

 Open access • Book Chapter • DOI:10.1007/978-0-387-30164-8_489

Logic of generality — Source link





Luc De Raedt

Published on: 01 Jan 2010

Topics: Computational logic, Dynamic logic (modal logic), Substructural logic, Autoepistemic logic and Multimodal logic

Related papers:

- [Logic of Generality.](#)
- [Functions and Generality of Logic](#)
- [On the logic of design](#)
- [A general logic](#)
- [Logic and Its Applications](#)

Share this paper:    

View more about this paper here: <https://typeset.io/papers/logic-of-generality-btfi3ei2zo>

The logic of generality

Luc De Raedt

Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, BE - 3001 Heverlee, Belgium, email: luc.deraedt@cs.kuleuven.be

Definition

One hypothesis is *more general* than another one if it covers all instances that are also covered by the later one. The former hypothesis is called a *generalization* of the later, and the later a *specialization* of the former. When using logical formulae as hypotheses, the generality relation coincides with the notion of logical entailment, which implies that the generality relation can be analyzed from a logical perspective. The logical analysis of generality, which is pursued in this entry, leads to the perspective of *induction as the inverse of deduction*. This forms the basis for an analysis of various logical frameworks for reasoning about generality and for traversing the space of possible hypotheses. Many of these frameworks (such as for instance, *θ -subsumption*) are employed in the field inductive logic programming and are introduced below.

Synonyms

generality and logic, induction as inverted deduction, is more general than, is more specific than, specialization, inductive inference rules,

Motivation

Symbolic machine learning methods typically learn by searching a hypothesis space. The hypothesis space can be (partially) ordered by the generality relation, which serves as the basis for defining operators to traverse the space as well as for pruning away unpromising parts of the search space. This is often realized through the use of refinement operators, that is, generalization and specialization operators. Because many learning methods employ a hypothesis language that is logical or that can be reformulated in logic, it is interesting to analyze the generality relation from a logical perspective. When using logical formulae as hypotheses, the generality relation closely corresponds to logical entailment. This allows us to directly transfer results from logic to a machine learning context. In particular, machine learning operators can be derived from logical inference rules. The logical

theory of generality provides a framework for transferring these results. Within the standard setting of inductive logic programming, learning from entailment, specialization is realized through deduction, and generalization through induction, which is considered to be the inverse of deduction. Different deductive inference rules lead to different frameworks for generalization and specialization. The most popular one is that of θ -subsumption, which is employed by the vast majority of contemporary inductive logic programming systems.

Theory

A hypothesis g is *more general than* a hypothesis s if and only if g covers all instances that are also covered by s , more formally, if $\text{covers}(s) \subseteq \text{covers}(g)$, where $\text{covers}(h)$ denotes the set of all instances covered by the hypothesis h .

There are several possible ways to represent hypotheses and instances in logic [3, 2], each of which results in a different setting with a corresponding *covers* relation. Some of the best known settings are *learning from entailment*, *learning from interpretations* and *learning from proofs*.

Learning from entailment

When learning from entailment, both hypotheses and instances are logical formulae, typically *definite clauses*, which underly the programming language Prolog [5]. Furthermore, when learning from entailment, a hypothesis h covers an instance e if and only if $h \models e$, that is when h logically entails e , or equivalently, when e is a logical consequence of h . For instance, consider the hypothesis h :

```
flies :- bird, normal.
bird  :- blackbird.
bird  :- ostrich.
```

The first clause or rules can be read as `flies` **if** `normal` **and** `bird`, that is, normal birds fly. The second and third state that blackbirds, resp. ostriches, are birds. Consider now the examples e_1 :

```
flies :- blackbird, normal, small.
```

and e_2 :

```
flies :- ostrich, small.
```

Example e_1 is covered by h , because it is a logical consequence of h , that is $h \models e_1$. On the other hand, example e_2 is not covered, which we denote as $h \not\models e_2$.

When learning from entailment, the following property holds.

Property 1 *A hypothesis g is more general than a hypothesis s if and only if g logically entails s , that is $g \models s$.*

This is easy to see. Indeed, g is more general than s if and only if $\text{covers}(s) \subseteq \text{covers}(g)$ if and only if for all examples e : $(s \models e) \rightarrow (g \models e)$ if and only if $g \models s$. For instance, consider the hypothesis h_1 :

```
flies :- blackbird, normal.
```

Because $h \models h_1$, it follows that h covers all examples by h_1 , and hence, h generalizes h_1 .

Property 1 states that the generality relation coincides with logical entailment when learning from entailment. In other learning settings, such as when *learning from interpretations*, this relationship also holds though the direction of the relationship might change.

Learning from interpretations

When learning from interpretations, hypotheses are logical formulae, typically sets of definite clauses, and instances are interpretations. For propositional theories, interpretations are assignments of truth-values to propositional variables. For instance, continuing the `flies` illustration, two interpretations could be

```
{blackbird, bird, normal, flies}
{ostrich, small}
```

where we specify interpretations through the set of propositional variables that are true. An interpretation specifies a kind of possible world. A hypothesis h then covers an interpretation if and only if the interpretation is a model for the hypothesis. An interpretation is a model for a hypothesis if it satisfies all clauses in the hypothesis. In our illustration, the first interpretation is a model for the theory h but the second is not. Because the condition part of the rule `bird :- ostrich.` is satisfied in the second interpretation (as it contains `ostrich`), the conclusion part, that is `bird`, should also belong to the interpretation in order to have a model. Thus, the first example is covered by the theory h , but the second is not.

