Search space

The model generated by the system consists of rules called *region-equation pairs*:

$$R_i \;\Rightarrow\; y = f_i(x_1, x_2, \ldots)$$

where R_i is the description of a region, i.e. a condition on attribute values. Variable y is the predicted variable, and $x_1, x_2, \ldots$ are predicting variables. Regions are hyperrectangles, hence their description consists of intervals for each attribute, e.g. $[.1 < x_1 < .7][3 < x_2 < 7]$.

A function $y = ?ax_1^2 + ?bx_2 + ?c$ is called a *template*, i.e. a function with unknown coefficients. The user defines a set of templates. These templates represent domain knowledge, i.e. they represent the user's expectations about possible forms of the numerical relationships.

The equation f_i in the region-equation pair is an instantiation of one of the user-defined templates. The model is a set of region-equation pairs, such that the regions form a partitioning of the training set.

2.2 Search algorithm
The KEDS algorithm is an iterative two phase top-down search process involving discovery and partitioning. Each invocation of the KEDS algorithm for a chosen template produces zero or more candidate region-equation pairs. KEDS uses some parameters to control the search: the *majority trend factor* m (the minimum fraction of the training set that must be described by the region-equation pair), and the *error bound* ϵ.

In the discovery phase, an equation is constructed that predicts part of the examples. First, a template is chosen, and examples are randomly chosen to compute the unknown coefficients in this template (if the template has n coefficients, n examples are needed to compute the values of these coefficients). The instantiated template – the equation f_i – has a *cover*, i.e. the set of examples whose response attributes are predicted with at most ϵ error by the equation.

If the fraction of the training set covered by the equation is greater than m, then the training set is partitioned into positive and negative examples, where positive examples are the examples covered by the equation.

These positive and negative examples are then used to determine the region in the second phase: the partitioning phase. A conventional supervised learning technique (a continuous version of the AQ algorithm) is used to find the description R_i of the covered region.

The search process is iterative, the discovery phase can be invoked to refine the equation, using only examples from the region. The refined equation is again passed to the partitioning phase, to find a refined region, and this process can be repeated until the region-equation pair stops improving (either in accuracy or in covering more data points).

Example 2 The following example, taken from [53], uses the training set depicted in Figure 8.1. The horizontal axis is multidimensional, representing the p-dimensional space x of predicting variables. In the initial discovery phase, the linear template $y = ?ax + ?b$ is chosen. To determine the coefficients, two example points from the training set are chosen (since there are two unknowns in this template). Two different candidate equations are shown in Figure 8.1a.

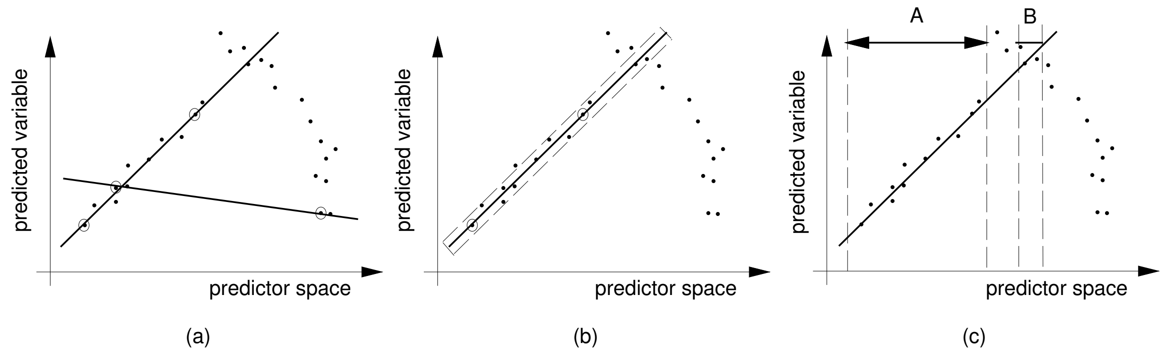


Figure 8.1: An example in KEDS: (a) sampling the training set, (b) computing the covers and (c) partitioning the training set.

The cover for each candidate equation is computed, and the equations that do not cover enough example points (according to the majority trend factor m) are discarded. Only contiguous regions are considered, so of the regions A and B (see Figure 8.1c), B is rejected because it covers fewer events than permitted by the majority trend factor m. For the remaining region A, the description is computed. ∎

2.3 Conclusion

Region-equation pairs are a more powerful knowledge representation than purely numerical functions, as used in the Bacon system. The models constructed by this system are comprehensible for humans. However, there is a trade off between comprehensibility and accuracy, but this can be controlled by the user. The learning speed is quite low, even on small datasets. Heuristics would be needed to guide the search, if we want to make this technique applicable to databases.

