Lecture Notes in Machine Learning

Zdravko Markov

May 28, 2003

Contents

Chapter 1

Introduction

1.1 Definition

Any change in a system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population (Simon, 1983).

1.2 Paradigms

Depending on the amount and type of knowledge available to the system before the learning phase (system's a priori knowledge) we can distinguish several situations:

- The simplest form of learning is just assigning values to specified parameters. This is a situation when the system contains all the knowledge required for a particular type of tasks.
- Another rudimentary type of learning is storing data as it is. This is called *rote learning*. An example of this type of learning is filling a database.
- The process of *knowledge acquisition* in an expert system is a kind of learning task where some pre-defined structures (rules, frames etc.) are filled with data specified directly or indirectly by an expert. In this case only the structure of the knowledge is known.
- The system is given a set of examples (*training data*) and it is supposed to create a description of this set in terms of a particular language. The a priori knowledge of the system is the syntax of the allowed language (*syntactical bias*) and possibly some characteristics of the domain from which the examples are drawn (*domain knowledge* or *semantic bias*). This is a typical task for *Inductive learning* and is usually called *Concept learning* or *Learning from examples*.
- There are learning systems (e.g.*Neural networks*) which given no a priori knowledge can learn to react properly to the data. Neural networks actually use a kind of a pre-defined structure of the knowledge to be represented (a network of neuron-like elements), which however is very general and thus suitable for various kinds of knowledge.

As in human learning the process of machine learning is affected by the presence (or absence) of a teacher. In the *supervised learning* systems the teacher explicitly specifies the desired output (e.g. the class or the concept) when an example is presented to the system (i.e. the system uses pre-classified data). In the *reinforcement learning* systems the exact output is unknown, rather an estimate of its quality (positive or negative) is used to guide the learning process. *Conceptual clustering* (category formation) and *Database discovery* are two instances of *Unsupervised learning.* The aim of such systems is to analyze data and classify them in categories or find some interesting regularities in the data without using pre-classified training examples.

Any change in the learning system can be seen as acquiring some kind of knowledge. So, depending on what the system learns we can distinguish:

- Learning to predict values of unknown function. This is often called *prediction* and is a task well known in statistics. If the function is binary, the task is called *classification.* For continuous-valued functions it is called *regression.*
- *Concept learning.* The systems acquires descriptions of concepts or classes of objects.
- *Explanation-based learning.* Using traces (explanations) of correct (or incorrect) performances the system learns rules for more efficient performance of unseen tasks.
- *Case-based (exemplar-based) learning.* The system memorizes cases (exemplars) of correctly classified data or correct performances and learns how to use them (e.g. by making analogies) to process unseen data.

Chapter 2

Concept learning

2.1 Learning semantic nets

The basic ideas of concept learning are introduced by P. Winston in his early system ARCH [4]. This system solves a concept learning task defined as follows: the input to the system are a priori knowledge (the knowledge that the system has before the learning stage takes place) and examples of some concept. The examples are of two types – positive (objects belonging to the concept) and negative (objects that do not belong to the concept). The task is to create a concept description (definition) that accepts (includes) the positive examples and rejects the negative ones.

The a priori knowledge consists of a language for describing the examples and other facts from the domain. The language used by ARCH is based on semantic networks. It includes objects (e.g. arch, block, pyramid) and relations (e.g. isa, partof, supports, touches, doesnottouch). The domain knowledge is represented by an object taxonomy, specified by a set of instances of the "isa" (is a) relation, such as `isa(pyramid, polygon)`, `isa(block, polygon)`.

Let us consider an example of building the concept of arch, given positive and negative examples. These exampes are the following (also shown in Figures 2.1 and 2.2):

```
Example1 = {partof(block1,arch), partof(block2,arch), partof(block3,arch),
            supports(block1,block3), supports(block2,block3)}

Example2 = {partof(block1,arch), partof(block2,arch), partof(pyramid1,arch),
            supports(block1,pyramid1), supports(block2,pyramid1)}

Apriori_knowledge = {isa(block1,block), isa(block2,block), isa(block3,block),
       isa(block,polygon), isa(pyramid1,pyramid), isa(pyramid,polygon)}
```

`Example1` and `example2` have the same structure and differ only in the object that is supported by `block1` and `block2`. This object is a block in example1 and pyramid in example2. According to the a priori knowledge both the block and the pyramid are polygons. This allows us to construct an object where these two parts are replaced by a polygon. Such an object will include both examples.

A transformation that carries relations from objects to other objects including the former ones as instances (successors in the object taxonomy) is called *generalization*. Thus the generalization of `example1` and `example2` is the object `hypothesis1`, shown below. Hereafter we call such an object *hypothesis*, since it is a kind of approximation of the *target object* (the object we are learning), and later on it will be a subject of verification (acception, rejection or update).

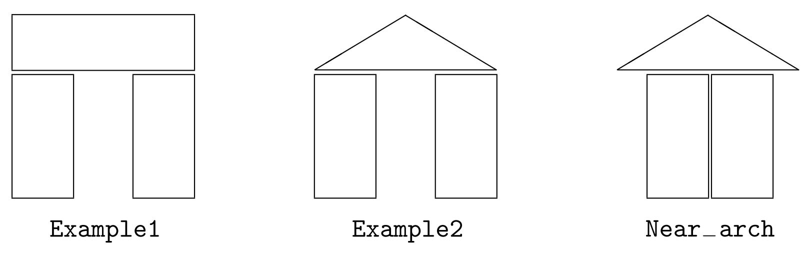


Figure 2.1: Examples of arch and "near arch"

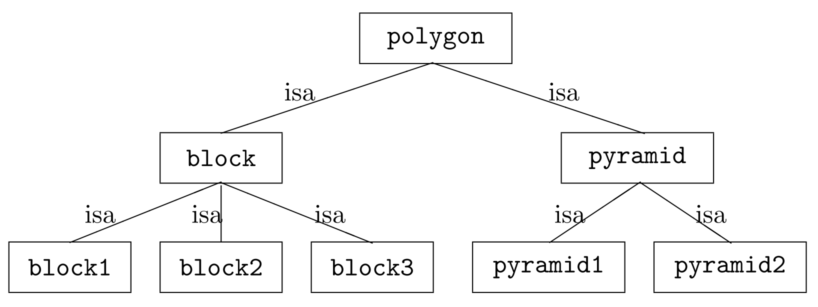


Figure 2.2: Object taxonomy (isa-hierarchy)

```
Hypothesis1 = {partof(block1,arch), partof(block2,arch), partof(polygon,arch),
               supports(block1,polygon), supports(block2,polygon)}
```

While the positive examples are instances of the target concept (arch), the negative ones are not just non-arch objects. They must be "near misses", i.e. objects with just one property (or relation) that if removed would make them belong to the target concept. This specific choice of negative examples is important for reducing the number of potential negative examples (imagine how many non-arhes exist) and possible hypotheses as well. Let us consider the following negative example of an arch (see Figure 2.1):

```
Near_miss = {partof(block1,arch), partof(block2,arch), partof(polygon,arch),
             supports(block1,polygon), supports(block2,polygon),
             touches(block1,block2)}
```

As the above object is a near miss, it is obvious that `touches(block1, block)` must be excluded from the hypothesis. This can be done by either imposing an explicit restriction to discard any objects with this relation, or to include a new relation that semantically excludes the "thouches" relation. For example, this might be the relation `doesnottouch(block1, block2)`. Thus the final hypothesis now is:

```
Hypothesis2 = {partof(block1,arch), partof(block2,arch), partof(polygon,arch),
               supports(block1,polygon), supports(block2,polygon),
               doesnottouches(block1,block2)}
```

This hypothesis satisfies the initial task by describing the concept of arch, including `example1` and `example2`, and excluding the negative example `near_miss`. The process of generating `hypothesis2` from `hypothesis1` is called *specialization*. That is, `hypothesis2` is *more specific* than `hypothesis1` (or conversely, `hypothesis1` is *more general* than `hypothesis2`), because `hypothesis1` includes (or covers) more examples (instances) than `hypothesis2`.

The ARCH algorithm developed by Patrick Winston in the early ages of AI implements the ideas described above. It is based on searching the space of possible hypotheses generated by generalization and specialization operators. The examples are processed one at a time and the for each one the best hypothesis is created. Then the system proceeds with the next example and modifies the currents hypothesis to accommodate the example (still accepting all previous examples). This kind of learning algorithm is called *incremental learning*.

The ARCH algorithm is applicable in other domains where the objects are described by relations (graphs or semantic nets). Despite of its simplicity, this algorithm illustrates the basic concepts and techniques of *inductive learning*: applying generalization/specialization operators, searching the hypothesis space, using background (domain) knowledge. It also exhibits some typical problems with inductive learning systems: the importance of the inductive bias and sensitivity to the type and the order of the training examples.

2.2 Induction task

Consider a formal system with a language L, three subsets of $L - L_B$ (*language of background knowledge*), L_E (*laguage of examples*) and L_H (*language of hypotheses*), and *a derivabilty relation* "$\rightarrow$" – a mapping between elements from L. An example of such a system is First-Order Logic (Predicate calculus), where the derivability relation is logical implication.

The induction task is defined as follows: Given *background knowledge* $B \in L_B$[1], *positive examples* $E^+ \in L_E$ and *negative examples* $E^- \in L_E$, find a hypothesis $H \in L_H$, under the following conditions:

[1]Hereafter we denote that sentence X is from language L as $X \in L$.

1. $B \not\rightarrow E^+$ (*nessecity*);
2. $B \not\rightarrow E^-$ (*consistency of* B);
3. $B \cup H \rightarrow E^+$ (*sufficiency*);
4. $B \cup H \not\rightarrow E^-$ (*consistency of* H).

There is an obvious solution to the above stated problem. This is the hypothesis $H = E^+$. However this solution is inappropriate due to the following reasons:

- This hypothesis derives only the positive examples E^+. However the solution to the induction task supposes a kind of inductive reasoning, i.e. the hypothesis must be able to accept new examples, that the learning system has not seen before. In other words, the hypothesis must be a piece of knowledge or a general rule applicable to a whole population of objects, where the examples are just a small sample from this population.

- The hypothesis does not explain the examples. Inductive reasoning assumes that the derived hypothesis not only accepts or rejects new examples (i.e. play the role of a classifier), but describes the examples in terms of the background knowledge as well. This, in turn, would allow the system to extend the background knowledge by adding hypotheses to it.

Despite of the above deficiencies the hypothesis $H = E^+$ is useful because it plays an essential role in searching the hypothesis space. It is called the *most specific* hypothesis and is denoted by the symbol $\perp$.

Obviously we need hypotheses more general than $\perp$. Here, the relation "more general" can be easily defined by using the intuitive notion of generality – the number of objects that a concept includes, i.e. a concept is more general than another concept if the former includes (derives, explains) more objects than the latter does.

Generality (subsumption, coverage) of hypotheses. Let H and H' be hypotheses, where $H \rightarrow E$ and $H' \rightarrow E'$. *H is more general than (subsumes, covers) H'*, denoted $H \geq H'$, if $E \supseteq E'$.

This ordering between hypotheses is often called *semantic ordering*, because it is based on the meaning of the hypothesis defined by the examples it covers and can be defined independently from the particular representation languages used.

Given the language of examples in most cases we can easily find the set of all possible examples and thus, the *most general* hypothesis $\top$, that covers all examples from L_E. Again, this hypothesis is unsuitable as a solution to the induction task, because it cannot be used to distinguish between positive and negative examples (it derives both E^+ and E^-). Even when $E^- = \emptyset$, $\top$ is not suitable either, because it is not constructive (does not describe the concept).

It is known that all subsets of a given set X form an algebraic structure called *lattice*. In this particular case the lattice is induced by the subset relation $\subseteq$ and usually denoted 2^X (because this is the number of elements in it). According to our definition of generality each hypothesis is associated with a set of examples. This fact suggests that the set of hypotheses might be a lattice. This in turn might be helpful when studying the hypothesis space because lattices are well studied algebraic structures with a lot of nice properties.

Unfortunately this approach gives just a theoretical framework for solving the induction task and cannot be used directly in practice. This is because the orderings between hypotheses generally do not conform to some specific requirements needed to define correct lattices. Even when all these requirements are met, further problems exists such as:

- Every hypothesis can be associated with a set of examples. The inverse however is not true. In many cases an explicit hypothesis for a given set of examples does not exits (within the given language) or if it does exist, there is a large number of such hypotheses (often infinite).
- In more complex languages (e.g. First-Order Logic) constructive operators for generalization/specialization cannot be found. In most cases such operators either do not exist or they are non-computable.

Due to the listed above reasons the orderings between hypotheses used in practice are mostly *syntactical*, i.e. they are determined by the representation language and have no relation to the examples they derive. These syntactical orderings are usually stronger (i.e. they hold for fewer objects) than the semantic ones. Consequently the syntactical orderings are *incomplete* – they do not guarantee exhaustive search in the hypothesis space. In other words, when searching the hypothesis space it is possible to skip over the desired hypothesis and generate a hypothesis that is either too specific or too general. These problems are known as *overspecialization* and *overgerenalization.*

It is clear that the hypotheses that meet the requirements of the induction task (extended with the requirements of inductive reasoning) can be found in a stepwise manner by generalizations of the most specific hypothesis $\perp$ or by specializations of the most general hypothesis $\top$. In other words the solution to the induction task comes to searching the hypothesis space, which is a kind of a hierarchical structure with an uppermost ($\top$) and a lowermost ($\perp$) elements. Each generalization/specialization is accomplished by the so called *generalization/specialization operators.* The choice of such operators and the way they are applied are determined by the following:

- The languages of examples and hypotheses (the so called *syntactic* or *language bias*);
- The strategy for searching the hypothesis space (*search bias*);
- The criteria for hypothesis evaluation and selection.

The choices of languages and a search strategy are the most important characteristics of the inductive learning system. These choices determine the type of examples and knowledge the system can deal with, and its performance as well. These two important choices are called *inductive bias.* Since in most cases there exist more than one hypotheses that satisfy the induction task, we need criteria for evaluating and selecting hypotheses as well.

Chapter 3

Languages for learning

3.1 Attribute-value language

The most popular way to represent examples and hypotheses is to use the so called *attribute-value* language. In this langauge the objects are represented as a set of pairs of an attribute (feature or characteristic) and its specific value. Formally this language can be defined as

$$L = \{A_1 = V_1, ..., A_n = V_n | V_1 \in V_{A_1}, ..., V_n \in V_{A_n}\},$$

where V_{A_i} is a set of all possible values of attribute A_i. For example, the set {`color = green`, `shape = rectangle`} describes a green rectangular object.

The attribute-value pair can be considered as a *predicate* (statement which can have a truth value) and the set of these pairs – as a *conjuction* of the corresponding predicates. Thus, denoting $p_1 =$ `(color = green)` and $p_2 =$ `(shape = rectangle)`, we get the formula $p_1 \wedge p_2$ in the language of *propositional calculus* (also called *propositional logic*). The propositional logic is a subset of the first order logic (or predicate calculus) without variables.