When learning from interpretations, a hypothesis g is more general than a hypothesis s if and only if for all examples e : $(e \text{ is a model of } s) \rightarrow (e \text{ is a model of } g)$ if and only if $s \models g$.

Because the learning from entailment setting is more popular than the learning from interpretations setting, we shall employ in this section the usual convention which states that one hypothesis g is more general than a hypothesis s if and only if $g \models s$.

An operational perspective

Property 1 lies at the heart of the theory of *inductive logic programming* and generalization because it directly relates the central notions of logic with those of machine learning [10]. It is also extremely useful because it allows us to directly transfer results from logic to machine learning.

This can be illustrated using traditional deductive inference rules, which start from a set of formulae and derive a formulae that is entailed by the original set. For instance, consider the *resolution* inference rule for propositional definite clauses:

$$\frac{h \leftarrow g, a_1, \dots, a_n \text{ and } g \leftarrow b_1, \dots, b_m}{h \leftarrow b_1, \dots, b_m, a_1, \dots, a_n} \quad (1)$$

This inference rule starts from the two rules above the line and derives the so-called *resolvent* below the line. This rule can be used to infer h_1 from h . An alternative deductive inference rule adds a condition to a rule:

$$\frac{h \leftarrow a_1, \dots, a_n}{h \leftarrow a, a_1, \dots, a_n} \quad (2)$$

This rule can be used to infer that h_1 is more general than the clause used in example e_1 . In general, a deductive inference rule can be written as

$$\frac{g}{s} \quad (3)$$

If s can be inferred from g and the operator is *sound*, then $g \models s$. Thus applying a deductive inference rule realizes specialization, and hence, deductive inference rules can be used as specialization operators. A *specialization operator* maps a hypothesis onto a set of its specializations. Because specialization is the inverse of generalization, *generalization operators* — which map a hypothesis onto a set of its generalizations — can be obtained by inverting deductive inference rules. The inverse of a deductive inference rule written in format (3) works from bottom to top, that is from s to g . Such an inverted deductive inference rule is called an *inductive* inference rule. This leads to the view of induction as the inverse of deduction. This view is operational as it implies that each deductive inference rule can be inverted into an inductive one, and also, that each inference rule provides in an alternative framework for generalization.

An example generalization operator is obtained by inverting the adding condition rule (2). It corresponds to the well-known “dropping condition” rule [6]. As we will see soon, it is also possible to invert the resolution principle (1).

Before deploying inference rules, it is necessary to determine their properties. Two desirable properties are *soundness* and *completeness*. These properties are based on the repeated application of inference rules. Therefore, we write $g \vdash_r s$ when there exists a sequence of hypotheses h_1, \dots, h_n such that

$$\frac{g}{h_1}, \frac{h_1}{h_2}, \dots, \frac{h_n}{s} \text{ using } r \quad (4)$$

A set of inference rules r is then *sound* whenever $g \vdash_r s$ implies $g \models s$; and *complete* whenever $g \models s$ implies $g \vdash_r s$. In practice, soundness is always enforced though completeness is not always required in a machine learning setting. When working with incomplete rules, one should realize that the generality relation “ \vdash_r ” is weaker than the logical one “ \models ”.

The most important logical frameworks for reasoning about generality, such as θ -subsumption and resolution, will be introduced below using the above introduced logical theory of generality.

Frameworks for generality

Propositional Subsumption

Many propositional learning systems employ hypotheses that consist of rules, often definite clauses as in the `flies` illustration above. The propositional subsumption relation defines a generality relation amongst clauses and is defined through the adding condition rule (2). The properties then follow by applying the logical theory of generalization presented above to this inference rule. More specifically, the generality relation \vdash_a induced by the adding condition rule states that a clause g is more general than a clause s if s can be derived from g by adding a sequence of conditions to g . Observing that a clause $h \leftarrow a_1, \dots, a_n$ is a disjunction of literals $h \vee \neg a_1 \dots \vee \neg a_n$ allows us to write it in set notation as $\{h, \neg a_1, \dots, \neg a_n\}$. The soundness and completeness of propositional subsumption then follow from

$$g \vdash_a s \text{ if and only if } g \subseteq s \text{ if and only if } g \models s \quad (5)$$

which also clarifies states that g subsumes s if and only if $g \subseteq s$.

The propositional subsumption relation induces a complete lattice on the space of possible clauses. A *complete lattice* is a partial order — a reflexive, anti-symmetric and transitive relation — where every two elements posses a unique

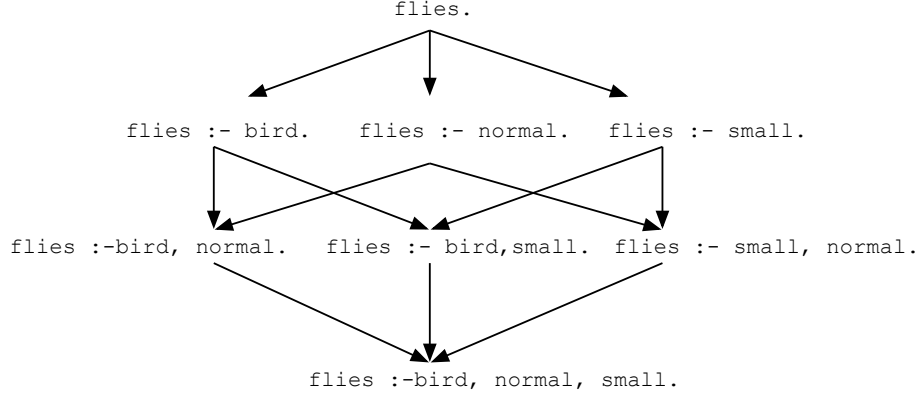


Figure 1: The Hasse diagram for the predicate `flies`.

least upper and greatest lower bound. An example lattice for rules defining the predicate `flies` in terms of `bird`, `normal` and `small` is illustrated in the Hasse diagram depicted in Figure 1.