An improvement has been to use a probabilistic cover, which does not use an ϵ tolerance, but for each event in the training set, the probability that the value is correctly predicted by the equation is determined. The construction of regions is done using these probabilistic (fuzzy) sets. This makes the algorithm less sensitive to noise.

The system could be made applicable to hybrid domains (examples with symbolic and numerical attributes) by allowing the region descriptions to range over symbolic attributes as well. Such a representation would generalize both numerical and propositional-like representations.

Chapter 9
Conclusions and further research

We first summarize the most important topics in this paper by presenting them as a short course for data miners. Then, we will take a look in the future, and attempt to describe the most important research topics for data mining.

1. Data mining for beginners

The construction of a data mine system for a particular application consists of the following stages, although their order should not be seen as strict. At each stage, we outline the most important design considerations, and describe which techniques have been used in the systems that we described in previous chapters.

1.1 Defining the mining task

First of all, the user should define which relationships he or she wants to discover in the data. In the case of supervised learning, the user has to define which characteristics (e.g. classes) of objects the knowledge structures (rules, decision trees etc.) should predict, and which information (predicting variables) can be used for this prediction.

1.2 Selecting the data

Once the task has been defined, the data can be collected. Often, the training set will be extracted from a relational database, hence examples are tuples. These tuples should contain as much relevant information as available, and if possible, contain no irrelevant information. However, the relevance of information cannot always be determined in advance, so selecting information reflects the user's expectations about the hidden relationships, and therefore restricts the relationships that can be found.

Once the information *per object* is defined, the training set has to be selected from the database. Either the entire table can serve as a training set, or a randomly chosen subset can be used. The remaining information in the database can be used to check the quality of the discovered rules, i.e. serve as a test set.

1.3 Knowledge representation

Choosing an appropriate knowledge representation is one of the key decisions in the construction of a data mine system. The knowledge representation must fit the application domain, i.e. it should be able to describe hidden relationships in the data.

This poses a problem: to describe the relationships accurately, the representation has to be sufficiently complex. On the other hand, the acquired knowledge structure should be comprehensible for humans, hence it should be limited in size, and simple in structure. The optimal solution for a particular application is some compromise between these requirements.

Most systems that we discussed in this report use some variant of the relational algebra selection operator in their propositional-like representation, either decision trees (e.g. ID3), production rules (e.g. AQ, DBlearn), or decision lists (as in CN2). Some systems, such as Meta-Dendral, use representations that are suited for special application domains (chemistry), other, such as RX, use representations for expressing special kinds of relationships (e.g. causal relationships).

Some of the systems make use of domain knowledge (constructive induction in AQ15, generalization hierarchies in DBlearn). It is our believe that a data mine system should be able to use application specific knowledge, supplied by the user. It may be useful to employ a single representation formalism for both the knowledge structures and the domain knowledge. This representation has to be sufficiently expressive, e.g. first order logic.

1.4 Transformation operations

Most intelligent search algorithms, as we describe below, rely on operations that transform one knowledge structure into another. An algorithm uses these operations to traverse the search space. Care should be taken with these operations. First of all, the entire space should be connected under these operations, guaranteeing that the optimal solution can always be reached from the initial structure. Secondly, it would be preferable if the quality of the knowledge structure changes only smoothly under application of the operations. In other words, the operations should be chosen such that they align as much as possible with the quality function.

1.5 Quality function

Next, we have to construct a quality function. With each knowledge structure, we associate a certain quality that describes how 'interesting' the structure is, i.e. some measure for its information content. This criterion can be defined in terms of the classification accuracy of the knowledge structure with respect to its class, i.e. the higher the accuracy, the higher the quality. The criterion can be based on its understandibility for humans as well, thus, the criterion would prefer simple structures.

1.6 Search algorithm

The data mine system has to find the most preferred knowledge structure in the space of all structures. A search algorithm can be either *bottom up*, applying a sequence of generalization operations to the training set, or *top down*, where it repeatedly modifies an initial knowledge structure until it is correct with respect to the training set.