The basic advantage of the attribute-value language is that it allows a straightforward definition of derivability (covering, subsumption) relation. Generally such a relation (denoted $\geq$ and usually called subsumption) can be defined in three different ways depending of the type of the attributes:

- Attributes whose values cannot be ordered are called *nominal.* Using nominal attributes the subsumption relation is defined by dropping condition. For example, the class of objects defined by `(shape = rectangle)` is more general (subsumes) the class of objects `(color = green)` $\wedge$ `(shape = rectangle)`. Formally, let $X \in L$ and $Y \in L$, then $X \geq Y$, if $X \subseteq Y$.

- If we have a full order on the atribute values, then the attributes are called *linear*. Most often these are *numeric attributes* with real or integer values. The subsumption relation in this case is defined as follows: let $X \in L$, i. e. $X = \{A_1 = X_1, ..., A_n = X_n\}$ and $Y \in L$, i. e. $Y = \{A_1 = Y_1, ..., A_n = Y_n\}$. Then $X \geq Y$, if $X_i \geq Y_i$ (the latter is a relation between numbers) $(i = 1, ..., n)$.

- Attribute whose values can be partially ordered are called *structural.* The subsumption relation here is defined similarly to the case of linear attributes, i. e. $X \geq Y$, if $X_i \geq Y_i$ $(i = 1, ..., n)$, where the relation $X_i \geq Y_i$ is usually defined by a taxonomic tree. Then, if X_i and Y_i are nodes in this tree, $X_i \geq Y_i$, when $X_i = Y_i$, Y_i is immediate successor of X_i or, if not, there is a path from X_i to Y_i. (An example of taxonomy is shown in Figure 2.2.)

Using the above described language L as a basis we can define languages for describing examples, hypotheses and background knowledge. The examples are usually described directly in L, i.e. $L_E = L$. The language of hypotheses L_H is extended with a *disjunction*:

$$L_H = \{C_1 \vee C_2 \vee ... \vee C_n | C_i \in L, i \geq 1\}.$$

A notational variant of this language is the so called internal disjunction, where the disjunction is applied to the values of a particular attribute. For example, $A_i = V_{i_1} \vee V_{i_2}$ means that attribute A_i has the value either of V_{i_1} or of V_{i_2}.

The derivability relation in L_H is defined as follows: $H \rightarrow E$, if there exists a conjunct $C_i \in H$, so that $C_i \geq E$.

Similarly we define *semantic subsumption*: $H \geq_{sem} H'$, if $H \rightarrow E$, $H' \rightarrow E'$ and $E \supseteq E'$.

The subsumption relation in L induces a *syntactic partial order* on hypotheses: $H \geq H'$, if $\forall C_i \in H, \exists C_j \in H'$, such that $C_i \geq C_j$. Obviously, if $H \geq H'$, then $H \geq_{sem} H'$. The reverse statement however is not true.

As the hypotheses are also supposed to explain the examples we need an easy-to-understand notation for L_H. For this purpose we usually use *rules*. For example, assuming that $H = \{C_1 \vee C_2 \vee ... \vee C_n\}$ describes the positive examples (class $+$), it can be written as

```
if $C_1$ then $+$,
if $C_2$ then $+$,
...
if $C_n$ then $+$
```

Often the induction task is solved for more than one concept. Then the set E is a union of more than two subsets, each one representing a different concept (category, class), i.e. $E = \cup_{i=1}^{k} E^i$. This *multi-concept learning task* can be represented as a series of two-class ($+$ and $-$) concept learning problems, where for i-th one the positive examples are E^i, and the negative ones are $E \backslash E^i$. In this case the hypothesis for $Class_j$ can be written as a set of rules of the following type:

```
if $C_i$ then $Class_j$
```

To search the space of hypotheses we need constructive generalization/specialization operators. One such operator is the direct application of the subsumption relation. For nominal attributes generalization/specialization is achieved by dropping/adding attribute-value pairs. For structural attributes we need to move up and down the taxonomy of attribute values.

Another interesting generalization operator is the so called *least general generalization* (lgg), which in the lattice terminology is also called supremum (least upper bound).

Least general generalization (lgg). Let $H_1, H_2 \in L$. H is a least general generalization of H_1 and H_2, denoted $H = lgg(H_1, H_2)$, if H is a generalization of both H_1 and H_2 ($H \geq H_1$ and $H \geq H_2$) and for any other H', which is also a generalization of both H_1 and H_2, it follows that $H' \geq H$.

Let $H_1 = \{A_1 = U_1, ..., A_n = U_n\}$ and $H_2 = \{A_1 = V_1, ..., A_n = V_n\}$. Then $lgg(H_1, H_2) = \{A_1 = W_1, ..., A_n = W_n\}$, where W_i are computed differently for different attribute types:

- If A_i is nominal, $W_i = U_i = V_1$, when $U_i = V_i$. Otherwise A_i is skipped (i.e. it may take an arbitrary value). That is, $lgg(H_1, H_2) = H_1 \cap H_2$.

- If A_i is linear, then W_i is the minimal interval, that includes both U_i and V_i. The latter can be also intervals if we apply lgg to hypotheses.

- If A_i is structural, W_i is the closest common parent of U_i and V_i in the taxonomy for A_i.

```
example(1,pos,[hs=octagon, bs=octagon, sm=no, ho=sword, jc=red, ti=yes]).
example(2,pos,[hs=square, bs=round, sm=yes, ho=flag, jc=red, ti=no]).
example(3,pos,[hs=square, bs=square, sm=yes, ho=sword, jc=yellow, ti=yes]).
example(4,pos,[hs=round, bs=round, sm=no, ho=sword, jc=yellow, ti=yes]).
example(5,pos,[hs=octagon, bs=octagon, sm=yes, ho=balloon, jc=blue, ti=no]).
example(6,neg,[hs=square, bs=round, sm=yes, ho=flag, jc=blue, ti=no]).
example(7,neg,[hs=round, bs=octagon, sm=no, ho=balloon, jc=blue, ti=yes]).
```

Figure 3.1: A sample from the MONK examples

In the attribute-value language we cannot represent background knowledge explicitly, so we assume that $B = \emptyset$. However, we still can use background knowledge in the form of taxonomies for structural attributes or sets (or intervals) of allowable values for the nominal (or linear) attributes. Explicit representation of the background knowledge is needed because this can allow the learning system to expand its knowledge by learning, that is, after every learning step we can add the hypotheses to B. This is possible with relational languages.

3.2 Relational languages

Figure 3.1 shows a sample from a set of examples describing a concept often used in ML, the so called MONKS concept [3]. The examples are shown as lists of attribute-value pairs with the following six attributes: *hs*, *bs*, *sm*, *ho*, *jc*, *ti*. The positive examples are denoted by *pos*, and the negative ones – by *neg*.

It is easy to find that the + concept includes objects that have the same value for attributes *hs* and *bs*, or objects that have the value `red` for the *jc* attribute. So, we can describe this as a set of rules:

```
if [hs=octagon, bs=octagon] then +
if [hs=square, bs=square] then +
if [hs=round, bs=round] then +
if [jc=red] then +
```

Similarly we can describe class $-$. For this purpose we need 18 rules – 6 (the number of *hs*-*bs* pairs with different values) times 3 (the number of values for *jc*).

Now assume that our language allows variables as well as equality and inequality relations. Then we can get a more concise representation for both classes + and $-$:

```
if [hs=bs] then +
if [jc=red] then +
if [hs≠bs,jc≠red] then -
```

Formally, we can use the language of *First-Order Logic* (FOL) or *Predicate calculus* as a representation language. Then the above examples can be represented as a set of first order *atoms* of the following type:

```
monk(round,round,no,sword,yellow,yes)
```

And the concept of + can be written as a set of two atoms (capital leters are variables, constant values start with lower case letters):

```
monk(A,A,B,C,D,E)
monk(A,B,C,D,red,E)
```

We can use even more expressive language – the language of *Logic programming* (or *Prolog*). Then we may have:

```
class(+,X) :- hs(X,Y),bs(X,Y).
class(+,X) :- jc(X,red).
class(-,X) :- not class(+,X).
```

Hereafter we introduce briefly the syntax and semantics of logic programs (for complete discussion of this topic see [1]). The use of logic programs as a representation language in machine leanring is discussed in the area of *Inductive logic programming.*

3.3 Language of logic programming

3.3.1 Syntax

Fisrtly, we shall define briefly the language of First-Order Logic (FOL) (or Predicate calculus). The alphabet of this language consists of the following types of symbols: *variables, constants, functions, predicates, logical connectives, quantifiers and punctuation symbols.* Let us denote variables with alphanumerical strings beginning with capitals, constants – with alphanumerical strings beginning with lower case letter (or just numbers). The functions are usually denotes as f, g and h (also indexed), and the predicates – as p, q, r or just simple words as $father$, $mother$, $likes$ etc. As these types of symbols may overlap, the type of a paricular symbol depends on the context where it appears. The logical connectives are: $\wedge$ (*conjunction*), $\vee$ (*disjunction*), $\neg$ (*negation*), $\leftarrow$ or $\rightarrow$ (*implication*) and $\leftrightarrow$ (*equivalence*). The quantifiers are: $\forall$ (*universal*) and $\exists$ +*existential*). The punctuation symbols are: "(", ")" and ",".

A basic element of FOL is called *term*, and is defined as follows:

- a variable is a term;
- a constant is a term;
- if f is a n-argument function ($n \geq 0$) and $t_1, t_2, ..., t_n$ are terms, then $f(t_1, t_2, ..., t_n)$ is a term.

The terms are used to construct *formulas* in the following way:

- if p is an n-argument predicate ($n \geq 0$) and $t_1, t_2, ..., t_n$ are terms, then $p(t_1, t_2, ..., t_n)$ is a formula (called *atomic formula* or just *atom*;)
- if F and G are formulas, then $\neg F$, $F \wedge G$, $F \vee G$, $F \leftarrow G$, $F \leftrightarrow G$ are formulas too;
- if F is a formula and X – a variable, then $\forall X F$ and $\exists X F$ are also formulas.

Given the alphabet, the language of FOL consists of all formulas obtained by applying the above rules.

One of the purpose of FOL is to describe the meaning of natural language sentences. For example, having the sentence "For every man there exists a woman that he loves", we may construct the following FOL formula:

$$\forall X \exists Y man(X) \rightarrow woman(Y) \wedge loves(X, Y)$$

Or, "John loves Mary" can be written as a formula (in fact, an atom) without variables (here we use lower case letters for John and Mary, because they are constants):

$$loves(john, mary)$$

Terms/formulas without variables are called *ground* terms/formulas.

If a formula has only universaly quantified variables we may skip the quantifiers. For example, "Every student likes every professor" can be written as:

$$\forall X \forall Y is(X, student) \wedge is(Y, professor) \rightarrow likes(X, Y)$$

and also as:

$$is(X, student) \wedge is(Y, professor) \rightarrow likes(X, Y)$$

Note that the formulas do not have to be always true (as the sentences they represent).

Hereafter we define a subset of FOL that is used in logic programming.

- An atom or its negation is called *literal.*
- If A is an atom, then the literals A and $\neg A$ are called *complementary.*
- A disjunction of literals is called *clause.*
- A clause with no more than one positive literal (atom without negation) is called *Horn clause.*
- A clause with no literals is called empty clause ($\square$) and denotes the logical constant "false".

There is another notation for Horn clauses that is used in *Prolog* (a programming language that uses the syntax and implement the semantics of logic programs). Consider a Horn clause of the following type:

$$A \vee \neg B_1 \vee \neg B_2 \vee ... \vee \neg B_m,$$

where $A, B_1, ..., B_m$ ($m \geq 0$) are atoms. Then using the simple transformation $p \leftarrow q = p \vee \neg q$ we can write down the above clause as an implication:

$$A \leftarrow B_1, B_2, ..., B_m$$

.

In Prolog, instead of $\leftarrow$ we use $: -$. So, the Prolog syntax for this clause is:

$$A : -B_1, B_2, ..., B_m$$

.

Such a clause is called *program clause* (or *rule*), where A is the clause *head*, and $B_1, B_2, ..., B_m$ – the clause *body.* According to the definition of Horn clauses we may have a clause with no positive literals, i.e.

$$: -B_1, B_2, ..., B_m,$$

.

that may be written also as

$$? - B_1, B_2, ..., B_m,$$

.

Such a clause is called *goal.* Also, if $m = 0$, then we get just A, which is another specific form of a Horn clause called *fact.*

A conjunction (or set) of program clauses (rules), facts, or goals is called *logic program.*

3.3.2 Substitutions and unification

A set of the type $\theta = \{V_1/t_1, V_2/t_2, ..., V_n/t_n\}$, where V_i are all different variables ($V_i \neq V_j \forall i \neq j$) and t_i – terms ($t_i \neq V_i$, $i = 1, ..., n$), is called *substitution*.

Let t is a term or a clause. Substitution θ is applied to t by replacing each variable V_i that appears in t with t_i. The result of this application is denoted by $t\theta$. $t\theta$ is also called an *instance* of t. The transformation that replaces terms with variables is called *inverse substitution*, denoted by θ^{-1}. For example, let $t_1 = f(a, b, g(a, b))$, $t_2 = f(A, B, g(C, D))$ and $\theta = \{A/a, B/b, C/a, D/b\}$. Then $t_1\theta = t_2$ and $t_2\theta^{-1} = t_1$.

Let t_1 and t_2 be terms. t_1 is *more general* than t_2, denoted $t_1 \geq t_2$ (t_2 is *more specific* than t_1), if there is a substitution θ (inverse substitution θ^{-1}), such that $t_1\theta = t_2$ ($t_2\theta^{-1} = t_1$).

The term generalization relation induces a *lattice* for every term, where the lowemost element is the term itself and the uppermost element is a variable.

A substitution, such that, when applied to two different terms make them identical, is called *unifier*. The process of finding such a substitution is called *unification*. For example, let $t_1 = f(X, b, U)$ and $t_2 = f(a, Y, Z)$. Then $\theta_1 = \{X/a, Y/b, Z/c\}$ and $\theta_2 = \{X/a, Y/b, Z/U\}$ and both unifiers of t_1 and t_2, because $t_1\theta_1 = t_2\theta_1 = f(a, b, c)$ and $t_1\theta_2 = t_2\theta_2 = f(a, b, U)$. Two thers may have more than one unifier as well as no unifiers at all. If they have at least one unifier, they also must have a *most general unifier* (*mgu*). In the above example t_1 and t_2 have many unifiers, but θ_2 is the most general one, because $f(a, b, U)$ is more general than $f(a, b, c)$ and all terms obtained by applying other unifiers to t_1 and t_2.