The Hasse diagram also visualizes the different operators that can be used. The *generalization* operator ρ_g maps a clause to the set of its children in the diagram, whereas the *specialization* operator ρ_s maps a clause to the set of its parents. So far we have defined such operators *implicitly* through their corresponding inference rules. In the literature, they are often defined *explicitly*:

$$\rho_g(h \leftarrow a_1, \dots, a_n) = \{h \leftarrow a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n \mid i = 1, \dots, n\} \quad (6)$$

In addition to using the inference rules directly, some systems like Golem [11] also exploit the properties of the underlying lattice by computing the least upper bound of two formulae. The least upper bound operator is known under the name of *least general generalization* (*lgg*) in the machine learning literature. It returns the least common ancestor in the Hasse diagram. Using set notation for clauses the definition of the *lgg* is:

$$lgg(c_1, c_2) = c_1 \cap c_2 \quad (7)$$

The least general generalization operator is used by machine learning systems that follow a cautious generalization strategy. They take two clauses corresponding to positive examples and minimally generalize them.

θ -subsumption

The most popular framework for generality within inductive logic programming is θ -subsumption [13]. It provides a generalization relation for clausal logic and it extends propositional subsumption to first order logic. A *definite clause* is an expression of the form $h \leftarrow a_1, \dots, a_n$ where h and the a_i are logical atoms. An *atom* is an expression of the form $p(t_1, \dots, t_m)$ where p is a *predicate name* (or, the name of a relation) and the t_i are terms. A *term* is either a constant (denoting an object in the domain of discourse), a variable, or a structured term of the form $f(u_1, \dots, u_k)$ where f is a functor symbol (denoting a function in the domain of discourse) and the u_i are terms, see [5] for more details. Consider for instance the clauses

```
likes(X, Y) :- neighbours(X, Y) .
likes(X, husbandof(Y)) :- likes(X, Y) .
likes(X, tom) :- neighbours(X, tom), male(X) .
```

The first clause states that X likes Y if X is a neighbour of Y . The second one that X likes the husband of Y if X likes Y . The third one that all male neighbour of tom like tom .

Θ -subsumption is not only based on the adding condition rule (2) but also on the *substitution rule*:

$$\frac{g}{g\theta} \quad (8)$$

The substitution rule applies a substitution θ to the definite clause g . A substitution $\{V_1/t_1, \dots, V_n/t_n\}$ is an assingment of terms to variables. Applying a substitution to a clause c yields the instantiated clause where all variables are simultaneously replaced by their corresponding terms.

Θ -subsumption is then the generality relation induced by combining the substitution with the adding condition rule in the set t .

$$g \text{ } \theta\text{-subsumes } s \text{ if and only if } g \vdash_t s \text{ if and only if } \exists \theta : g\theta \subseteq s \quad (9)$$

For instance, the first clause for `likes` subsumes the third one with the substitution $\{Y/\text{tom}\}$.

θ -subsumption has some interesting properties:

- θ -subsumption is sound.
- θ -subsumption is complete for clauses that are not self-recursive. It is incomplete for self-recursive clauses such as

$\text{nat}(s(X)) :- \text{nat}(X)$
 $\text{nat}(s(s(Y))) :- \text{nat}(Y)$

for which one can use resolution to that the first clause logically entails the second one, even though it does not θ -subsume it.

- Deciding θ -subsumption is an NP-complete problem.

Because θ -subsumption is relatively simplicity and decidable whereas logical entailment between single clauses is undecidable, it is used as the generality relation by the majority of Inductive logic programming systems. These systems typically employ a specialization or refinement operator to traverse the search space. To guarantee the systematic enumeration of the search space, the specialization operator ρ_s can be employed. $\rho_s(c)$ is obtained by applying the adding condition or substitution rule with the following restrictions.

- the adding condition rule only adds atoms of the form $p(V_1, \dots, V_n)$ where the V_i are variables not yet occurring in the clause c ;
- the substitution rule only employs *elementary substitutions*, which are of the form
 - $\{V/Y\}$, where X and Y are two variables appearing in c
 - $\{V/ct\}$, where V is a variable in c and ct a constant
 - $\{V/f(V_1, \dots, V_n)\}$, where V is a variable in c , f a functor of arity n and the V_i are variables not yet occurring in c .

A generalization operator can be obtained by inverting ρ_s , which requires one to invert substitutions. Inverting substitutions is not easy. Whereas applying a substitution $\theta = \{V/a\}$ to a clause c replaces all occurrences of V by a and yields a unique clause $c\theta$, applying the substitution rule in the inverse direction does not necessarily yield a unique clause. If we assume the elementary substitution applied to

$$\frac{c}{q(a, a)} \quad (10)$$

was $\{V/a\}$, then there are at least three possibilities for c : $q(a, V)$, $q(V, a)$, and $q(V, V)$.

Θ -subsumption is reflexive, transitive but unfortunately not anti-symmetric, which can be seen by consider the clauses

```

parent(X,Y) :- father(X,Y) .
parent(X,Y) :- father(X,Y), father(U,V) .