The simplest strategy, applicable to small spaces only, is *exhaustive search*. However, the search space is often too large, so only a small part can be searched. This part can be represented as a tree, where the nodes are structures, the edges are operations, and the initial structure is the root of the tree. The system navigates its way through the search space by selecting a sequence of operations. At each step, either a single operation is applied (a *hill climber*), or the n best operations (a *beam search*, i.e. multiple hill climbers in parallel, where n is the beam width), as depicted in Figure 9.1.

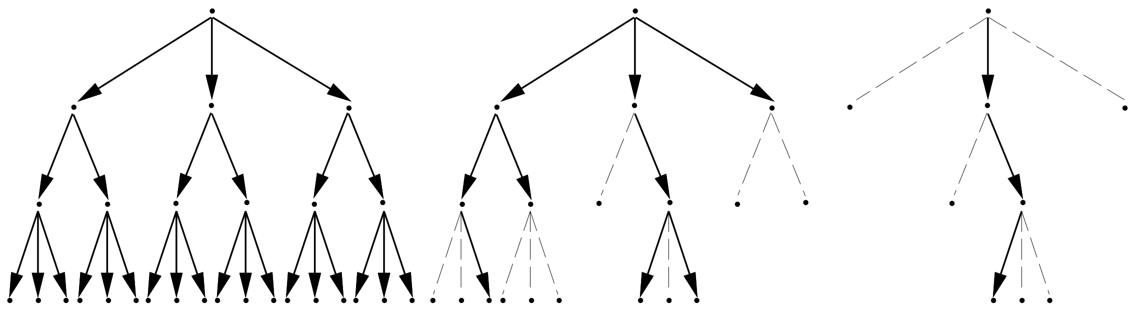


Figure 9.1: Search strategies.

1.7 Heuristics
Generally, the search algorithm does not chose the next operation at random, but uses heuristics to select the operations that are most likely to be on the shortest path towards an optimal solution. An often used heuristic is to estimate the improvements resulting from the application of operations to the current knowledge structure. The quality of the resulting structure can either be computed (using the quality function), or if this is too expensive, the quality can be estimated, using another (cheaper) criterion.

The algorithm can either estimate the quality of all possible extensions, or the quality of only some extensions. Some heuristics do not estimate the quality, but use other information from the training set, e.g. the Bacon system uses rules to select the next operation. The table below provides an overview of the search algorithms used in the different systems.

system	search strategy		extensions estimated
ID3	top down	hill climber	all
AQ15	top down	beam search	some
CN2	top down	beam search	all
DBlearn	bottom up	hill climber	all
Meta-Dendral	top down	exhaustive	–
RX	top down	exhaustive (constrained)	–
Bacon	top down	hill climber	none
KEDS (discovery)	top down	exhaustive	–
KEDS (partitioning)	top down	beam search	some

1.8 Noise and missing information
Data mining algorithms should be enhanced with techniques to deal with noise and missing information. These problems have been discussed extensively in Chapter 5, so we only briefly overview different techniques.

Most systems use some kind of *pruning* to deal with noise. The basic idea is that a small number of exceptional data is caused by noise, and can therefore be ignored. Hence, data mine systems need a statistical test to decide whether an observed relationship is due to chance or actually existent. Structures that do not stand this test are removed, i.e. pruned, either in the algorithm itself (e.g. decision tree pruning in ID3), or by postprocessing the knowledge structures (e.g. rule truncation in AQ).

Information can be incomplete, i.e. no value may be given for some attributes. A data mine system can either preprocess the training set (remove examples with missing values, or replace the missing value with the most likely one), or the algorithm can be adapted to handle missing values itself (e.g. ID3).

2. Research topics

As we stated in the introductory chapter, data mining becomes increasingly more important as the amount of information, stored in databases, grows. Hence, we may expect a growing interest for data mine tools, which in turn, is a reason for research in this area. The development of these tools benefits from research in three well founded area's: machine learning, statistics and databases.

2.1 Machine learning

Machine learning has a long tradition of design and experiments with knowledge representations and search strategies. Besides important theoretical results, e.g. PAC learnability, many learning algorithms and their implementations have been designed and tested for a wide variety of applications. Most of these systems search for totally *deterministic* relationships, and assume irregularities to be caused by noise. However, most systems have been enhanced with techniques to deal with noise, which makes them, to some extend, applicable to data mining applications, where most relationships will be probabilistic instead of deterministic.