An inverse substitution, such that, when applied to two different terms makes them identical, is called *anti-unifier*. In contrast to the unifiers, two terms have always an anti-unifier. In fact, any two terms t_1 and t_2 can be made identical by applying the inverse substitution $\{t_1/X, t_2/X\}$. Consequently, for any two terms, there exists a least general anti-unifier, which in the ML terminology we usually call *least general generalization* (*lgg*).

For example, $f(X, g(a, X), Y, Z) = lgg(f(a, g(a, a), b, c), f(b, g(a, b), a, a)$ and all the other anti-unifiers of these terms are more general than $f(X, g(a, X), Y, Z)$, including the most general one – a variable.

Graphically, all term operations defined above can be shown in a lattice (note that the lower part of this lattice does not always exist).

```
                 V
                ...
     anti-unifiers of t1 and t2
                ...
             lgg(t1,t2)
                /\
               /  \
              /    \
             /      \
           t1        t2
             \      /
              \    /
               \  /
                \/
            mgu(t1,t2)
                ...
      unifiers of t1 and t2
                ...
```

3.3.3 Semanics of logic programs and Prolog

Let P be a logic program. The set of all ground atoms that can be built by using predicates from P with arguments – functions and constants also from P, is called *Herbrand base* of P, denoted B_P.

Let M is a subset of B_P, and $C = A$ `:-` $B_1, ..., B_n$ $(n \geq 0)$ – a clause from P. M is a *model* of C, if for all ground instances $C\theta$ of C, either $A\theta \in M$ or $\exists B_j, B_j\theta \notin M$. Obviously the empty clause $\Box$ has no model. That is way we usually use the symbol $\Box$ to represent the logic constant "false".

M is a *model of a logic program* P, if M is a model of any clause from P. The intersection of all models of P is called *least Herbrand model*, denoted M_P. The intuition behind the notion of model is to show *when a clause or a logic program is true.* This, of course depends on the context where the clause appears, and this context is represented by its model (a set of ground atoms, i.e. facts).

Let P_1 and P_2 are logic programs (sets of clauses). P_2 is a *logical consequence* of P_1, denoted $P_1 \models P_2$, if every model of P_1 is also a model of P_2.

A logic program P is called *satisfiable* (intuitively, consistent or true), if P has a model. Otherwise P is unsatisfiable (intuitively, inconsistent or false). Obviously, P is unsatisfiable, when $P \models \Box$. Further, the *deduction theorem* says that $P_1 \models P_2$ is equivalent to $P_1 \wedge \neg P_2 \models \Box$.

An important result in logic programming is that the least Herbrand model of a program P is unique and consists of all ground atoms that are logical consequences of P, i.e.

$$M_P = \{A | A\ is\ a\ ground\ atom, P \models A\}$$

.

In particular, this applies to clauses too. We say that a clause C *covers* a ground atom A, if $C \models A$, i.e. A belongs to all models of C.

It is interesting to find out the logical consequences of a logic program P, i.e. *what follows from a logic program.* However, according to the above definition this requires an exhaustive search through all possible models of P, which is computationally very expensive. Fortunately, there is another approach, called *inference rules*, that may be used for this purpose.

An *inference rule* is a procedure I for transforming one formula (program, clause) P into another one Q, denoted $P \vdash_I Q$. A rule I is *correct and complete*, if $P \vdash_I P$ only when $P_1 \models P_2$.

Hereafter we briefly discuss a correct and complete inference rule, called *resolution*. Let C_1 and C_2 be clauses, such that there exist a pair of literals $L_1 \in C_1$ and $L_2 \in C_2$ that can be made complementary by applying a most general unifier μ, i.e. $L_1\mu = \neg L_2\mu$. Then the clause $C = (C_1 \backslash \{L_1\} \cup C_2 \backslash \{L_2\})\mu$ is called *resolvent* of C_1 and C_2. Most importantly, $C_1 \wedge C_2 \models C$.

For example, consider the following two clauses:

$C_1 = grandfather(X, Y) : -parent(X, Z), father(Z, Y).$
$C_2 = parent(A, B) : -father(A, B).$

The resolvent of C_1 and C_2 is:

$C_1 = grandfather(X, Y) : -father(X, Z), father(Z, Y),$

where the literals $\neg parent(X, Z)$ in C_1 and $parent(A, B)$ in C_2 have been made complementary by the substitution $\mu = \{A/X, B/Z\}$.

By using the resolution rule we can check, if an atom A or a conjunction of atoms $A_1, A_2, ..., A_n$ logically follows from a logic program P. This can be done by applying a specific type of the resolution rule, that is implemented in Prolog. After loading the logic program P

in the Prolog database, we can execute queries in the form of $?-A.$ or $?-A_1, A_2, ..., A_n.$ (in fact, goals in the language of logic programming). The Prolog system answers these queries by printing "yes" or "no" along with the substitutions for the variables in the atoms (in case of yes). For example, assume that the following program has been loaded in the database:

```
grandfather(X,Y) :- parent(X,Z), father(Z,Y).
parent(A,B) :- father(A,B).
father(john,bill).
father(bill,ann).
father(bill,mary).
```

Then we may ask Prolog, if $grandfather(john, ann)$ is true:

```
?- grandfather(jihn,ann).
yes
?-
```

Another query may be "Who are the grandchildren of John?", specified by the following goal (by typing ; after the Prolog answer we ask for alternative solutions):

```
?- grandfather(john,X).
X=ann;
X=mary;
no
?-
```

Chapter 4

Version space learning

Let us consider an example. We shall use an *attribute-value* language for both the examples and the hypotheses $L = \{[A, B], A \in T_1, B \in T_2\}$. T_1 and T_2 are taxonomic trees of attribute values. Let's consider the taxonomies of colors (T_1) and planar geometric shapes (T_2), defined by the relation *cat* (short for category).

```
Taxonomy of Colors:                  Taxonomy of Shapes:

cat(primary_color,any_color).        cat(polygon,any_shape).
cat(composite_color,any_color).      cat(oval,any_shape).
cat(red,primary_color).              cat(triangle,polygon).
cat(blue,primary_color).             cat(quadrangle,polygon).
cat(green,primary_color).            cat(rectangle,quadrangle).
cat(orange,composite_color).         cat(square,quadrangle).
cat(pink,composite_color).           cat(trapezoid,quadrangle).
cat(yellow,composite_color).         cat(circle,oval).
cat(grey,composite_color).           cat(ellipse,oval).
```

Using the hierarchically ordered attribute values in taxonomies we can define the derivability relation ($\rightarrow$) by the *cover* relation ($\geq$), as follows: $[A_1, B_1] \geq [A_2, B_2]$, if A_2 is a successor of A_1 in T_1, and B_2 is a successor of B_1 in T_2. For example, $[red, polygon] \geq [red, triangle]$, and $[any\_color, any\_shape]$ covers all possible examples expressed in the language L.

Let $P \in L, Q \in L$, and $P \geq Q$. Then P is a *generalization* of Q, and Q is a *specialization* of P.

Let $E^+ = \{E_1^+, E_2^+\}, E_1^+ = [red, square], E_2^+ = [blue, rectangle]$,
$E^- = [orange, triangle]$, and $B = \oslash$.

Then the problem is to find such a hypothesis H, that $H \geq E_1^+$, $H \geq E_2^+$, i.e. H is a *generalization* of E^+.

Clearly there are a number of such generalizations, i.e. we have a *hypothesis space*
$S_H = \{[primary\_color, quadrangle]$, $[primary\_color, polygon]$, ...,
$[any\_color, any\_shape]\}$.

However not all hypotheses from S_H satisfy the consistency requirement, ($H \not\geq E^-$), i.e. some of them are *overgeneralized.* So, the elements $H \in S$, such that $H \geq E^-$, have to be excluded, i.e they have to be *specialized*, so that not to cover any more the negative example. Thus we obtain a set of *correct* (consistent with the examples) hypotheses, which is called *version space*, $VS = \{[primary\_color, quadrangle]$, $[any\_color, quadrangle]\}$.

Now we can add the obtained hypotheses to the background knowledge and further process other positive and negative examples. Learning systems which process a sequence of examples one at a time and at each step maintain a consistent hypotheses are called *incremental learning systems.* Clearly the basic task of these systems is to search through the version space. As we have shown above this search can be directed in two ways – *specific to general* and *general to specific.*

4.1 Search strategies in version space

To solve the induction problem the version space have to be searched through in order to find the best hypothesis. The simplest algorithm for this search could be the generate-and-test algorithm, where the generator produces all generalizations of the positive examples and the tester filters out those of them which cover the negative examples. Since the version space could be very large such an algorithm is obviously unsuitable. Hence the version space has to be structured and some directed search strategies have to be applied.

4.1.1 Specific to general search

This search strategy maintains a set S (a part of the version space) of *maximally specific generalizations.* The aim here is to avoid overgeneralization. A hypothesis H is maximally specific if it covers all positive examples, none of the negative examples, and for any other hypothesis H' that covers the positive examples, $H' \geq H$. The algorithm is the following:

```
Begin
Initialize $S$ to the first positive example
Initialize $N$ to all negative examples seen so far
For each positive example $E^+$ do begin
    Replace every $H \in S$, such that $H \not\geq E^+$, with all its generalizations that cover $E^+$
    Delete from $S$ all hypotheses that cover other hypotheses in $S$
    Delete from $S$ all hypotheses that cover any element from $N$
    End
For every negative example $E^-$ do begin
    Delete all members of $S$ that cover $E^-$
    Add $E^-$ to $N$
    End
End
```

4.1.2 General to specific search

This strategy maintains a set G (a part of the version space) of *maximally general hypotheses.* A hypothesis H is maximally general if it covers none of the negative examples, and for any other hypothesis H' that covers no negative examples, $H \geq H'$. The algorithm is the following:

```
Begin
Initialize $G$ to the most general concept in the version space
Initialize $P$ to all positive examples seen so far
For each negative example $E^-$ do begin
    Replace every $H \in G$, such that $H \geq E^-$, with all its specializations that do not cover
        $E^-$
    Delete from $G$ all hypotheses more specific (covered by) other hypotheses in $G$
    Delete from $G$ all hypotheses that fail to cover some example from $P$
    End
For every positive example $E^+$ do begin
    Delete all members of $G$ that fail to cover $E^+$
    Add $E^+$ to $P$
    End
End
```

4.2 Candidate Elimination Algorithm

The algorithms shown above generate a number of plausible hypotheses. Actually the sets S and G can be seen as boundary sets defining all hypotheses in the version space. This is expressed by the *boundary set theorem* (Genesereth and Nilsson, 1987), which says that for every element H from the version space there exist $H' \in S$ and $H'' \in G$, such that $H \geq H'$ and $H'' \geq H$. In other words the boundary sets S and G allows us to generate every possible hypothesis by generalization and specialization of their elements, i.e. every element in the version space can be found along the generalization/specialization links between elements of G and S. This suggests an algorithm combining the two search strategies of the version space, called *candidate elimination algorithm* (Mitchel, 82).

The candidate elimination algorithm uses bi-directional search of the version space. It can be easily obtained by putting together the algorithms from section 2.1 and 2.2 and replacing the following items from them:

1. Replace "Delete from S all hypotheses that cover any element from N" with "Delete from S any hypothesis not more specific than some hypothesis in G"

2. Replace "Delete from G all hypotheses that fail to cover some example from P" with "Delete from G any hypothesis more specific than some hypothesis in S"

These alterations are possible since each one of them implies what it alters. Thus collecting all positive and negative examples in the sets P and N becomes unnecessary. Clearly this makes the bi-directional algorithm more efficient. Furthermore using the boundary sets two stopping conditions can be imposed:

1. If $G = S$ and both are singletons, then stop. The algorithm has found a single hypothesis consistent with the examples.

2. If G or S becomes empty then stop. Indicate that there is no hypothesis that covers all positive and none of the negative examples.

4.3 Experiment Generation, Interactive Learning

The standard definition of the inductive learning problem assumes that the training examples age given by an independent agent and the learner has no control over them. In many cases, however, it is possible to select an example and then to acquire information about its classification. Learning systems exploring this strategy are called *interactive learning systems.* Such a system use an agent (called oracle) which provides the classification of any example the systems asks for. Clearly the basic problem here is to ask in such a way that the number of further questions is minimal.

A common strategy in such situations is to select an example which halves the number of hypotheses, i.e. one that satisfies one halve of the hypotheses and does not satisfy the other halve.

Within the framework of the version space algorithm the halving strategy would be to find an example that does not belong to the current version space (otherwise its classification is known - it has to be positive) and to check it against all other hypotheses outside the version space. Clearly this could be very costly. Therefore a simple strategy is the following (this is actually and interactive version of the candidate elimination algorithm):

1. Ask for the first positive example
2. Calculate S and G using the candidate elimination algorithm
3. Find E, such that $G \geq E, \forall s \in S, E \not\geq s$ (E is not in the version space).
4. Ask about the classification of E

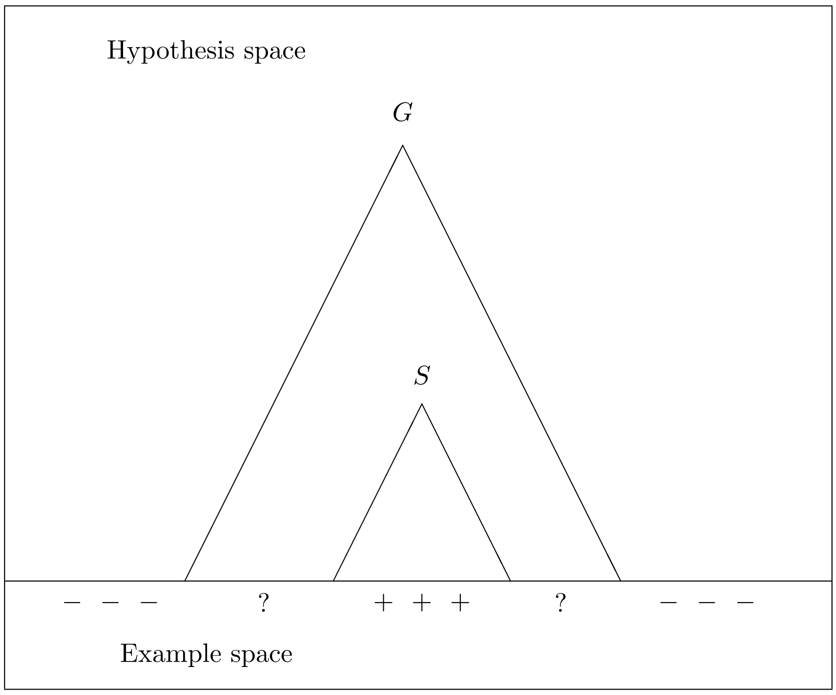


Figure 4.1: Graphical representation of version space. Question marsk denote the areas from where new examples are chosen