```

The first clause clearly subsumes the second one because it is a subset. The second one subsumes the first with the substitution $\{X/U, V/Y\}$. The two clauses are therefore equivalent under θ -subsumption, and hence also logically equivalent. The loss of the anti-symmetry complicates the search process. The naive application of the specialization operator ρ_s may yield syntactic specializations that are logically equivalent. This is illustrated above where the second clause for `parent` is a refinement of the first one using the adding condition rule. In this way, useless clauses are generated and there is a danger that if the resulting clauses are further refined, there is a danger that the search will end up in an infinite loop.

Plotkin [13] has studied the quotient set induced by θ -subsumption and proven various interesting properties. The quotient set consists of classes of clauses that are equivalent under θ -subsumption. The class of clauses equivalent to a given clause c is denoted by

$$[c] = \{c' | c' \text{ is equivalent with } c \text{ under } \theta\text{-subsumption}\} \quad (11)$$

Plotkin proved that

- the quotient set is well-defined w.r.t. θ -subsumption.
- there is a representative, a canonical form, of each equivalence class, the so-called *reduced clause*. The reduced clause of an equivalence class is the shortest clause belonging to class. It is unique up to variable renaming. For instance, in the `parent` example above, the first clause is in reduced form.
- the quotient set forms a complete lattice, which implies that there is a least general generalization of two equivalence classes. In the inductive logic programming literature, one often talks about the least general generalization of two clauses.

Several variants of θ -subsumption have been developed. One of the most important ones is that of *OI*-subsumption [4]. For functor-free clauses, it modifies the substitution rule by disallowing substitutions that unify two variables or that substitute a variable by a constant already appearing in the clause. The advantage is that the resulting relation is anti-symmetric, which avoids some of the above mentioned problems with refinement operators. On the other hand, the minimally general generalization of two clauses is not necessary unique, and hence, there exists no least general generalization operator.

Inverse resolution

Applying resolution is a sound deductive inference rule, and therefore, realizes specialization. Reversing it yields inductive inference rules or generalization operators [7, 9]. This is typically realized by combining the resolution principle with a copy operator. The resulting rules are called *absorption* (19) and *identification* (21). They start from the clauses below and induce the clause above the line. They are shown here only for the propositional case, as the first order case requires one to deal with substitutions as well as inverse substitutions.

$$\frac{h \leftarrow g, a_1, \dots, a_n \text{ and } g \leftarrow b_1, \dots, b_m}{h \leftarrow b_1, \dots, b_m, a_1, \dots, a_n \text{ and } g \leftarrow b_1, \dots, b_m} \quad (12)$$

$$\frac{h \leftarrow g, a_1, \dots, a_n \text{ and } g \leftarrow b_1, \dots, b_m}{h \leftarrow b_1, \dots, b_m, a_1, \dots, a_n \text{ and } h \leftarrow g, a_1, \dots, a_n} \quad (13)$$

Other interesting inverse resolution operators perform *predicate invention*, that is they introduce new predicates that were not yet present in the original data. These operators invert two resolution steps. One such operator is the *intra-construction* operator (14). Applying this operator from bottom to top introduces the new predicate q that was not present before.

$$\frac{q \leftarrow l_1, \dots, l_k \text{ and } p \leftarrow k_1, \dots, k_n, q \text{ and } q \leftarrow l'_1, \dots, l'_m}{p \leftarrow k_1, \dots, k_n, l_1, \dots, l_k \text{ and } p \leftarrow k_1, \dots, k_n, l'_1, \dots, l'_m} \quad (14)$$

The idea of inverting the resolution operator is very appealing because it aims at inverting the most popular deductive inference operator, but is also rather complicated due to the non-determinism and the need to invert substitutions. Due to these complications, there are only few systems that employ inverse resolution operators.

Background knowledge

Inductive logic programming systems employ background knowledge during the learning process. Background knowledge typically takes the form of a set of clauses B , which is then used by the covers relation. When learning from entailment in the presence of background knowledge B an example e is covered by a hypothesis h if and only if $B \cup h \models e$. This notion of coverage is employed in most

of the work on inductive logic programming. in the initial `flies` example the two clauses defining `bird` would typically be considered background knowledge.

The incorporation of background knowledge in the induction process has resulted in frameworks for generality *relative* to a background theory. More formally, a hypothesis g is more general than a hypothesis s relative to the background theory B if and only if $B \cup g \models s$. The only already seen inference rules that deal with multiple clauses are those based on (inverse) resolution. The other frameworks can be extended to cope with this generality relation following the logical theory of generalization. Various frameworks have been developed along these lines. Some of the most important ones are relative subsumption [14] and generalized subsumption [1], which extend θ -subsumption and the notion of least general generalization towards the use of background knowledge. Computing the least general generalization of two clauses relative to the background theory is realized by first computing the most specific clauses covering the examples with regard to the background theory and then generalizing them using the least general generalization operator of θ -subsumption.

The first step is the most interesting one, and has been tackled under the name of *saturation* [15] and *bottom-clauses* [8]. We illustrate it within the framework of *inverse entailment* due to Stephen Muggleton [8]. The bottom clause $\perp(c)$ of a clause c with regard to a background theory B is the most specific clause $\perp(c)$ such that

$$B \cup \perp(c) \models c \quad (15)$$

If B consist of

```

polygon :- rectangle.
rectangle :- square.
oval :- circle.

```

and the example c is

```

positive :- red, square.

```

Then the bottom-clause $\perp(c)$ is

```

positive :- red, rectangle, square, polygon.

