Lecture Notes in Machine Learning – Chapter 3: Languages for learning

Zdravko Markov

February 9, 2004

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 **??**.)

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 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.

2 Relational languages

Figure 1 shows a sample from a set of examples describing a concept often used in ML, the so called MONKS concept [2]. 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 Language of logic programming

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 ($\Box$) 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*.

3.2 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 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
?-
```

References

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

[2] 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.