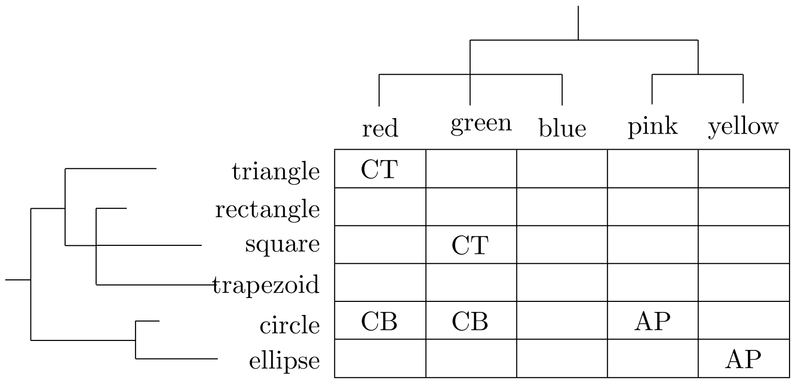
5. Go to 2

The exit from this loop is through the stopping conditions of the candidate elimination algorithm (item 2). A graphical illustration of the experiment generation algorithm is shown in Figure 4.1

4.4 Learning multiple concepts – Aq, AQ11

4.4.1 Aq

4.4.2 AQ11



	red	green	blue	pink	yellow
triangle	CT				
rectangle					
square		CT			
trapezoid					
circle	CB	CB		AP	
ellipse					AP

Figure 4.2: Multi-concept example space

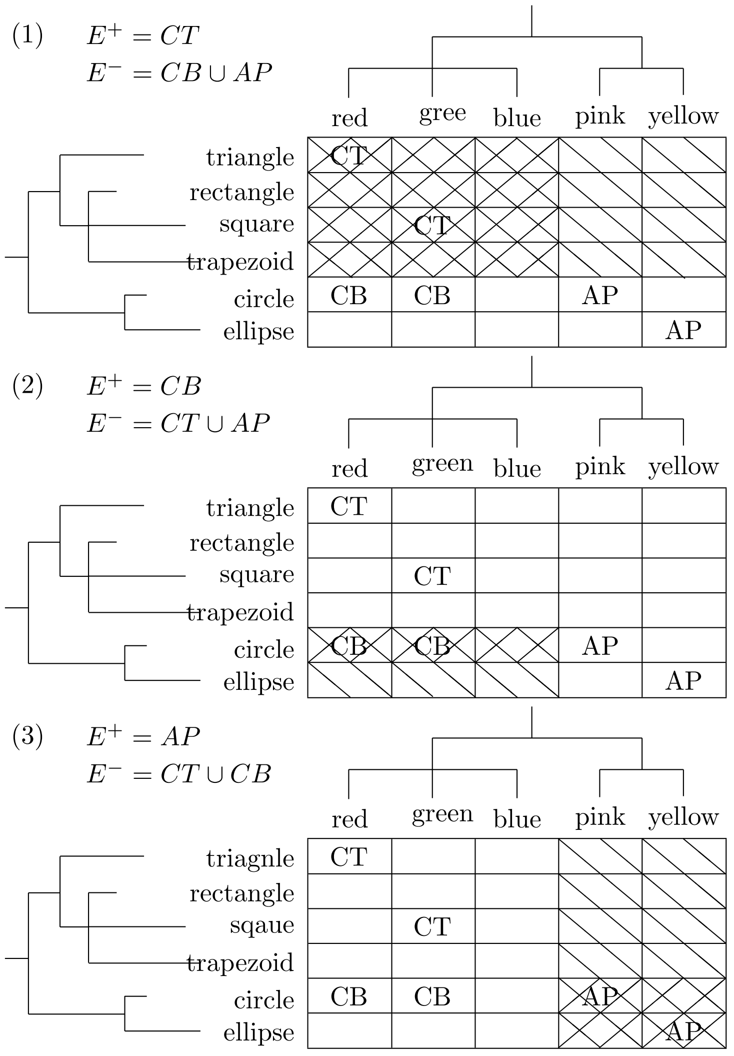

(1) $E^+ = CT$
$E^- = CB \cup AP$
red gree blue pink yellow
triangle rectangle square trapezoid circle ellipse
CT CT CB CB AP AP
(2) $E^+ = CB$
$E^- = CT \cup AP$
red green blue pink yellow
triangle rectangle square trapezoid circle ellipse
CT CT CB CB AP AP
(3) $E^+ = AP$
$E^- = CT \cup CB$
red green blue pink yellow
triagnle rectangle sqaue trapezoid circle ellipse
CT CT CB CB AP AP

Figure 4.3:

Chapter 5

Induction of Decision Trees

5.1 Representing disjunctive concepts

Consider the problem domain described by the attribute-value language L (discussed in Chapter 2) and the following set of classified examples:

$E^+ = \{[red, circle], [blue, triangle], [blue, square]\}$
$E^- = \{[red, square], [red, triangle]\}$

The candidate elimination algorithm applied to these data cannot produce a correct hypothesis. This is because there exist positive examples, whose least generalization covers some negative examples. For example, there is no hypothesis covering both $[red, circle]$, and $[blue, square]$ and at the same time not covering the negative example $[red, square]$.

This problem is due to the very restricted language for the hypotheses we use in the candidate elimination algorithm. Clearly the above data require a concept description involving a *disjunction* between two subdomains in the hypothesis space.

A description which can cope with such data is the *decision tree*. A decision tree can be represented in various forms. Here is a decision tree for classification of the above training set shown in two ways:

1. *Tree.* Each node represents an attribute (e.g. color), and the branches from the node are the different choices of the attribute values. Clearly the branches represent disjunctive relations between attribute values (the color can be blue OR red, and both branches belong to the concept). The leaves of the tree actually represent the classification. Each one is marked YES or NO, depending on whether the particular choice of values along the path to this leaf specifies a positive or a negative example, correspondingly.

```
     color
       |
  ---------
  |         |
 blue      red
  |         |
 YES      shape
            |
    ----------------
   |        |       |
triangle  square  circle
   |        |       |
   NO       NO     YES
```

2. *Set of rules.* These rules actually represent the paths in the decision tree.

```
IF color = blue THEN YES
IF color = red AND shape = circle THEN YES
IF color = red AND shape = square THEN NO
IF color = red AND shape = triangle THEN NO
```

A natural question is "how can the decision tree generalize". The above tree is a typical example of generalization. The first left branch of the tree leads to a leaf, which is determined by fixing only one of the attributes (color). Thus we allow the other one (shape) to have any value and hence to cover a number of positive examples. Actually in the taxonomic language L thiswould be the concept $[blue, any\_shape]$.

5.2 Building a decision tree

There are many algorithms for building decision trees. Many of them refer to ID3 [Quinlan, 1986]. Actually it is a family of concept learning algorithms, called TDIDT (Top-Down Induction of Decision Trees), which originated from the Concept Learning System (CLS) of [Hunt, Marin and Stone, 1966].

The basic algorithm is the following [Dietterich et al, 1982]. Its input is a set of training instances E, and its output is a decision tree.

1. If all instances in E are positive, then create a YES node and halt. If all instances in E are negative, then create a NO node and halt. Otherwise, select (using some heuristic criterion) an attribute, A, with values $V_1 ... V_n$ and create the decision node

```
         A
         |
   -------------
  |   |  |      |
  |   |  |      |
 V1  V2  V3 ... Vn
```

2. Partition the training instances in E into subsets $E_1, E_2, ..., E_n$, according to the values of $A(\{V_1, V_2, ..., V_n\})$

3. Apply the algorithm recursively to each of the sets E_1, E_2 etc.

Here is an example how this algorithm works. Consider the following set E of 6 instances (the positive are marker with "+", and the negative – with "-"):

```
1. [red,circle] +
2. [red,square] -
3. [red,triangle] -
4. [blue,triangle] -
5. [blue,square] -
6. [blue,circle] -
```

Initially the set of instances E is just the complete sequence of instances. These are neither uniformly positive or uniformly negative so the algorithm selects an attribute A and creates a decision node. Assume that the *shape* attribute is chosen. It has possible values

$\{triangle, square, circle\}$. Therefore a decision node is created which has a branch corresponding to each value.

The set E is now partitioned into subsets E_1, E_2 etc. according to the possible values of "shape". Instances with $shape = triangle$ all end up in one subset, instances with $shape = circle$ all end up in another and so on.

The algorithm is now applied recursively to the subsets E_1, E_2 etc. Two of these now contain single instances. The set of instances with $shape = triangle$ is just $\{3\}$, while the set of instances with $shape = square$ is just $\{2\}$. Thus two NO nodes are created at the end of the corresponding branches.

The set of instances with $shape = circle$ is $\{1, 6\}$. It does not contain uniformly positive or negative instances so a new feature is selected on which to further split the instances. The only feature left now is *color*. Splitting the instances on this feature produces the final two leaves (a YES node and a NO node) and the algorithm terminates, having produced the following decision-tree:

```
          shape
            |
    ------------------
   |        |          |
triangle  square     circle
   |        |          |
   NO       NO       color
                       |
                   ---------
                  |         |
                 red       blue
                  |         |
                 YES        NO
```

5.3 Informaton-based heuristic for attribute selection

Clearly, we will want the algorithm to construct small, bushy trees, i.e. simple decision rules. However, the degree to which it will do so depends to an extent on how clever it is at selecting "good" attributes on which to split instances.

Selecting "good" attributes means giving priority to attributes which will best sort the instances out into uniform groups. So the question is, how can the algorithm be provided with a criterion, which will enable it distinguish (and select) this sort of attribute.

Several approaches have been explored. The most well-known is Quinlan's which involves calculating the *entropy* of the distribution of the positive/negative instances resulting from splitting on each of the remaining attributes and then using the attribute which achieves the lowest entropy distribution.

The entropy measure is based on the information theory of Shannon. According to this theory we can calculate the information content of the training set, and consequently, of any decision tree that covers this set of examples.

If we assume that all the instances in the training set E (the example above) occur with equal probability, then $p(YES) = 1/6$, $p(NO) = 5/6$. The information in E is:

$$I(E) = -\frac{1}{6}log_2(\frac{1}{6}) - \frac{5}{6}log_2(\frac{5}{6}) = 0.4308 + 0.2192 = 0.65$$

We want to find a measure of "goodness" (*information gain*) for each attribute chosen as a root of the current tree. This could be the total information in the tree minus the amount of information needed to complete the tree after choosing that attribute as a root. The amount of information needed to complete the tree is defined as the weighted average of the information in all its subtrees. The weighted average is computed by multiplying the information content of each subtree by the percentage of the examples present in that subtree and summing these products.

Assume that making attribute A, with n values, the root of the current tree, will partition the set of training examples E into subsets $E_1, ..., E_n$. Then, the information needed to complete that tree after making A the root is:

$$R(A) = \sum_{i=1}^{n} \frac{|E_i|}{|E|} I(E_i)$$

Then, the information gain of choosing attribute A is:

$$gain(A) = I(E) - R(A)$$

For the above example we can calculate the gain of choosing attributes *color* and *shape*. For *color* we have two subsets $C_1 = \{1, 2, 3\}$ and $C_2 = \{4, 5, 6\}$.

$$I(C_1) = -\frac{1}{3} log_2(\frac{1}{3}) - \frac{2}{3} log_2(\frac{2}{3}) = 0.5383 + 0.3840 = 0.9723$$

$$I(C_2) = -\frac{0}{3} log_2(\frac{0}{3}) - \frac{3}{3} log_2(\frac{3}{3}) = 0$$

$$gain(color) = I(E) - R(color) = 0.65 - (\frac{3}{6}0.9723 + \frac{3}{6}0) = 0.1638$$

For *shape* we have three subsets $S_1 = \{1, 6\}$, $S_2 = \{2, 5\}$ and $S_3 = \{3, 4\}$.

$$I(S_1) = -\frac{1}{2} log_2(\frac{1}{2}) - \frac{1}{2} log_2(\frac{1}{2}) = 1$$

Clearly $I(S_2) = 0$, and $I(S_3) = 0$, since they contain only one-class instances. Then

$$gain(shape) = I(E) - R(shape) = 0.65 - (\frac{2}{6}1 + 0 + 0) = 0.3166$$

Because *shape* provides grater information gain the algorithm will select it first to partition the tree (as it was shown in the examples above).

5.4 Learning multiple concepts

The algorithm described in Section 2 actually builds decision trees for classification of the training instances into two classes (YES – belonging to the concept, and NO – not belonging to the concept). This algorithm can be easily generalized to handle more than two classes (concepts) as follows:

1. If all instances in E belong to a single class, then create a node marked with the class name and halt. Otherwise, select an attribute A (e.g. using the information gain heuristic), with values $V_1...V_n$ and create the decision node

```
        A
        |
  -------------
 |   |  |      |
 |   |  |      |
V1  V2  V3 ... Vn
```

2. Partition the training instances in E into subsets $E_1, E_2, ..., E_n$, according to the values of $A(\{V_1, V_2, ..., V_n\})$

3. Apply the algorithm recursively to each of the sets E_1, E_2 etc.

5.5 Learning from noisy data

In many situations the training data are imperfect. For example, the attribute or class values for some instances could be incorrect, because of errors. We call such data *noisy data*. In case of noise we usually abandon the requirement the hypothesis to cover all positive and none of the negative examples. So, we allow the learning system to misclassify some instances and we hope that the misclassified instances are those that contain errors.

Inducing decision trees from nosy data will cause basically two problems: first, the trees misclassify new data, and second, the trees tend to become very large and thus hard to understand and difficult to use.

Assume the following situation. At some step of the algorithm we have chosen an attribute A partitioning the current set S of 100 training instances into two classes – C_1 and C_2, where C_1 contains 99 instances and C_2 - one instance. Knowing that there is a noise in the training data, we can assume that all instances from S belong to class C_1. In this way we force the algorithm to stop further exploring the decision tree, i.e. we prune the subtree rooted at A. This technique is called *forward pruning*. There is another kind of pruning, called *post-pruning*, where first the whole tree is explored completely, and then the subtrees are estimated on their reliability with respect to possible errors. Then those of them with low estimates are pruned. Both techniques for pruning are based on probability estimates of the classification error in each node of the tree.

Chapter 6

Covering strategies

6.1 Basic idea

The covering strategy is used for searching the hypothesis space for *disjunctive hypotehses* – hypotheses consisting of more than one component (e.g. a set of attribute-value pairs, a propositional or relational rule). Each of this components covers a subset of the set of examples and all the components jointly (the whole hypothesis) cover the whole set of examples. The basic covering algorithm consists of three steps:

1. Applying some induction technique (e.g. a generalization opertor) to infer a correct component of the hypothsis (e.g. a rule or a clause), that covers a subset (possibly maximal) of the positive examples.
2. Excluding the covered examples from the set of positive examples.
3. Repeating the above two steps until the set of positive examples becomes empty.