2.2 Statistical techniques

However, we believe that results can be improved once we recognize that relationships in the data are actually *probabilistic* in nature. In searching for these relationships, the use of statistical techniques seems to be a natural choice. Experiments show that application of even very simple statistical techniques is a very promising direction, as for example shown in [12]. Here, the performance of the AQ, CN2 and ID3 algorithms is compared with the performance of a simple bayesian classifier on several domains. This classifier did not perform significantly worse in terms of accuracy than other algorithms, and completely outperformed all other systems in learning speed.

2.3 Intelligent database interfaces

One of the main obstacles in applying machine learning techniques to databases, is the size of the database. As we outlined in Chapter 5, this has consequences for the cost of evaluating the quality of a rule, and for the size of the search space.

A solution for the first of these problems is the application of database optimization techniques. Instead of using the entire database, only a subset will be used for the initial search phase. During the search process this set will be incrementally extended with data (i.e. additional proof) from the database, using incremental browsing optimization techniques. These techniques exploit the fact that previous database queries already loaded part of the data the current query has to load. Another topic will be the design of efficient caching algorithms, that closely interact with the search algorithm, and removes any obsolete data from the cache.

Solutions for the second of the above problems, i.e. the size of the search space, include using effective search strategies and heuristics. A very valuable source of heuristic information is the user—an expert in the application domain, so part of the research will focus on designing user interaction during the search process, and understandible representations for the knowledge.

Furthermore, domain knowledge, provided by the user may allow the discovery of relationships that would remain hidden otherwise. Representation of domain knowledge, and its application in the search algorithm will be an important topic.

3. Future directions

Current research focusses mainly on the discovery of classification rules (supervised learning) from database relations. This can be extended to the discovery of relationships involving database aggregates such as summation of values.

If multiple copies, taken at different times, of the same database are available, a data mine tool can search for trends (i.e. global changes over time), thus exploring another source of valuable information. However, the potentially most interesting direction may be in applying *unsupervised learning* techniques to a database. This would generate a compact representation of the information in the database, representing it as a number of subsets, each subset with its own description. In such a representation, again trends can be discovered, i.e. one can search for *migration streams* of objects from one subset to another over a period of time. Furthermore, supervised learning techniques can be used to search for characteristics for each of these migration streams.

Morik (in [38]) foresees three different scenarios for the application of machine learning techniques. So far, we followed the first scenario: the application of machine learning systems to current applications, such as databases. However, for many applications, using only a single learning strategy will not work. *Multistrategy* learning techniques use different algorithms depending on the application domain, or a combination of algorithms. Intelligent systems could select an appropriate learning technique themselves (see [10]). A third scenario consists of enhancing all kinds of conventional systems (e.g. editors, databases management systems, user-interfaces) by incorporating learning capabilities in these systems. As a result, computers will learn from their environment, thus narrowing the gap between man and computer.

References

1. Hassan Aït-Kaci and Patrick Lincoln. LIFE, a natural language for natural language. *T. A. Informations, Revue Internationale du traitement automatique du langage*, 30(1-2):51–89, 1989.

2. Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

3. Martin Anthony and Norman Biggs. *Computational learning theory: an introduction*, volume 30 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

4. Chidanand Apté, Sholom Weiss, and Gordon Grout. Predicting defects in disk drive manufacturing: a case study in high-dimensional classification. In *Proceedings of the 9th Conference on Artificial Intelligence for Applications*, pages 212 – 218, Orlando, Florida, 1993.

5. Robert L. Blum. *Discovery and Representation of Causal Relationships from a Large Time-Oriented Clinical Database: The RX Project*, volume 19 of *Lecture Notes in Medical Informatics*. Spinger-Verlag, 1982.

6. Pierre Brézellec and Henri Soldano. Samia: a bottom-up learning method using a simulated annealing algorithm. In *Proceedings of the European conference on Machine Learning*, Lecture notes in Artificial Intelligence, pages 297 – 309. Springer-verlag, 1993.

7. Yandong Cai, Nick Cercone, and Jiawei Han. Attribute-oriented induction in relational databases. In Piatetsky-Shapiro and Frawley [44], pages 213 – 228.

8. Jaime G. Carbonell, Ryszard S. Michalski, and Tom M. Mitchell. An overview of machine learning. In Michalski et al. [30], pages 3 – 24.

9. Chris Carter and Jason Catlett. Assessing credit card applications using machine learning. *IEEE Expert*, Fall 1987:71 – 79, 1987.