```

The bottom-clause is useful because it only lists those atoms that are relevant to the example, and only generalizations (under θ -subsumption) of $\perp(c)$ will cover the example. For instance, in the illustration, the bottom-clause does neither mention `oval` or `circle` as clauses for `pos` containing these atoms will never cover the

example clause c . Once the bottom-clause covering an example has been found the search process continues as if no background knowledge were present. Either specialization operators (typically under θ -subsumption) would search the space of clauses more general than $\perp(c)$, or else the least general generalization of multiple bottom-clauses would be computed.

Equation (15) is equivalent to

$$B \cup \neg c \models \neg \perp(c) \quad (16)$$

which explains why the bottom-clause is computed by finding all factual consequences of $B \cup \neg c$ and then inverting the resulting clause again. On the example:

$$\neg c = \{\neg \text{positive}, \text{red}, \text{square}\}$$

and the set of all consequences is

$$\neg \perp(c) = \neg c \cup \{\text{rectangle}, \text{polygon}\}$$

which then yields $\perp(c)$ mentioned above. When dealing with first order logic, bottom-clauses can become infinite, and therefore, one typically imposes further restrictions on the atoms that appear in bottom-clauses. These restrictions are part of the syntactic *bias*.

Further reading

The textbook by Nienhuys-Cheng and De Wolf [12] is the best reference for an in-depth formal description of various frameworks for generality in logic, in particular, for θ -subsumption and some of its variants. The book by De Raedt [3] contains a more complete introduction to inductive logic programming and relational learning, and also introduces the key frameworks for generality in logic. An early survey of inductive logic programming and the logical theory of generality is contained in [10]. Plotkin [13, 14] studied the use θ -subsumption and relative subsumption (under a background theory) for machine learning. Buntine [1] extended these frameworks towards generalized subsumption, and [4] introduced *OI*-subsumption. Inverse resolution was first used in the system Marvin [16], and then elaborated by [7] for propositional logic and by [9] for definite clause logic. Various learning settings are studied by [2] and discussed extensively by [3]. They are also relevant to probabilistic logic learning and statistical relational learning.

References

- [1] W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36:375–399, 1988.
- [2] L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.
- [3] L. De Raedt. *Logical and Relational Learning*. Springer, 2008.
- [4] F. Esposito, A. Laterza, D. Malerba, and G. Semeraro. Refinement of Datalog programs. In *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programing*, pages 73–94, 1996.
- [5] P.A. Flach. *Simply Logical: Intelligent Reasoning by Example*. John Wiley, 1994.
- [6] R. S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20(2):111–161, 1983.
- [7] S. Muggleton. Duce, an oracle based approach to constructive induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 287–292. Morgan Kaufmann, 1987.
- [8] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13(3-4):245–286, 1995.
- [9] S. Muggleton and W. Buntine. Machine invention of first order predicates by inverting resolution. In *Proceedings of the 5th International Workshop on Machine Learning*, pages 339–351. Morgan Kaufmann, 1988.
- [10] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.
- [11] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.
- [12] S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer, 1997.
- [13] G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

- [14] G. D. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6, pages 101–124. Edinburgh University Press, 1971.
- [15] C. Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14(2):219–232, 1994.
- [16] C. Sammut and R. B. Banerji. Learning concepts by asking questions. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2, pages 167–192. Morgan Kaufmann, 1986.

Coverage relation

Definition

In concept-learning (and also in rule-learning and inductive logic programming), the goal is often to find a hypothesis that covers, or matches, all positive examples by means of a search through the hypothesis space. The *covers* relation specifies the relationship between the hypothesis language, which is used to represent concepts, and the language used to represent instances or examples. The covers relation forms the basis for defining the generality relation, used to structure the search space. More formally, an example e is *covered* by a hypothesis h when the example e belongs to the concept defined by h , that is, when the hypothesis classifies the example as positive. Sometimes the set of all examples covered by a hypothesis h is denoted as $covers(h)$.

Synonyms

matches

See Also

concept-learning; rule-learning; inductive logic programming; generality relation; learning from entailment; learning from interpretations; learning from proofs; hypothesis language

References and recommended reading

1. T.M. Mitchell. Machine Learning. McGraw-Hill. 1997.

Generality Relation

Definition

The generality relation is used in concept-learning, rule-learning and inductive logic programming. It specifies when one hypothesis g is more general than another hypothesis s . It is defined in terms of the covers relation, which specifies for each hypothesis the set of examples it covers or matches. More precisely, a hypothesis g is *more general* than a hypothesis s if g covers all instances that are also covered by s . This is sometimes written as $\text{covers}(s) \subseteq \text{covers}(g)$. The hypothesis g is called a *generalization* of s , and s a *specialization* of g . The generality relation (partially) orders the hypotheses space and can therefore be employed to prune the search for consistent hypotheses. For instance, when a hypothesis h does not cover a positive example e , then no specialization of h will cover e , and hence, all specializations of h can be pruned.