The final hypothesis is a dijunction of all components found by the above procedure. Step 1 of this algorithm is usualy based on searching the hypothesis space where hypotheses that are correct (not covering negative examples) and covering more positive examples are prefered.

6.2 Lgg-based propositional induction

In the attribute-value language as well as in the language of first order atomic formulas the examples and the hypotheses have the same representation. This allows the generalization operators (e.g. the lgg) to be applied on examples and hypotheses at the same time, which in turn simplifies the learning algorithms.

Consider a learning problem where a set of examples E, belonging to k different classes is given, i.e. $E = \cup_{i=1}^{k} E_i$. The following algortihm finds a set of hypotheses jointly covering E.

1. Select (randomly or using some criterion) two examples/hypotheses from the same class, say k, i.e. $e_i, e_j \in E_k$.
2. Find $h_{ij}^k = lgg(e_i, e_j)$.
3. If h_{ij}^k does not cover examples/hypotheses from classes other than k, then add h_{ij}^k to E_k and remove from E_k the elements covered by h_{ij}^k (these are at least e_i and e_j). Otherwise go to step 1.

4. The algorithm terminates when step 1 or 3 is impossible to acomplish. Then the set E contains the target hypotesis.

To illustrate the above algorithm let us consider the following set of examples (instances of animals):

```
1, mammal,   [has_covering=hair,milk=t,homeothermic=t,habitat=land,eggs=f,gills=f]
2, mammal,   [has_covering=none,milk=t,homeothermic=t,habitat=sea,eggs=f,gills=f]
3, mammal,   [has_covering=hair,milk=t,homeothermic=t,habitat=sea,eggs=t,gills=f]
4, mammal,   [has_covering=hair,milk=t,homeothermic=t,habitat=air,eggs=f,gills=f]
5, fish,     [has_covering=scales,milk=f,homeothermic=f,habitat=sea,eggs=t,gills=t]
6, reptile,  [has_covering=scales,milk=f,homeothermic=f,habitat=land,eggs=t,gills=f]
7, reptile,  [has_covering=scales,milk=f,homeothermic=f,habitat=sea,eggs=t,gills=f]
8, bird,     [has_covering=feathers,milk=f,homeothermic=t,habitat=air,eggs=t,gills=f]
9, bird,     [has_covering=feathers,milk=f,homeothermic=t,habitat=land,eggs=t,gills=f]
10,amphibian,[has_covering=none,milk=f,homeothermic=f,habitat=land,eggs=t,gills=f]
```

After termination of the algorithm the above set is transformed into the following one (the hypothesis ID's show the way they are generated, where "+" means lgg):

```
8+9, bird, [has_covering=feathers,milk=f,homeothermic=t,eggs=t,gills=f]
6+7, reptile, [has_covering=scales,milk=f,homeothermic=f,eggs=t,gills=f]
4+(3+(1+2)), mammal, [milk=t,homeothermic=t,gills=f]
5, fish, [has_covering=scales,milk=f,homeothermic=f,habitat=sea,eggs=t,gills=t]
10, amphibian, [has_covering=none,milk=f,homeothermic=f,habitat=land,eggs=t,gills=f]
```

A drawback of the lgg-based covering algorithm is that the hypotheses depend on the order of the examples. The generality of the hypotheses depend on the similarity of the examples (how many attribute-value pairs they have in common) that are selected to produce an lgg. To avoid this some criteria for selection of the lgg candidate pairs can be applied. For example, such a criterion may be the maximum similarity between the examples in the pair.

Another problem may occur when the examples from a single class are all very similar (or very few, as in the animals example above). Then the generated hypothesis may be too specific, although more general hypotheses that correctly separate the classes may exists. In fact, this is a general problem with the covering strategies, which is avoided in the separate and conquer aproaches (as desicion trees).

Lgg-based relational induction

θ-subsumption. Given two clauses C and D, we say that C *subsumes* D (or C is a *generalization* of D), if there is a substitution θ, such that $C\theta \subseteq D$. For example,

```
parent(X,Y):-son(Y,X)
```

θ-subsumes ($\theta = \{X/john, Y/bob\}$)

```
parent(john,bob):- son(bob,john),male(john)
```

because
$\{parent(X,Y), \neg son(Y,X)\}\theta \subseteq \{parent(john,bob), \neg son(bob,john), \neg male(john)\}$.

The θ-subsumption relation can be used to define an *lgg* of two clauses.
***lgg* under θ-subsumption (*lggθ*)**. The clause C is an *lggθ* of the clauses C_1 and C_2 if C θ-subsumes C_1 and C_2, and for any other clause D, which θ-subsumes C_1 and C_2, D also θ-subsumes C. Here is an example:
$C_1 = parent(john, peter) : -son(peter, john), male(john)$
$C_2 = parent(mary, john) : -son(john, mary)$

$lgg(C_1, C_2) = parent(A, B) : -son(B, A)$

The *lgg* under θ-subsumption can be calculated by using the *lgg* on terms. $lgg(C_1, C_2)$ can be found by collecting all *lgg*'s of one literal from C_1 and one literal from C_2. Thus we have

$$lgg(C_1, C_2) = \{L | L = lgg(L_1, L_2), L_1 \in C_1, L_2 \in C_2\}$$

Note that we have to include in the result *all* literals L, because any clause even with one literal L will θ-subsume C_1 and C_2, however it will not be the least general one, i.e. an *lgg*.

When background knowledge BK is used a special form of *relative lgg* (or rlgg) can be defined on atoms. Assume BK is a set of facts, and A and B are facts too (i.e. clauses without negative literals). Then

$$rlgg(A, B, BK) = lgg(A : -BK, B : -BK)$$

The relative lgg (rlgg) can be used to implement an inductive learning algorithm that induces Horn clauses given examples and background knowledge as first order atoms (facts). Below we illustrate this algorithm with an example.

Consider the following set of facts (desribing a directed acyclic graph): $BK = \{link(1, 2), link(2, 3), link(3, 4), link(3, 5)\}$, positive examples $E^+ = \{path(1, 2), path(3, 4), path(2, 4), path(1, 3)\}$ and negative examples E^- – the set of all instances of $path(X, Y)$, such that there is not path between X and Y in BK. Let us now apply an rlgg-based version of the covering algorithm desribed in the previous section:

1. Select the first two positive examples $path(1, 2)$, $path(3, 4)$ and find their *rlgg*, i.e. the *lgg* of the following two clauses (note that the bodies of these clauses include also all positive examples, because they are part of BK):
   $path(1, 2) : -link(1, 2), link(2, 3), link(3, 4), link(3, 5),$
   $\quad path(1, 2), path(3, 4), path(2, 4), path(1, 3)$
   $path(3, 4) : -link(1, 2), link(2, 3), link(3, 4), link(3, 5),$
   $\quad path(1, 2), path(3, 4), path(2, 4), path(1, 3)$
   According to the above-mentioned algorithm this is the clause:
   $path(A, B) : -path(1, 3), path(C, D), path(A, D), path(C, 3),$
   $\quad path(E, F), path(2, 4), path(G, 4), path(2, F), path(H, F), path(I, 4),$
   $\quad path(3, 4), path(I, F), path(E, 3), path(2, D), path(G, D), path(2, 3),$
   $\quad link(3, 5), link(3, \_), link(I, \_), link(H, \_), link(3, \_), link(3, 4),$
   $\quad link(I, F), link(H, \_), link(G, \_), link(G, D), link(2, 3), link(E, I),$
   $\quad link(A, \_), link(A, B), link(C, G), link(1, 2).$

2. Here we perform an additional step, called *reduction*, to simplify the above clause. For this purpose we remove from the clause body:

   - all ground literals;
   - all literals that are not connected with the clause head (none of the head variables A and B appears in them);
   - all literals that make the clause *tautology* (a clause that is always true), i.e. body literals same as the clause head;
   - all literals that when removed do not reduce the clause coverage of positive examples and do not make the clause incorrect (covering negative examples).

   After the reduction step the clause is $path(A, B) : -link(A, B)$.

3. Now we remove from E^+ the examples that the above clause covers and then $E^+ = \{path(2,4), path(1,3)\}$.

4. Since E^+ is not empty, we further select two examples ($path(2,4)$, $path(1,3)$) and find their *rlgg*, i.e. the lgg of the following two clauses:
   $path(2,4) : -link(1,2), link(2,3), link(3,4), link(3,5),$
   $path(1,2), path(3,4), path(2,4), path(1,3)$
   $path(1,3) : -link(1,2), link(2,3), link(3,4), link(3,5),$
   $path(1,2), path(3,4), path(2,4), path(1,3)$
   which is:
   $path(A,B) : -path(1,3), path(C,D), path(E,D), path(C,3),$
   $path(A,B), path(2,4), path(F,4), path(2,B), path(G,B), path(H,4),$
   $path(3,4), path(H,B), path(A,3), path(2,D), path(F,D), path(2,3),$
   $link(3,5), link(3,\_), link(H,\_), link(G,\_), link(3,\_), link(3,4),$
   $link(H,B), link(G,\_), link(F,\_), link(F,D), link(2,3), link(A,H),$
   $link(E,\_), link(C,F), link(1,2).$
   After reduction we get $path(A,B) : -link(A,H), link(H,B)$.

The last two clauses form the sandard definition of a procedure to find a path in a graph.

Chapter 7

Searching the generalization/specialization graph

7.1 Basic idea

Given a constructive operator for generalization/specialization, for each example $e \in E^+$ we can build a directed graph (hierarchy) of hypotheses with two terminal nodes – the most specific element e, and the most general hypothesis $\top$. This graph will also include all correct hypotheses (not covering negative examples) that cover at least one positive example (e). And, as usual we will be looking for hypotheses covering more positive examples, i.e. maximally general ones. As this graph is a strict hierarchical structure, for each hypothesis h, the length of the path between h and e or h and $\top$ can play the role of a measure for the generality/specificity of h. Thus some standard graph search strategies as depth-first, breadth-first or hill-climbing can be applied. Depending on the starting point of the search we may have two basic search approaches:

- Specific to general search: starting from e and climbing up the hierarchy (applying generalization operators), i.e. searching among the correct generalizations of e for the one with maximal coverage.
- General to specific search: starting from $\top$ and climbing down the hierarchy (applying specialization operators), i.e. searching among the incorrect hypothesis which at the next specialization step can produce a correct hypothesis with maximal coverage.

The above discussed search procedures are usually embeded as step one in a covering learning algortihm. To illustrate this in the following two section we discuss two examples – searching the space of propositional hypothesis and heuristic general to specific search for relational hypotheses.

7.2 Searching the space of propositional hypotheses

Consder the MONKS concept discussed in chapter 3. Let us select a positive example, say $e = [octagon, octagon, no, sword, red, yes]$ (example 1 from Figure 3.1, where the attribute names are omitted for brevity). The most general hypothesis in this language is $\top = [\_, \_, \_, \_, \_, \_]$

(underscores denote "don't care" or any value). The first level of the specialization hierarchy starting from $\top$ is:

```
[octagon,_,_,_,_,_]
[_,octagon,_,_,_,_]
[_,_,no,_,_,_]
[_,_,_,sword,_,_]
[_,_,_,_,red,_]
[_,_,_,_,_,yes]
```

Note that the values that fill the don't care slots are taken from e, i.e. the choice of e determines completely the whole generaliztion/specialization hierarchy. Thus the bottom level of the hierarchy is:

```
[_,octagon,no,sword,red,yes]
[octagon,_,no,sword,red,yes]
[octagon,octagon,_,sword,red,yes]
[octagon,octagon,no,_,red,yes]
[octagon,octagon,no,sword,_,yes]
[octagon,octagon,no,sword,red,_]
```

A part of the generalization/specialization hierarchy for the above example is shown in Figure 7.1. This figure also illustrates two search algorithms, both version of depth-first search with evaluation function (hill climbing). The top-down (general to specific) search uses the hypothesis accuracy $A(h_k)$ for evaluation. It is defined as

$$A(h^k) = \frac{|\{e|e \in E^k, h^k \geq e\}|}{|\{e|e \in E, h^k \geq e\}|},$$

i.e. the number of examples from class k that the hypothesis covers over the total number of examples covered. The bottom-up search uses just the total number of examples covered, because all hypotheses are 100% correct, i.e. $A(h_k) = 1$ for all k. The bottom-up search stops when all possible specializations of a particular hypothesis lead to incorrect hypotheses. The top-down search stops when a correct hypothesis is found, i.e. the maximum value of the evaluation function is reached. The figure shows that both search algorithms stop at the same hypothesis – $[octagon, octagon, \_, \_, \_, \_]$, which is, if fact, a part of the target concept (as shown in Section 3.1)

The above described process to find the hypothesis $h = [octagon, octagon, \_, \_, \_, \_]$ is just one step in the overall covering algorithm, where the examples covered by h are then removed from E and the same process is repeated until E becomes empty.

7.3 Searching the space of relational hypotheses

In this section we shall discuss a basic algorithm for learning Horn clauses from examples (ground facts), based on general to specific search embeded in a covering strategy. At each pass of the outermost loop of the algorithm a new clause is generated by θ-subsumption specialization of the most general hypothesis $\top$. Then the examples covered by this clause are removed and the process continues until no uncovered exampes are left. The negative examples are used in the inner loop that finds individual clauses to determine when the current clause needs further specialization. Two types of specialization operators are applied:

1. Replacing a variable with a term.
2. Adding a literal to the clause body.

Figure 7.1: Generalization specialization graph for MONKS data

These operators are minimal with respect to θ-subsumption and thus they ensure an exhaustive search in the θ-subsumption hierarchy.

There are two stopping conditions for the inner loop (terminal nodes in the hierarchy):

- Correct clauses, i.e. clauses covering at least one positive example and no negative examples. These are used as components of the final hypothesis.
- Clauses not covering any positive examples. These are just omitted.

Let us consider an illustration of the above algorithm. The target predicate is $member(X, L)$ (returning true when X is a member of the list L). The examples are
$E^+ = \{member(a, [a, b]), member(b, [b]), member(b, [a, b])\}$,
$E^- = \{member(x, [a, b])\}$.
The most general hypothesis is $\top = member(X, L)$. A part of the generalization/specialization graph is shown in Figure 7.2. The terminal nodes of this graph:
$member(X, [X|Z])$
$member(X, [Y|Z]) : -member(X, Z)$
are correct clauses and jointly cover all positive examples. So, the goal of the algorithm is to reach these leaves.

A key issue in the above algorithm is the search stategy. A possible approach to this is the so called iterative deepening, where the graph is searched iteratively at depths 1, 2, 3,..., etc. until no more specializations are needed. Another appraoch is a depth-first search with an evaluation function (hill climbing). This is the approach taken in the popular system FOIL that is briefly described in the next section.