10. Philip K. Chan and Salvatore J. Stolfo. Experiments in multistrategy learning by meta-learning. In *Proceedings of the second international conference on information and knowledge management*, pages 314 – 323, Washington, DC, 1993.

11. Peter Clark. Knowledge representation in machine learning. In Yves Kodratoff and Alan Hutchinson, editors, *Machine and Human Learning, advances in European Research*, pages 35 – 49. Michael Horwood, London, 1989.

12. Peter Clark and Tim Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261 – 283, 1989.

13. P. Compton and R. Jansen. Knowledge in context: a strategy for expert system maintenance. In *Proceedings of the 2nd Australian Joint Artificial Intelligence conference*, volume 406 of *Lecture Notes in Artificial Intelligence*, pages 292 – 306, Adelaide, 1988. Springer.

14. Thomas G. Dietterich and Ryszard S. Michalski. A comparative review of selected methods for learning from examples. In Michalski et al. [30], pages 41 – 81.

15. Benjamin S. Duran and Patrick L. Odell. *Cluster analysis: a survey*, volume 100 of *Lecture Notes in Economics and Mathematical Systems*. Spinger-Verlag, 1974.

16. Jiawei Han, Yandong Cai, and Nick Cercone. Knowledge discovery in databases: An

attribute-oriented approach. In *Proceedings of the 18th VLDB Conference*, pages 547 – 559, Vancouver, British Columbia, Canada, 1992.

17. Geoffrey E. Hinton. Connectionist learning procedures. In Kodratoff and Michalski [25], pages 555 – 610.
18. John H. Holland. *Adaptation in natural artificial systems.* University of Michigan Press, Ann Arbor, 1975.
19. John H. Holland. Escaping brittleness: the possibilities of general purpose algorithms applied to parallel rule-based systems. In Michalski et al. [31], pages 593 – 623.
20. John H. Holland, Keith J. Holyoak, Richard E. Nisbett, and Paul R. Thagard. *Induction: processes of inference, learning and discovery.* Computational models of cognition and perception. MIT Press, Cambridge, 1986.
21. Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Conference record of the 14th annual ACM symposium on principles of programming languages*, pages 111 – 119, Munich, Germany, 1987.
22. Kenneth De Jong. Genetic-algorithm-based learning. In Kodratoff and Michalski [25], pages 611 – 638.
23. H.J. Kappen. Neurale netwerken, fuzzy rules en artificiele intelligentie. Foundation for Neural Networks.
24. Jyrki Kivinen, Heikki Mannila, and Esko Ukkonen. Learning rules with local exceptions. Technical report, University of Helsinki, 1993.
25. Yves Kodratoff and Ryszard S. Michalski, editors. *Machine Learning, an Artificial Intelligence approach*, volume 3. Morgan Kaufmann, San Mateo, California, 1990.
26. Pat Langley, Gary L. Bradshaw, and Herbert A. Simon. Rediscovering chemistry with the Bacon system. In Michalski et al. [31], pages 307 – 329.
27. D. Lenat. EURISKO: A program that learns new heuristics and domain concepts. The nature of heuristics III: Background and examples. *Artificial Intelligence*, 21:61 – 98, 1983.
28. Richard P. Lippmann. An introduction to computating with neural nets. *IEEE ASSP Magazine*, April:4 – 22, 1987.
29. Ryszard S. Michalski. A theory and methodology of inductive learning. In Michalski et al. [30], pages 83 – 134.
30. Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors. *Machine Learning, an Artificial Intelligence approach*, volume 1. Morgan Kaufmann, San Mateo, California, 1983.
31. Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors. *Machine Learning, an Artificial Intelligence approach*, volume 2. Morgan Kaufmann, San Mateo, California, 1986.
32. Ryszard S. Michalski, Igor Mozetic, Jiarong Hong, and Nada Lavrac. The AQ15 inductive learning system: an overview and experiments. Technical Report UIUCDCS-R-86-1260, University of Illinois, July 1986.
33. Ryszard S. Michalski, Igor Mozetic, Jiarong Hong, and Nada Lavrac. The multi-purpose

incremental learning system AQ15 and its testing application to three medical domains. In *Proceedings of the 5th national conference on Artificial Intelligence*, pages 1041 – 1045, Philadelphia, 1986.