Synonyms

See Also

Logic of Generality; Refinement Operators; Subsumption; Minimally general generalization; least general generalization

References and recommended reading

1. T.M. Mitchell. Machine Learning. McGraw-Hill. 1997.

Subsumption

Definition

Subsumption is the term used in inductive logic programming to refer to various generality relations between hypotheses in the form of clauses. In propositional logic, one clause g subsumes a clause s if and only if $g \models s$, that is, if and only if, $g \subseteq s$. For instance, the clause `flies :- bird, normal`, which states that normal birds fly, subsumes the clause `flies :- bird, normal, black` because $\{\text{flies}, \neg \text{bird}, \neg \text{normal}\} \subseteq \{\text{flies}, \neg \text{bird}, \neg \text{normal}, \neg \text{black}\}$. While in the propositional case, the subsumption relation is relatively straightforward, it is more involved when working with first order logic as in inductive logic programming. In this case, one typically employs the θ -subsumption relation, which states that a clause g θ -subsumes a clause s if and only if there exists a substitution θ such that $g\theta \subseteq s$. For instance, the clause `father(X,Y) :- male(X), parent(X,Y)` θ -subsumes the clause `father(Z,mary) :- male(Z), parent(Z,mary), female(mary)` because applying $\theta = \{X/Z, Y/\text{mary}\}$ to the first clause yields $\{\text{father}(\text{Z}, \text{mary}), \neg \text{male}(\text{Z}), \neg \text{parent}(\text{Z}, \text{mary})\}$ which is a subset of the second clause $\{\text{father}(\text{Z}, \text{mary}), \neg \text{male}(\text{Z}), \neg \text{parent}(\text{Z}, \text{mary}), \neg \text{female}(\text{mary})\}$.

Synonyms

See also

Logic of generality; inductive logic programming; refinement operator;

References and recommended reading

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.
2. S.-H. Nienhuys-Cheng and R. de Wolf. Foundations of Inductive Logic Programming. Springer, 1997.

Refinement operator

Definition

Many machine learning methods search through a hypothesis space structured by a generality relation. For instance in concept-learning, one searches for a hypothesis that covers all positive and none of the negative examples. These search procedures repeatedly consider candidate hypotheses and refine them to generate successor candidates when needed. The successor candidates are generated using a refinement operator. So, a refinement operator that maps a hypothesis to its set of refinements. More formally, a refinement operator is a function $\rho : \mathcal{L} \rightarrow \mathcal{P}(\mathcal{L})$, where \mathcal{L} is the set of hypotheses and $\mathcal{P}(\mathcal{L})$ denotes its powerset. Usually, one distinguishes two types of refinement operators: specialization operators, which return a set of specializations of the hypothesis, and generalization operators, which return a set of generalizations.

As an illustration of a refinement operator consider the “adding condition rule” of Ryszard Michalski, which specializes rules. For instance, consider the rule

```
class = + IF outlook = sunny AND temperature = high
```

it can be specialized by adding conditions to yield, for instance,

```
class = + IF outlook = sunny AND temperature = high AND humidity = low
```

Various types of refinement operators have been studied in an inductive logic programming context.

See also

Logic of generality; inductive logic programming

References and recommended reading

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.
2. R. S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20(2):111–161, 1983.

Learning from entailment

Definition

Learning from entailment is an inductive logic programming setting that is also relevant to statistical relational learning. In this setting, both hypotheses and instances are logical formulae, typically definite clauses, which underlie the programming language Prolog. Furthermore, when learning from entailment, a hypothesis h covers an instance e if and only if $h \models e$, that is, when h logically entails e , or equivalently, when e is a logical consequence of h . For instance, consider the hypothesis h :

```
flies :- bird, normal.  
bird  :- blackbird.  
bird  :- ostrich.
```

The first clause or rule can be read as `flies` **if** `normal` **and** `bird`, that is, normal birds fly. The second and third state that blackbirds, resp. ostriches, are birds. Consider now the examples e_1 :

```
flies :- blackbird, normal, small.
```

and e_2 :

```
flies :- ostrich, small.
```

Example e_1 is covered by h , because it is a logical consequence of h , that is $h \models e_1$. On the other hand, example e_2 is not covered, which we denote as $h \not\models e_2$.

Learning from entailment is often employed in a concept-learning context, where the goal is to learn a hypothesis that covers all positive and none of the negative examples. It can also be employed when in statistical relational learning.

Synonyms

See also

Logic of generality, inductive logic programming, hypothesis language; statistical relational learning; entailment.

References and recommended reading

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.
2. L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.

Learning from interpretations

Definition

Learning from interpretations is an inductive logic programming setting that is also relevant to statistical relational learning. When learning from interpretations, hypotheses are logical formulae, typically definite clauses, which underlie the programming language Prolog, and instances are logical interpretations. For propositional theories, interpretations are assignments of truth-values to propositional variables. For instance, when describing birds, two possible interpretations could be

```
{ostrich, small}
{blackbird, bird, normal, flies}
```

where we specify interpretations through the set of propositional variables that are true. This means that in the first interpretation the only true propositions are `ostrich` and `small`. An interpretation specifies a kind of possible world. A hypothesis h then covers an interpretation if and only if the interpretation is a model for the hypothesis. An interpretation is a model for a hypothesis if it satisfies all clauses in the hypothesis. Consider now the hypothesis h consist of the following clauses.

```
flies :- bird, normal.
bird  :- blackbird.
bird  :- ostrich.
```

The first clause or rule can be read as flies **if** normal **and** bird, that is, normal birds y. The second and third state that blackbirds, resp. ostriches, are birds. the second interpretation is a model for the theory h but the first is not. Because the condition part of the rule `bird :- ostrich.` is satisfied in the first interpretation (as it contains `ostrich`), the conclusion part, that is `bird`, should also belong to the interpretation in order to have a model. Thus, the first example is covered by the theory h , but the second is not.

Learning from interpretations is often employed in a concept-learning context, where the goal is to learn a hypothesis that covers all positive and none of the negative examples. It can also be employed in statistical relational learning.