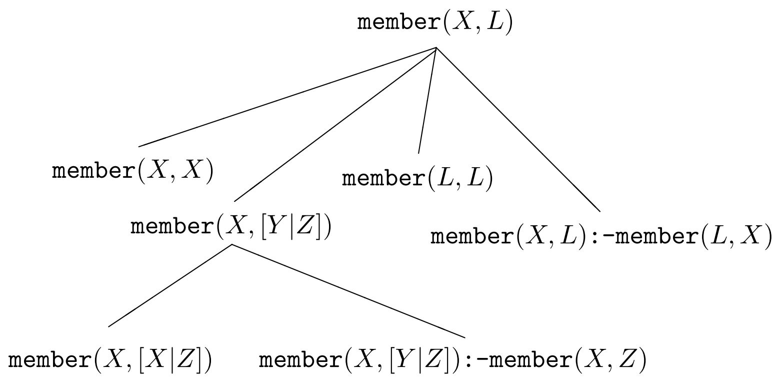


Figure 7.2: A generalization/specialization graph for $member(X, L)$

7.4 Heuristic search – FOIL

7.4.1 Setting of the problem

Consider the simple relational domain also discussed in Section 6.3 – the *link* and *path* relations in a directed acyclic graph. The background knowledge and the positive examples are:

$BK = \{link(1,2), link(2,3), link(3,4), link(3,5)\}$
$E^+ = \{path(1,2), path(1,3), path(1,4), path(1,5),$
$path(2,3), path(2,4), path(2,5), path(3,4), path(3,5)\}$

The negative examples can be specified explicitly. If we assume however, that our domain is closed (as the particular *link and path* domain) the negative examples can be generated automatically using the *Closed World Assumption (CWA)*. In our case these are all ground instances of the *path* predicate with arguments – constants from E^+. Thus

$E^- = \{path(1,1), path(2,1), path(2,2), path(3,1), path(3,2), path(3,3),$
$path(4,1), path(4,2), path(4,3), path(4,4), path(4,5), path(5,1), path(5,2),$
$path(5,3), path(5,4), path(5,5)\}$

The problem is to find a hypothesis H, i.e. a Prolog definition of *path*, which satisfies the *necessity* and *strong consistency* requirements of the induction task (see Chapter 2). In other words we require that $BK \wedge H \vdash E^+$ and $BK \wedge H \not\vdash E^-$. To check these condition we use *logical consequence* (called also *cover*).

7.4.2 Illustrative example

We start from the *most general* hypothesis

$$H_1 = path(X, Y)$$

Obviously this hypothesis covers all positive examples E^+, however many negative ones too. Therefore we have to specialize it by adding body literals. Thus the next hypothesis is

$$H_2 = path(X, Y) : -L.$$

The problem now is to find a proper literal L. Possible candidates are literals containing only variables with predicate symbols and number of arguments taken from the set E^+, i.e. $L \in \{link(V_1, V_2), path(V_1, V_2)\}$.

Clearly if the variables V_1, V_2 are both different from the head variables X and Y, the new clause H_2 will not be more specific, i.e. it will cover the same set of negatives as H_1. Therefore we impose a restriction on the choice of variables, based on the notion of *old variables*. Old variables are those appearing in the previous clause. In our case X and Y are old variables. So, we require *at least one* of of V_1 and V_2 to be an old variable.

Further, we need a criterion to choose the *best literal* L. The system described here, FOIL uses an *information gain measure* based on the ratio between the number of positive and negative examples covered. Actually, each newly added literal has *to decrease the number of covered negatives maximizing at the same time the number of uncovered positives.* Using this criterion it may be shown that the best candidate is $L = link(X, Y)$. That is

$$H_2 = path(X, Y) : -link(X, Y)$$

This hypothesis does not cover any negative examples, hence we can stop further specialization of the clause. However there are still uncovered positive examples. So, we save H_2 as a part of the final hypothesis and continue the search for a new clause.

To find the next clause belonging to the hypothesis we exclude the positive examples covered by H_2 and apply the same algorithm for building a clause using the rest of positive examples. This leads to the clause $path(X, Y) : -link(X, Z), path(Y, Z)$, which covers these examples and is also correct. Thus the final hypothesis H_3 is the usual definition of path:

```
path(X,Y):-link(X,Y).
path(X,Y):-link(X,Z),path(Z,Y).
```

7.4.3 Algorithm FOIL

An algorithm based on the above ideas is implemented in the system called FOIL (First Order Inductive Learning) [2]. Generally the algorithm consists of two nested loops. The inner loop constructs a clause and the outer one adds the clause to the predicate definition and calls the inner loop with the positive examples still uncovered by the current predicate.

The algorithm has several critical points, which are important for its efficiency and also can be explored for further improvements: +

- The algorithm performs a search strategy by choosing the *locally best branch* in the search tree and further exploring it *without backtracking.* This actually is a *hill climbing* strategy which may drive the search in a local maximum and prevent it from finding the best global solution. Particularly, this means that in the inner loop there might be a situation when there are still uncovered negative examples and there is no proper literal literal to be added. In such a situation we can allow the algorithm to add a new literal without requiring an increase of the information gain and then to proceed in the usual way. This means to force a further step in the search tree hopping to escape from the local maximum. This further step however *should not lead to decrease of the information gain* and also *should not complicate the search space* (increase the branching). Both requirements are met if we choose *determinate literals* (see Chapter 8) for this purpose.

  Using determinate literals however does not guarantee that the best solution can be found. Furthermore, this can complicate the clauses without actually improving the hypothesis with respect to the *sufficiency* and *strong consistency.*

- When dealing with *noise* the *strong consistency* condition can be weakened by allowing the inner loop to terminate even when the current clause covers some of the negative examples. In other words these examples are considered as noise.

- If the set of positive examples is *incomplete*, then CWA will add the missing positive examples to the set of negative ones. Then if we require strong consistency, the constructed hypothesis will be specialized to exclude the examples, which actually we want to generalize. A proper *stopping condition* for the inner loop would cope with this too.

Chapter 8

Inductive Logic Programming

8.1 ILP task

Generally *Inductive Logic Programming (ILP)* is an area integrating Machine Learning and Logic Programming. In particular this is a version of the induction problem (see Chapter 2), where all languages are subsets of Horn clause logic or Prolog.

The setting for ILP is as follows. B and H are logic programs, and E^+ and E^- – usually sets of ground facts. The conditions for construction of H are:

- *Necessity:* $B \not\vdash E^+$
- *Sufficiency:* $B \wedge H \vdash E^+$
- *Weak consistency:* $B \wedge H \not\vdash []$
- *Strong consistency:* $B \wedge H \wedge E^- \not\vdash []$

The strong consistency is not always required, especially for systems which deal with noise. The necessity and consistency condition can be checked by a theorem prover (e.g. a Prolog interpreter). Further, applying *Deduction theorem* to the sufficiency condition we can transform it into

$$B \wedge \neg E^+ \vdash \neg H \tag{8.1}$$

This condition actually allows to infer *deductively* the hypothesis from the background knowledge and the examples. In most of the cases however, the number of hypotheses satisfying (1) is too large. In order to limit this number and to find only useful hypotheses some additional criteria should be used, such as:

- *Extralogical restrictions* on the background knowledge and the hypothesis language.
- *Generality* of the hypothesis. The simplest hypothesis is just E^+. However, it is too specific and hardly can be seen as a generalization of the examples.
- *Decidability and tractability* of the hypothesis. Extending the background knowledge with the hypothesis should not make the resulting program indecidable or intractable, though logically correct. The point here is that such hypotheses cannot be tested for validity (applying the sufficiency and consistency conditions). Furthermore the aim of ILP is to construct real working logic programs, rather than just elegant logical formulae.

In other words condition (1) can be used to generate a number of initial approximations of the searched hypothesis, or to evaluate the correctness of a currently generated hypothesis. Thus the problem of ILP comes to *construction of correct hypotheses and moving in the space of possible hypotheses* (e.g. by generalization or specialization). For this purpose a number of techniques and algorithms are developed.

8.2 Ordering Horn clauses

A logic program can be viewed in two ways: as a *set of clauses* (implicitly conjoined), where each clause is a *set of literals* (implicitly disjoined), and as a logical formula in *conjunctive normal form* (conjunction of disjunction of literals). The first interpretation allows us to define a clause ordering based on the subset operation, called θ - *subsumption.*

8.2.1 θ-subsumption

θ-subsumption. Given two clauses C and D, we say that C *subsumes* D (or C is a *generalization* of D), iff there is a substitution θ, such that $C\theta \subseteq D$.

For example,

```
parent(X,Y):-son(Y,X)
```

θ-subsumes ($\theta = \{X/john, Y/bob\}$)

```
parent(john,bob):- son(bob,john),male(john)
```

since

$\{parent(X,Y), \neg son(Y,X)\}\theta \subseteq \{parent(john,bob), \neg son(bob,john), \neg male(john)\}$.

θ-subsumption can be used to define an *lgg* of two clauses.

***lgg* under θ-subsumption ($lgg\theta$)**. The clause C is an $lgg\theta$ of the clauses C_1 and C_2 iff C θ-subsumes C_1 and C_2, and for any other clause D, which θ-subsumes C_1 and C_2, D also θ-subsumes C.

Consider for example the clauses $C_1 = p(a) \leftarrow q(a), q(f(a))$ and $C_2 = p(b) \leftarrow q(f(b))$. The clause $C = p(X) \leftarrow a(f(X))$ is an $lgg\theta$ of C_1 and C_2.

The *lgg* under θ-subsumption can be calculated by using the *lgg* on terms. Consider clauses C_1 and C_2. $lgg(C_1, C_2)$ can be found by collecting all *lgg*'s of one literal from C_1 and one literal from C_2. Thus we have

$$lgg(C_1, C_2) = \{L | L = lgg(L_1, L_2), L_1 \in C_1, L_2 \in C_2\}$$

Note that we have to include in the result *all* such literals L, because any clause even with one literal L will θ-subsume C_1 and C_2, however it will not be the least general one, i.e. an *lgg*.

8.2.2 Subsumption under implication

When viewing clauses as logical formulae we can define another type of ordering using *logical consequence (implication).*

Subsumption under implication. The clause C_1 is *more general* than clause C_2, (C_1 *subsumes under implication* C_2), iff $C_1 \vdash C_2$. For example, $(P : -Q)$ is more general than $(P : -Q, R)$, since $(P : -Q) \vdash (P : -Q, R)$.

The above definition can be further extended by involving a *theory* (a logic program).

Subsumption relative to a theory. We say that C_1 subsumes C_2 w.r.t. theory T, iff $P \wedge C_1 \vdash C_2$.

For example, consider the clause:

```
cuddly_pet(X) :- small(X), fluffy(X), pet(X)        (C)
```

and the theory:

```
pet(X) :- cat(X)                                    (T)
pet(X) :- dog(X)
small(X) :- cat(X)
```

Then C is more general than the following two clauses w.r.t. T:

```
cuddly_pet(X) :- small(X), fluffy(X), dog(X)        (C1)
cuddly_pet(X) :- fluffy(X), cat(X)                  (C2)
```

Similarly to the terms, the ordering among clauses defines a lattice and clearly the most interesting question is to find the *least general generalization* of two clauses. It is defined as follows. $C = lgg(C_1, C_2)$, iff $C \geq C_1$, $C \geq C_2$, and any other clause, which subsumes both C_1 and C_2, subsumes also C. If we use a relative subsumption we can define a *relative least general generalization (rlgg).*

The subsumption under implication can be tested using *Herbrand's theorem.* It says that $F_1 \vdash F_2$, iff for every substitution σ, $(F_1 \wedge \neg F_2)\sigma$ is false ([]). Practically this can be done in the following way. Let F be a clause or a conjunction of clauses (a theory), and $C = A : -B_1, ..., B_n$ - a clause. We want to test whether $F \wedge \neg C$ is always false for any substitution. We can check that by skolemizing C, adding its body literals as facts to F and testing whether A follows from the obtained formula. That is, $F \wedge \neg C \vdash []$ is equivalent to $F \wedge \neg A \wedge B_1 \wedge ... \wedge B_n \vdash []$, which in turn is equivalent to $F \wedge B_1 \wedge ... \wedge B_n \vdash A$. The latter can be checked easily by Prolog resolution, since A is a ground literal (goal) and $F \wedge B_1 \wedge ... \wedge B_n$ is a logic program.

8.2.3 Relation between θ-subsumption and subsumption under implication

Let C and D be clauses. Clearly, if $C\theta$-subsumes D, then $C \vdash D$ (this can be shown by the fact that all models of C are also models of D, because D has just more disjuncts than C). However, the opposite is not true, i.e. from $C \vdash D$ does not follow that C θ-subsumes D. The latter can be shown by the following example.

Let $C = p(X) \leftarrow q(f(X))$ and $D = p(X) \leftarrow q(f(f(X)))$. Then $C \vdash D$, however C does not θ-subsume D.

8.3 Inverse Resolution

A more constructive way of dealing with clause ordering is by using *the resolution principle.* The idea is that the resolvent of two clauses is subsumed by their conjunction. For example, $(P \vee \neg Q \vee \neg R) \wedge Q)$ is more general than $P \vee \neg R$, since $(P \vee \neg Q \vee \neg R) \wedge Q) \vdash (P \vee \neg R)$. The clauses C_1 and C_2 from the above example are resolvents of C and clauses from T.

The resolution principle is an effective way of deriving logical consequences, i.e. *specializations.* However when building hypothesis we often need an algorithm for inferring *generalizations* of clauses. So, this could be done by an inverted resolution procedure. This idea is discussed in the next section.

Consider two clauses C_1 and C_2 and its resolvent C. Assume that the resolved literal appears positive in C_1 and negative in C_2. The three clauses can be drawn at the edges of a "V" – C_1 and C_2 at the arms and C – at the base of the "V".

```
C1    C2
 \    /
  \  /
   \/
   C
```

A resolution step derives the clause at the base of the "V", given the two clauses of the arms. In the ILP framework we are interested to infer the clauses at the arms, given the clause at the base. Such an operation is called *"V" operator*. There are two possibilities.

A "V" operator which given C_1 and C constructs C_2 is called *absorption*. The construction of C_1 from C_2 and C is called *identification*.

The "V" operator can be derived from the equation of resolution:

$$C = (C_1 - \{L_1\})\theta_1 \cup (C_2 - \{L_2\})\theta_2$$

where L_1 is a positive literal in C_1, L_2 is a negative literal in C_2 and $\theta_1\theta_2$ is the *mgu* of $\neg L_1$ and L_2.