34. Ryszard S. Michalski and Robert E. Stepp. Learning from observation: conceptual clustering. In Michalski et al. [30], pages 331 – 363.
35. Marvin Minsky. A framework for representating knowledge. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, pages 211 – 277. McGraw-Hill, New York, 1975.
36. Tom M. Mitchell, Paul E. Utgoff, and Ranan Banerji. Learning by experimentation: acquiring and refining problem-solving heuristics. In Michalski et al. [30], pages 163 – 190.
37. Raymond J. Mooney. Encouraging experimental results on learning CNF. Technical report, University of Texas, October 1992.
38. Katharina Morik. Applications of machine learning. In *Proc. 6th European Knowledge Acquisition Workshop*, pages 9 – 13. Springer-Verlag, Berlin, 1992.
39. Stephen Muggleton. *Inductive Logic Programming*, volume 38 of *A.P.I.C. series.* Academic Press Ltd., London, 1992.
40. Berndt Müller and Joachim Reinhardt. *Neural Networks, an introduction.* Physics of Neural Networks. Springer-Verlag, Berlin, 1991.
41. Nils J. Nilsson. *Principles of Artificial Intelligence.* Symbolic Computation. Springer-Verlag, 1982.
42. Kamran Parsaye and Mark Chignell. *Intelligent databases: tools & applications*, chapter 4. John Wiley & Sons, New York, 1993.
43. Gregory Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In Piatetsky-Shapiro and Frawley [44], pages 229 – 248.
44. Gregory Piatetsky-Shapiro and William J. Frawley, editors. *Knowledge Discovery in Databases.* AAAI Press, Menlo Park, California, 1991.
45. J. Ross Quinlan. Comparing connectionist and symbolic learning methods.
46. J. Ross Quinlan. Learning efficient classification procedures and their application to chess end games. In Michalski et al. [30], pages 463 – 482.
47. J. Ross Quinlan. The effect of noise on concept learning. In Michalski et al. [31], pages 149 – 166.
48. J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81 – 106, 1986.
49. J. Ross Quinlan. Generating production rules from decision trees. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 304 – 307, Milan, 1987.
50. J. Ross Quinlan. An emperical comparision of genetic and decision-tree classifiers. In *Proceedings of the 5th International Conference on Machine Learning*, pages 135 – 141, Ann Arbor, 1988.
51. J. Ross Quinlan. Determining literals in inductive logic programming. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 746 – 750,

Sydney, Austalia, 1991.

52. J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1992.

53. R. Bharat Rao and Stephen C-Y. Lu. A knowledge-based equation discovery system for engineering domains. *IEEE Expert*, August 1993:37 – 42, 1993.

54. Gordon A. Ringland and David A. Duce, editors. *Approaches to Knowledge Representation: An Introduction.* Research studies press Ltd., Letchworth, England, 1988.

55. Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2:229 – 246, 1987.

56. Claude Sammut and Ranan B. Banerji. Learning concepts by asking questions. In Michalski et al. [31], pages 167 – 191.

57. Sabrina Sestito and Tharam Dillon. Using single layered neural networks for the extraction of conjunctive rules and hierarchical classifications. *Journal of Applied Intelligence*, 1:157 – 173, 1991.

58. David C. Sills. William of Ockham. In *International Encyclopedia of the Social Sciences*, pages 269 – 270. Macmillan Company & The Free Press, New York, 1968.

59. William M. Spears and Kenneth De Jong. Using genetic algorithms for supervised concept learning. In *Proceedings of tools for AI*, 1990.

60. Jeffrey D. Ullman. *Principles of database and knowledge-base systems, volume 2*, volume 14 of *Principles of Computer Science.* Computer Science Press, 1989.

61. Paul E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161 – 186, 1989.

62. Les G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134 – 1142, 1984.

63. Michael G. Walker. How feasible is automated discovery. *IEEE Expert*, Spring 1987:69 – 82, 1987.

64. Gio C.M. Wiederhold, Michael G. Walker, Robert L. Blum, and Stephen M. Downs. Acquisition of knowledge from data. In *ACM SIGART International Symposium on Methodologies for Intelligent Systems*, pages 74 – 84, Knoxville, Tennessee, 1986.

65. Jan M. Zytkow. Combining many searches in the FAHRENHEIT discovery system. In *Proceedings of the fourth international workshop on machine learning*, pages 281 – 287, San Mateo, California, 1987. Morgan Kaufmann.

66. Jan M. Zytkow and John Baker. Interactive mining for regularities in databases. In Piatetsky-Shapiro and Frawley [44], pages 31 – 53.