Synonyms

See also

Logic of generality, inductive logic programming, hypothesis language; statistical relational learning;

References and recommended reading

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.
2. L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.

Learning from proofs

Definition

Learning from proofs is an inductive logic programming setting that is also relevant to statistical relational learning and inductive programming and trace-based programming. When learning from proofs, hypotheses are logical formulae, typically definite clauses, which underlie the programming language Prolog, and instances are logical proofs. A hypothesis h then covers an instance if and only if the instance is a proof in the theory h .

Consider the following theory:

```
nat(0) .  
nat(s(X)) :- nat(X) .
```

where the first clause states that 0 is a natural number and the second one, that if, X is a natural number then also $s(X)$ is a natural number. Given this theory, the example

```
:-nat(s(s(0))) --- :-nat(s(0)) --- :-nat(0) --- []
```

is a legal proof, but the example

```
:-nat(s(s(5))) --- :-nat(s(5)) --- :-nat(5) --- []
```

is not because the last step is invalid (where the proofs are shown as SLD-refutation proofs).

Learning from proofs is often employed in an inductive logic programming context where the goal is to learn a hypothesis that covers all positive and none of the negative examples. It can also be employed in statistical relational learning and inductive programming.

Synonyms

learning from traces.

See also

Logic of generality, inductive logic programming, hypothesis language; statistical relational learning; trace-based programming; inductive programming.

References and recommended reading

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.

Minimally general generalization

Definition

One hypothesis is said to be more general than another one if the former hypothesis covers all instances that are covered by the later one. The resulting generality relation typically partial orders the hypothesis space, that is, the generality relation is reflexive, anti-symmetric and transitive. One cautious strategy for learning performs cautious generalization by repeatedly computing the minimally general generalizations of hypotheses. The minimally general generalizations of h_1 and h_2 are the most specific hypotheses that are more general than h_1 and h_2 . Using the notation $g \preceq s$ to denote that g is more general than s , the set of minimally general generalizations $mgg(h_1, h_2)$ is defined as follows:

$$mgg(h_1, h_2) = \max\{h \in \mathcal{L} \mid h \preceq h_1 \text{ and } h \preceq h_2\}$$

where \mathcal{L} denotes the hypothesis language and \max denotes the most specific elements of the sets.

As one example of a minimally general generalizations consider the hypothesis language consisting of strings and as generality relation the substring relation. A string $s_0 \dots s_n$ is a substring of a string $t_0 \dots t_k$ if and only if $\exists j : s_0 = t_j$ and \dots and $s_n = t_{j+n}$. For instance, `achin` is a substring of `machine learning`. The minimally general generalizations of `abcdefabc` and `defabcdef` are `defabc` and `abcdef`.

If it is guaranteed that there exists a unique minimally general generalization, then one talks about the least general generalization.

Synonyms

See also

least general generalization, logic of generality, inductive logic programming

References and recommended reading

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.
2. T. Mitchell. Machine Learning. McGraw-Hill. 1997.

Least general generalization

Definition

One hypothesis is said to be more general than another one if the former hypothesis covers all instances that are covered by the later one. The resulting generality relation typically partial orders the hypothesis space, that is, the generality relation is reflexive, anti-symmetric and transitive. One cautious strategy for learning performs cautious generalization by repeatedly computing the least general generalization of hypotheses. The least general generalization of two hypotheses h_1 and h_2 is the unique most specific generalization of h_1 and h_2 , that is, it is the least upper bound of h_1 and h_2 in the partial order. The least general generalization of two hypotheses does not always exist. Also, in case that the most specific generalizations are not unique, one talks about the minimally general generalization.

For instance, when the hypotheses h_1 and h_2 are propositional conjunctions, then their least general generalization is the intersection of the conjunctions. Indeed, consider the hypotheses

```
red and large and square and filled
blue and large and square and closed
then the least general generalization is
large and square
```

The least general generalization is especially known in the context of inductive logic programming, where it is used to refer to Plotkin's operation for computing the least general generalization of two clauses under θ -subsumption.

Synonyms

See also

least general generalization, logic of generality, generality relation

References and recommended reading

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.
2. T. Mitchell. Machine Learning. McGraw-Hill. 1997.

Resolution

Definition

Resolution is a logical inference rule used in theorem provers and logic programming to perform deductive inference. The resolution inference rule starts from two clauses and generates a so-called resolvent, that is, a clause that is logically entailed by the two clauses. For propositional logic, the resolution rule can be written as follows:

$$\frac{h \leftarrow g, a_1, \dots, a_n \text{ and } g \leftarrow b_1, \dots, b_m}{h \leftarrow b_1, \dots, b_m, a_1, \dots, a_n} \quad (17)$$

or when writing clauses as disjunctions

$$\frac{h \vee \neg g \vee \neg a_1 \vee \dots \vee \neg a_n \text{ and } g \vee \neg b_1 \vee \dots \vee \neg b_m}{h \vee \neg b_1 \vee \dots \vee \neg b_m \vee \neg a_1 \vee \dots \vee \neg a_n} \quad (18)$$

For instance, consider the clauses

```
flies :- bird, normal.  
bird  :- blackbird.
```

where the first clause states that normal birds fly and the second one that blackbirds are bird. The resolvent of these two clauses is

```
flies :- blackbird, normal.
```

and is therefore logically entailed by the two clauses.

Even though we only introduced resolution for propositional logic, there exists also a (more involved) version of resolution for first order logic.