Let $C = C_1' \cup C_2'$, where $C_1' = (C_1 - \{L_1\})\theta_1$ and $C_2' = (C_2 - \{L_2\})\theta_2$. Also let $D = C_1' - C_2'$. Thus $C_2' = C - D$, or $(C_2 - \{L_2\})\theta_2 = C - D$. Hence:

$$C_2 = (C - D)\theta_2^{-1} \cup \{L_2\}$$

Since $\theta_1\theta_2$ is the *mgu* of $\neg L_1$ and L_2, we get $L_2 = \neg L_1\theta_1\theta_2^{-1}$. By θ_2^{-1} we denote an *inverse substitution*. It replaces terms with variables and uses *places* to select the term arguments to be replaced by variables. The places are defined as n-tuples of natural numbers as follows. The term at place <i> within $f(t_0, .., t_m)$ is t_i and the term at place $<i_0, i_1, .., i_n>$ within $f(t_0, .., t_m)$ is the term at place $<i_1, .., i_n>$ within t_{i_0}. For example, let $E = f(a, b, g(a, b))$, $Q = f(A, B, g(C, D))$. Then $Q\sigma = E$, where $\sigma = \{A/a, B/b, C/a, D/b\}$. The inverse substitution of σ is $\sigma^{-1} = \{<a,<0>>/A, <b,<1>>/B, <a,<2,0>>/C, <b,<2,1>>\ /D\}$. Thus $E\sigma^{-1} = Q$. Clearly $\sigma\sigma^{-1} = \{\}$.

Further, substituting L_2 into the above equation we get

$$C_2 = ((C - D) \cup \{\neg L_1\}\theta_1)\theta_2^{-1}$$

The choice of L_1 is unique, because as a positive literal, L_1 is the head of C_1. However the above equation is still not well defined. Depending on the choice of D it give a whole range of solutions, i.e. $\oslash \cap D \cap C_1'$. Since we need the *most specific* C_2, D should be $\oslash$. Then we have

$$C_2 = (C \cup \{\neg L_1\}\theta_1)\theta_2^{-1}$$

Further we have to determine θ_1 and θ_2^{-1}. Again, the choice of most specific solution gives that θ_2^{-1} has to be empty. Thus finally we get the *most specific solution of the absorption operation* as follows:

$$C_2 = C \cup \{\neg L_1\}\theta_1$$

The substitution θ_1 can be partly determined from C and C_1. From the resolution equation we can see that $C_1 - \{L_1\})$ θ-subsumes C with θ_1. Thus a part of θ_1 can be constructed by matching literals from C_1 and C, correspondingly. However for the rest of θ there is a free choice, since θ_1 is a part of the *mgu* $\neg L_1$ and L_2 and L_2 is unknown. This problem can be avoided by assuming that every variable within L_1 also appear in C_1. In this case θ can be fully determined by matching all literals within $(C_1 - \{L_1\})$ with literals in C. Actually this is a constraint that all variables in a head (L_1) of a clause (C_1) have to be found in its body ($C_1 - \{L_1\}$). Such clauses are called *generative* clauses and are often used in the ILP systems.

For example, given the following two clauses

```
mother(A,B) :- sex(A,female),daughter(B,A)                     (C1)
grandfather(a,c) :- father(a,m),sex(m,female),daughter(c,m)    (C )
```

the absorption "V" operator as derived above will construct

```
grandfather(a,c) :- mother(m,c),father(a,m),
                    sex(m,female),daughter(c,m)                (C2)
```

Note how the substitution θ_1 was found. This was done by unifying a literal from `C` – `daughter(c,m)` with a literal from `C1` – `daughter(B,A)`. Thus $\theta_1 = \{A/m, B/c\}$ and $L_1\theta_1 =$ `mother(m,c)`. (The clause `C1` is generative.)

The clause `C2` can be reduced by removing the literals `sex(m,female)` and `daughter(c,m)`. This can be done since these two literals are redundant (`C2` without them resolved with `C1` will give the same result, `C`). Thus the result of the absorption "V" operator is finally

```
grandfather(a,c) :- mother(m,c),father(a,m)                    (C2)
```

8.4 Predicate Invention

By combining two resolution V's back-to-back we get a *"W" operator*.

```
C1    A     C2
 \    /\    /
  \  /  \  /
   \/    \/
   B1    B2
```

Assume that C_1 and C_2 resolve on a common literal L in A and produce B_1 and B_2 respectively. The "W" operator constructs A, C_1 and C_2, given B_1 and B_2. It is important to note that the literal L does not appear in B_1 and B_2. So, the "W" operator has to introduce a *new predicate symbol*. In this sense this predicate is *invented* by the "W" operator.

The literal L can appear as negative or as positive in A. Consequently there are to types of "W" operators - *intra-construction* and *inter-construction* correspondingly.

Consider the two resolution equations involved in the "W" operator.

$$B_i = (A - \{L\})\theta_{A_i} \cup (C_i - \{L_i\})\theta_{C_i}$$

where $i \in \{1, 2\}$, L is negative in A, and positive in C_i, and $\theta_{A_i}\theta_{C_i}$ is the *mgu* of $\neg L$ and L_i. Thus $(A - \{L\})$ θ-subsumes each clause B_i, which in turn gives one possible solution $(A - \{L\}) = lgg(B_1, B_2)$, i.e.

$$A = lgg(B_1, B_2) \cup \{L\}$$

Then θ_{A_i} can be constructed by matching $(A - \{L\})$ with literals of B_i.

Then substituting A in the resolution equation and assuming that θ_{C_i} is empty (similarly to the "V" operator) we get

$$C_i = (B_i - lgg(B_1, B_2)\theta_{A_i}) \cup \{L_i\}$$

Since $L_i = \neg L\theta_{A_i}\theta_{C_i}^{-1}$, we obtain finally

$$C_i = (B_i - lgg(B_1, B_2)\theta_{A_i}) \cup \{\neg L\}\theta_{A_i}$$

For example the intra-construction "W" operator given the clauses

```
grandfather(X,Y) :- father(X,Z), mother(Z,Y)            (B1)
grandfather(A,B) :- father(A,C), father(C,B)            (B2)
```

constructs the following three clauses (the arms of the "W").

```
p1(_1,_2) :- mother(_1,_2)                              (C1)
p1(_3,_4) :- father(_3,_4)                              (C2)
grandfather(_5,_6) :- p1(_7,_6), father(_5,_7)          (A )
```

The "invented" predicate here is `p1`, which obviously has the meaning of "parent".

8.5 Extralogical restrictions

The background knowledge is often restricted to *ground facts*. This simplifies substantially all the operations discussed so far. Furthermore, this allows all ground hypotheses to be derived directly, i.e. in that case $B \wedge \neg E^+$ is a set of positive and negative literals.

The hypotheses satisfying all logical conditions can be still too many and thus difficult to construct and generate. Therefore *extralogical* constraints are often imposed. Basically all such constraint restrict the language of the hypothesis to a smaller subset of Horn clause logic. The most often used subsets of Horn clauses are:

- *Function-free clauses* (Datalog). These simplifies all operations discussed above. Actually each clause can be transformed into a function-free form by introducing new predicate symbols.

- *Generative clauses.* These clauses require all variables in the clause head to appear in the clause body. This is not a very strong requirement, however it reduces substantially the space of possible clauses.

- *Determinate literals.* This restriction concerns the body literals in the clauses. Let P be a logic program, $M(P)$ – its model, E^+ – positive examples and $A : -B_1, ..., B_m, B_{m+1}, ..., B_n$ – a clause from P. The literal B_{m+1} is *determinate*, iff for any substitution θ, such that $A\theta \in E^+$, and $\{B_1, ..., B_m\}\theta \subseteq M(P)$, there is a *unique* substitution δ, such that $B_{m+1}\theta\delta \in M(P)$.

  For example, consider the program

  ```
  p(A,D):-a(A,B),b(B,C),c(C,D).
  a(1,2).
  b(2,3).
  c(3,4).
  c(3,5).
  ```

  Literals $a(A, B)$ and $b(B, C)$ are determinate, but $c(C, D)$ is not determinate.

8.6 Illustrative examples

In this section we shall discuss three simple examples of solving ILP problems.

Example 1. Single example, single hypothesis.

Consider the background knowledge B

```
haswings(X):-bird(X)
bird(X):-vulture(X)
```

and the example $E^+ = \{haswings(tweety)\}$. The ground unit clauses, which are logical consequences of $B \wedge \neg E^+$ are the following:

$C = \neg bird(tweety) \wedge \neg vulture(tweety) \wedge \neg haswings(tweety)$

This gives three most specific clauses for the hypothesis. So, the hypothesis could be any one of the following facts:

```
bird(tweety)
vulture(tweety)
haswings(tweety)
```

Example 2.

Suppose that $E^+ = E_1 \wedge E_2 \wedge ... \wedge E_n$ is a set of ground atoms, and C is the set of ground unit positive consequences of $B \wedge \neg E^+$. It is clear that

$$B \wedge \neg E^+ \vdash \neg E^+ \wedge C$$

Substituting for E^+ we obtain

$$B \wedge \neg E^+ \vdash (\neg E_1 \wedge C) \vee (\neg E_2 \wedge C) \vee ... \vee (\neg E_n \wedge C)$$

Therefore $H = (E_1 \vee \neg C) \wedge (E_1 \vee \neg C) \wedge ... \wedge (E_n \vee \neg C)$, which is a set of clauses (logic program).

Consider an example.

$B = \{father(harry, john), father(john, fred), uncle(harry, jill)\}$

$E^+ = \{parent(harry, john), parent(john, fred)\}$

The ground unit positive consequences of $B \wedge \neg E^+$ are

$C = father(harry, john) \wedge father(john, fred) \wedge uncle(harry, jill)$

Then the most specific clauses for the hypothesis are $E_1 \vee \neg C$ and $E_2 \vee \neg C$:

```
parent(harry,john):-father(harry,john),
                    father(john,fred),
                    uncle(harry,jill)

parent(john,fred):-father(harry,john),
                    father(john,fred),
                    uncle(harry,jill)
```

Then $lgg(E_1 \vee \neg C, E_2 \vee \neg C)$ is

```
parent(A,B):-father(A,B),father(C,D),uncle(E,F)
```

This clause however contains redundant literals, which can be easily removed if we restrict the language to determinate literals. Then the final hypothesis is:

```
parent(A,B):-father(A,B)
```

Example 3. Predicate Invention.
$B = \{min(X, [X]), 3 > 2\}$
$E^+ = \{min(2, [3, 2]), min(2, [2, 2])\}$
The ground unit-positive consequences of $B \wedge \neg E^+$ are the following:
$C = min(2, [2]) \wedge min(3, [3]) \wedge 3 > 2$
As before we get the two most specific hypotheses:

```
min(2,[3,2]):-min(2,[2]),min(3,[3]),3>2
min(2,[2,2]):-min(2,[2]),min(3,[3]),3>2
```

We can now generalize and simplify these clauses, applying the restriction of determinate literals.

```
min(X,[Y|Z]):-min(X,Z),Y>X
min(X,[X|Y]):-min(X,Y)
```

Then we can apply the "W"-operator in the following way (the corresponding substitutions are shown at the arms of the "W"):

```
   p(B,A):-B>A     min(A,[B|C]):-min(A,C),p(B,A)      p(B,B)
         \                     /\                      /
          \                   /  \                    /
           \                 /    \                  /
            \               /      \                /
         {B/Y,A/Y}         /  {A/X,B/X,C/Y}      {B/X}
              \           /          \            /
               \   {A/X,B/Y,C/Z}      \          /
                \       /              \        /
                 \     /                \      /
                  \   /                  \    /
                   \ /                    \  /
                    \/                     \/
     min(X,[Y|Z]):-min(X,Z),Y>X   min(X,[X|Y]):-min(X,Y)
```

Obviously the semantics of the "invented" predicate `p` is "$\geq$" (greater than or equal to).

8.7 Basic strategies for solving the ILP problem

Generally two strategies can be explored:

- *Specific to general search.* This is the approach suggested by condition (1) (Section 1) allowing deductive inference of the hypothesis. First, a number of most specific clauses are constructed and then using "V", "W", *lgg* or other generalization operators this set is converged in one of several generalized clauses. If the problem involves negative examples, then the currently generated clauses are tested for correctness using the strong consistency condition. This approach was illustrated by the examples.

- *General to specific search.* This approach is mostly used when some heuristic techniques are applied. The search starts with the most general clause covering E^+. Then this clause is further specialized (e.g. by adding body literals) in order to avoid covering of E^-. For example, the predicate $parent(X, Y)$ covers E^+ from example 2, however it is too general and thus coves many other irrelevant examples too. So, it should be specialized by adding body literals. Such literals can be constructed using predicate symbols from B and E^+. This approach is explored in the system FOIL [Quinlan, 1990].

Chapter 9

Bayesian approach and MDL

9.1 Bayesian induction

$$P(H_i|E) = \frac{P(H_i)P(E|H_i)}{\sum_{i=1}^{n} P(H_i)P(E|H_i)}$$

9.2 Occams razor

$E^+ = \{0, 000, 00000, 000000000\}$,
$E^- = \{\varepsilon, 00, 0000, 000000\}$.

$G_1 : S \to 0|000|00000|000000000$,
$G_2 : S \to 00S|0$,

9.3 Minimum Description Length (MDL) principle

$$-\log_2 P(H_i|E) = -\log_2 P(H_i) - \log_2 P(E|H_i) + C,$$

$$L(H|E) = L(H) + L(E|H),$$

$$L(E) > L(H) + L(E|H).$$

9.4 Evaluating propositional hypotheses

$$L(R_i) = -\log_2 \frac{1}{\binom{ts}{k_i}} = \log_2 \binom{ts}{k_i}.$$

9.4.1 Encoding exceptions

$$L(E|H) = \log_2 \binom{tp + fp}{fp} + \log_2 \binom{tn + fn}{fn}.$$

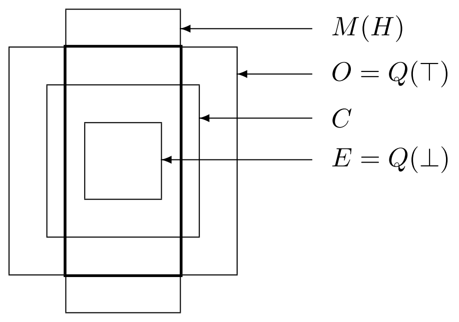


Figure 9.1:

9.4.2 Encoding entropy

$$e_i = -\frac{p_i}{n_i} * \log_2 \frac{p_i}{n_i} - \frac{n_i - p_i}{n_i} * \log_2 \frac{n_i - p_i}{n_i},$$

9.5 Evaluating relational hyporheses

9.5.1 Complexity of logic programs

$$L_{PC}(E|H) = \sum_{A \in E} L_{PC}(A|H).$$

$$L_{PC}(E|\top) = \sum_{A \in E} \log_2 c^n = |E| * n * \log_2 c,$$

$$L_{PC}(E|\bot) = \sum_{A \in E} L_{PC}(A|\bot) = |E| * \log_2 |E|.$$

9.5.2 Learning from positive only examples

Chapter 10

Unsupervised Learning

10.1 Introduction

Two basic approaches can be distinguished in this area: *finding regularities in data* (discovery) and *conceptual clustering.* The former approach is basically connected with the famous AM program [Davis and Lenat,1982], designed originally to discover concepts in mathematics. AM uses some notions of set theory, operations for creating new concepts by modifying and combining existing ones, and a set of heuristics for detecting "interesting" concepts. For example, AM discovered the natural numbers by modifying its notion of multiset (set with multiple occurrences of the same element). By using multiset of a single element AM discovered a representation of natural numbers as multisets and the operation on numbers as set operations (e.g. $\{1,1,1\} \cup \{1,1\} = \{1,1,1,1,1\}$ corresponds to $3+2=5$). The heuristics which AM used to evaluate the interesting concepts were very domain dependent. Therefore it was difficult to apply the system in other fields beyond the elementary number theory.

Clustering is known from mathematics, where the basic idea is to group some object in a cluster using the euclidean distance between them as a criterion for their "similarity". When using structural description of the objects, however, the traditional clustering algorithms fail to capture any domain knowledge related to the object descriptions. Another disadvantage is that these algorithms represent clusters *extensionally,* i.e. by enumerating all their members. However, an *intensional* description of the clusters (i.e. such that can be used for classifying objects by using their descriptions in terms of relations, properties, features etc.) can produce a semantic explanation of the resulting categories, which is more human-like.

Conceptual clustering is an approach, which addresses the above mentioned problems. We shall discuss briefly two instances of this approach - CLUSTER/2 [Michalski and Stepp, 1983] and COBWEB [Gennari et al, 1989].

10.2 CLUSTER/2

This algorithm forms k categories by constructing individual objects grouped around k *seed* objects. It is as follows:

1. Select k objects (seeds) from the set of observed objects (randomly or using some selection function).

2. For each seed, using it as a positive example the all the other seeds as negative examples, find a maximally general description that covers all positive and none of the negative examples.

3. Classify all objects form the sample in categories according to these descriptions. Then replace each maximally general descriptions with a maximally specific one, that cover all objects in the category. (This possibly avoids category overlapping.)
4. If there are still overlapping categories, then using some metric (e.g. euclidean distance) find central objects in each category and repeat steps 1-3 using these objects as seeds.
5. Stop when some quality criterion for the category descriptions is satisfied. Such a criterion might be the complexity of the descriptions (e.g. the number of conjuncts)
6. If there is no improvement of the categories after several steps, then choose new seeds using another criterion (e.g. the objects near the edge of the category).

The underlying idea of the above algorithm is to find *necessary and sufficient conditions for category membership.* There is however some psychological evidence that human categorization is based on the notion of *prototypicality.* For example, the *family resemblance theory* [Wittgenstein, 1953] argues that categories are defined by a system of similarities between the individual members of a category.

Another feature of human categorization is the use of *base-level* categories. In contrast to the formal hierarchies used often in AI (e.g. the taxonomic trees, Chapter 1), the humans mostly use categories which are neither most general, nor most specific. For example, the concept of "chair" is most basic than both its generalization "furniture" and its specialization "office chair", and "car" is more basic than both "porshe" and "vehicle".

The COBWEB algorithm, though not designed as a cognitive model, accounts for the above mentioned features of human categorization.

10.3 COBWEB

COBWEB is an incremental learning algorithm, which builds a taxonomy of categories without having a predefined number of categories. The categories are represented *probabilistically* by the conditional probability $p(f_i = v_{ij}|c_k)$ with which feature f_i has value v_{ij}, given that an object is in category c_k.

Given an instance COBWEB evaluates the quality of either placing the instance in an existing category or modifying the hierarchy to accommodate the instance. The criterion used for this evaluation is based on *category utility*, a measure that Gluck and Corter have shown predicts the basic level ound in psychological experiments. Category utility attempts to maximize both the probability that two objects in the same category have values in common and the probability that objects from different categories have different feature values. It is defined as follows:

$$\sum_k \sum_i \sum_j p(f_i = v_{ij}) p(f_i = v_{ij}|c_k) p(c_k|f_i = v_{ij})$$

The sum is calculates for all categories, all features and all values. $p(f_i = v_{ij}|c_k)$ is called *predictability*, i.e. this is the probability that an object has value v_{ij} for its feature f_j, given that it belongs to category c_k. The higher this probability, the more likely two objects in a category share the same feature values. $p(c_k|f_i = v_{ij})$ is called *predictiveness*, i.e. the probability that an object belongs to category c_k, given that it has value v_{ij} for its feature f_i. The greater this probability, the less likely objects from different categories will have feature values in common. $p(f_i = v_{ij})$ is a weight, assuring that frequently occurring feature values will have stronger influence on the evaluation.

Using the Bayes' rule we have $p(a_i = v_{ij})p(c_k|a_i = v_{ij}) = p(c_k)p(a_i = v_{ij}|c_k)$. Thus we can transform the above expression into an equivalent form:

$$\sum_k p(c_k) \sum_i \sum_j p(f_i = v_{ij}|c_k)^2$$

Gluck and Corter have shown that the subexpression $\sum_i \sum_j p(f_i = v_{ij}|c_k)^2$ is the *expected* number of attribute values that one can correctly guess for an arbitrary member of class c_k. This expectation assumes a *probability matching* strategy, in which one guesses an attribute value with a probability equal to its probability of occurring. They define the category utility as the *increase* in the expected number of attribute values that can be correctly guessed, given a set of n categories, over the expected number of correct guesses without such knowledge. The latter term is $\sum_i \sum_j p(f_i = v_{ij})^2$, which is to be subtracted from the above expression. Thus the complete expression for the category utility is the following:

$$\frac{\sum_k p(c_k) \sum_i \sum_j [p(f_i = v_{ij}|c_k)^2 - p(f_i = v_{ij})^2]}{k}$$

The difference between the two expected numbers is divided by k, which allows us to compare different size clusterings.

When a new instance is processed the COBWEB algorithm uses the discuss above measure to evaluate the possible clusterings obtained by the following actions:

– classifying the instance into an existing class;
– creating a new class and placing the instance into it;
– combining two classes into a single class (merging);
– dividing a class into two classes (splitting).

Thus the algorithm is a *hill climbing* search in the space of possible clusterings using the above four operators chosen by using the category utility as an evaluation function.

```
cobweb($Node$,$Instance$)
begin
  • If $Node$ is a leaf then begin
      Create two children of Node - $L_1$ and $L_2$;
      Set the probabilities of $L_1$ to those of $Node$;
      Set the probabilities of $L_2$ to those of $Insnatce$;
      Add $Instance$ to $Node$, updating $Node$'s probabilities.
    end
  • else begin
      Add $Instance$ to $Node$, updating $Node$'s probabilities; For each child $C$ of $Node$, compute the category utility of clustering achieved by placing $Instance$ in $C$;
      Calculate:
      $S_1$ = the score for the best categorization ($Instance$ is placed in $C_1$);
      $S_2$ = the score for the second best categorization ($Instance$ is placed in $C_2$);
      $S_3$ = the score for placing $Instance$ in a new category;
      $S_4$ = the score for merging $C_1$ and $C_2$ into one category;
      $S_5$ = the score for splitting $C_1$ (replacing it with its child categories.
    end
  • If $S_1$ is the best score then call cobweb($C_1$, $Instance$).
  • If $S_3$ is the best score then set the new category's probabilities to those of $Instance$.
  • If $S_4$ is the best score then call cobweb($C_m$, $Instance$), where $C_m$ is the result of merging $C_1$ and $C_2$.
  • If $S_5$ is the best score then split $C_1$ and call cobweb($Node$, $Instance$).
```

end

Consider the following set of instances. Each one defines a one-celled organism using the values of three features: *number of tails*, *color* and *number of nucleis* (the first element of the list is the name of the instance):

```
instance([cell1,one,light,one]).
instance([cell2,two,dark,two]).
instance([cell3,two,light,two]).
instance([cell4,one,dark,three]).
```

The following is a trace of a program written in Prolog implementing the COBWEB algorithm:

```
 Processing instance cell1 ...
 Root initialized with instance: node(root,1,1)

 Processing instance cell2 ...
 Root node:node(root,1,1) used as new terminal node:node(node_1,1,1)
 Case cell2 becomes new terminal node(node_2,1,1)
 Root changed to: node(root,2,0.5)

 Processing instance cell3 ...
 Root changed to: node(root,3,0.555553)
 Incorporating instance cell3 into node: node(node_4,2,0.833333)
 Using old node: node(node_2,1,1) as terminal node.
 New terminal node: node(node_7,1,1)

 Processing instance cell4 ...
 Root changed to: node(root,4,0.458333)
 New terminal node: node(node_10,1,1)
yes

?- print_kb.
d_sub(root,node_10).
d_sub(node_4,node_7).
d_sub(node_4,node_2).
d_sub(root,node_4).
d_sub(root,node_1).

node_10(nuclei,[three-1]).
node_10(color,[dark-1]).
node_10(tails,[one-1]).
root(nuclei,[one-1,three-1,two-2]).
root(color,[dark-2,light-2]).
root(tails,[one-2,two-2]).
node_7(nuclei,[two-1]).
node_7(color,[light-1]).
node_7(tails,[two-1]).
node_4(nuclei,[two-2]).
node_4(color,[dark-1,light-1]).
node_4(tails,[two-2]).
node_2(nuclei,[two-1]).
```

```
node_2(color,[dark-1]).
node_2(tails,[two-1]).
node_1(tails,[one-1]).
node_1(color,[light-1]).
node_1(nuclei,[one-1]).
```

The *print_kb* predicate prints the category hierarchy (*d_sub* structures) and the description of each category (*node_i* structures). The first argument of the *node_i* structures is the corresponding feature, and the second one is a list of pairs $Val - Count$, where $Count$ is a number indicating the number of occurrences of Val as a value of the feature.

Chapter 11

Explanation-based Learning

11.1 Introduction

The inductive learning algorithms discussed in Chapters 1-4 generalize on the basis of regularities in training data. These algorithms are often referred to as *similarity based*, i.e. generalization is primarily determined by the syntactical structure of the training examples. The use of domain knowledge is limited to specifying the syntax of the hypothesis language and exploring the hierarchy of the attribute values.

Typically a learning system which uses domain knowledge is expected to have some ability to solve problems. Then the point of learning is to improve the system's knowledge or system's performance using that knowledge. This task could be seen as *knowledge reformulation* or *theory revision.*

Explanation-based learning (EBL) uses a domain theory to construct an explanation of the training example, usually a proof that the example logically follows from the theory. Using this proof the system filters the noise, selects only the relevant to the proof aspects of the domain theory, and organizes the training data into a systematic structure. This makes the system more efficient in later attempts to deal with the same or similar examples.

11.2 Basic concepts of EBL

1. *Target concept.* The task of the learning system is to find an effective definition of this concept. Depending on the specific application the target concept could be a classification, theorem to be proven, a plan for achieving goal, or heuristic to make a problem solver more efficient.

2. *Training example.* This is an instance of the target concept.

3. *Domain theory.* Usually this is a set of rules and facts representing domain knowledge. They are used to explain how the training example is an instance of the target concept.

4. *Operationality criteria.* Some means to specify the form of the concept definition.

11.3 Example

Consider the following *domain theory* in the form of Prolog facts and rules.

```
on(object1,object2).

isa(object1,box).
isa(object2,table).
isa(object3,box).

color(object1,red).
color(object2,blue).

volume(object1,1).
volume(object3,6).

density(object1,2).
density(object3,2).


safe_to_stack(X,Y) :- lighter(X,Y).

lighter(O1,O2) :- weight(O1,W1), weight(O2,W2), W1 < W2.

weight(O,W) :- volume(O,V), density(O,D), W is V * D.
weight(O,5) :- isa(O,table).
```

The *operational criterion* defines the predicates which can be used to construct the concept definition. Those include the facts and arithmetic predicates such as ">", "<" and "*is*" (actually this set can be extended to all Prolog built-in predicates).

Depending on the training example - an instance of a *safe_to_stack* predicate, the following definitions of the *target concept* can be obtained:

- For training example $safe\_to\_stack(object1, object3)$ the concept definition is as follows:

  ```
  safe_to_stack(A,B) :-
     volume(A,C),
     density(A,D),
     E is C * D,
     volume(B,F),
     density(B,G),
     H is D * G,
     E < H.
  ```

- For training example $safe\_to\_stack(object1, object2)$ the concept definition is as follows:

  ```
  safe_to_stack(A,B) :-
     volume(A,C),
     density(A,D),
     E is C * D,
     isa(B,table),
     E < 5
  ```

- For training example $safe\_to\_stack(object2, object3)$ the concept definition is as follows:

```
safe_to_stack(A,B) :-
   isa(A,table),
   volume(B,C),
   density(B,D),
   E is C * D,
   5 < E
```

The process of building the target concept definition is accomplished by generalization of the proof tree for the training example. This process is called *goal regression*. In the particular case described above, the goal regression is simply variabalization (replacing same constants with same variables) of the leaves of the Prolog proof tree and then using that leaves as goals in the body of the target concept.

11.4 Discussion

In the form discussed here EBL can be seen as *partial evaluation*. In terms of Prolog this technique is called some times *unfolding*, i.e. replacing some body literals with the bodies of the clauses they match, following the order in which Prolog reduces the goals. Hence in its pure form EBL doesn't learn anything new, i.e. all the rules inferred belong to the *deductive closure* of the domain theory. This means that these rules can be inferred from the theory without using the training example at all. The role of the training example is only to focus the theorem prover on relevant aspects of the problem domain. Therefore EBL is often viewed as a form of *speed up learning* or *knowledge reformulation*.

Consequently EBL can be viewed not as a form of generalization, but rather as *specialization*, because the rule produced is more specific than a theory itself (it is applicable only to one example).

All these objections however do not undermine the EBL approach as a Machine Learning one. There are three responses to them:

1. There are small and well defined theories, however practically inapplicable. For example, consider the game of chess. The rules of chess combined with an ability to perform unlimited look-ahead on the board states will allow a system to play well. Unfortunately this approach is not practically useful. An EBL system given well chosen training examples will not add anything new to the rules of chess playing, but will learn actually some heuristics to apply these rules, which might be practically useful.

2. An interesting application of the EBL techniques is to incomplete or incorrect theories. In such cases an incomplete (with some failed branches) or incorrect proof can indicate the deficiency of the theory and give information how to refine or complete it.

3. An integration of EBL and other similarity based approaches could be fruitful. This can be used to refine the domain theory by building explanations of successful and failed examples and passing them as positive and negative examples correspondingly to an inductive learning system.

Bibliography

[1] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[2] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[3] S. B. Thrun et al. The MONK's problems - a performance comparison of different learning algorithms. Technical Report CS-CMU-91-197, Carnegie Mellon University, Dec. 1991.

[4] P. H. Winston. Learning structural descriptions form examples. In P. H. Winston, editor, *The Psychology of Computer Vision.* McGraw Hill, New York, 1975.