Synonyms

See also

Logic of generality, entailment, inverse resolution

References and recommended reading

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.
2. P. Flach. Simply Logical. Wiley. 1994.
3. S. Russell and P. Norvig. Artificial Intelligence: a Modern Approach. 2nd Edition. Prentice Hall.

Inverse Resolution

Definition

Inverse resolution is, as the name indicates, a rule that inverts resolution. This follows the idea of induction as the inverse of deduction formulated in logic of generality. The resolution rule is the best-known deductive inference rule, used in many theorem provers and logic programming systems. Resolution starts from two clauses and derives the resolvent, a clause that is entailed by the two clauses. This can be graphically represented using the following schema (for propositional logic).

$$\frac{h \leftarrow g, a_1, \dots, a_n \text{ and } g \leftarrow b_1, \dots, b_m}{h \leftarrow b_1, \dots, b_m, a_1, \dots, a_n} \quad (19)$$

Inverse resolution operators, such as *absorption* and *identification*, invert this process. To this aim they typically copy assume the resolvent is given together with *one* of the original clauses and then derive the missing clause. This leads to the following two operators, which start from the clauses below and induce the clause above the line.

$$\frac{h \leftarrow g, a_1, \dots, a_n \text{ and } g \leftarrow b_1, \dots, b_m}{h \leftarrow b_1, \dots, b_m, a_1, \dots, a_n \text{ and } g \leftarrow b_1, \dots, b_m} \quad (20)$$

$$\frac{h \leftarrow g, a_1, \dots, a_n \text{ and } g \leftarrow b_1, \dots, b_m}{h \leftarrow b_1, \dots, b_m, a_1, \dots, a_n \text{ and } h \leftarrow g, a_1, \dots, a_n} \quad (21)$$

The operators are shown here only for the propositional case, as the first order case is more involved at it requires one to deal with substitutions as well as inverse substitutions.

As one example, consider the clauses

- (1) `flies :- bird, normal.`
- (2) `bird :- blackbird.`
- (3) `flies :- blackbird, normal.`

Here, (3) is the resolvent of (1) and (2). Furthermore, starting from (3) and the clause (2) the absorption operator would generate (1), and starting from (3) and (1) the identification operator would generate (2).

Synonyms

See also

Logic of generality, resolution,

References and recommended reading

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.
2. S. Muggleton. Duce, an oracle based approach to constructive induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 287–292. Morgan Kaufmann, 1987.
3. S. Muggleton and W. Buntine. Machine invention of first order predicates by inverting resolution. In *Proceedings of the 5th International Workshop on Machine Learning*, pages 339–351. Morgan Kaufmann, 1988.

Entailment

Definition

The term entailment is used in the context of logical reasoning. Formally, a logical formula H entails a formula c if and only if all models of H are also a model of c . This is usually denoted as $H \models c$ and means that c is a logical consequence of T or that c is implied by T .

Let us elaborate this definition for propositional clausal logic, where the formulae are expressions such as:

```
flies :- bird, normal.  
bird :- blackbird.  
bird :- ostrich.
```

Here, the first clause or rule can be read as flies **if** normal **and** bird, that is, normal birds fly, the second and third one stating that blackbirds, resp. ostriches, are birds. An interpretation is then an assignment of truth-values to the propositional variables. For instance, for the above domain

```
{ostrich, bird}  
{blackbird, bird, normal}
```

are interpretations, specified through the set of propositional variables that are true. This means that in the first interpretation the only true propositions are `ostrich` and `bird`. An interpretation specifies a kind of possible world. An interpretation I is then a model for a clause $h : -b_1, \dots, b_n$ if and only if $\{b_1, \dots, b_n\} \subseteq I \rightarrow h \in I$ and it is model for a clausal theory if and only if it is a model for all clauses in the theory. Therefore, the first interpretation above is a model for the theory, but the second one is not because the interpretation is not a model for the first clause (as $\{\text{bird, normal}\} \subseteq I$ but $\text{flies} \notin I$). Using these notions it can now be verified that the clausal theory T above logically entails the clause

```
flies :- ostrich, normal.
```

because all models of the theory are also a model for this clause.

In machine learning, the notion of entailment is used as a covers relation in inductive logic programming, where hypotheses are clausal theories, instances are clauses, and an example is covered by the hypothesis when it is entailed by the hypothesis.

Synonyms

logical consequence, implication

See also

Learning from entailment, learning from interpretations, logic of generality, inverse entailment.

References and recommended reading

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.
2. P. Flach. Simply Logical. Wiley. 1994.
3. S. Russell and P. Norvig. Artificial Intelligence: a Modern Approach. 2nd Edition. Prentice Hall.

Inverse entailment

Definition

Inverse entailment is a generality relation in inductive logic programming. More specifically, when learning from entailment using a background theory B , a hypothesis H covers an example e , relative to the background theory B if and only if $B \wedge H \models e$, that is, the background theory B and the hypothesis H together entail the example (see entailment). For instance, consider the background theory B :

```
bird :- blackbird.  
bird :- ostrich.
```

and the hypothesis H :

```
flies :- bird.
```

Together $B \wedge H$ entail the example e :

```
flies :- blackbird, normal.
```

This can be decided through deductive inference. Now when learning from entailment in inductive logic programming, one starts from the example e and the background theory B and the aim is to induce a rule H that together with B entails the example. Inverting entailment now refest on the observation that $B \wedge H \models e$ is logically equivalent to $B \wedge \neg e \models \neg H$, which in turn can be used to compute a hypothesis H that will cover the example relative to the background theory. Indeed, the negation of the example is $\neg e$:

```
blackbird.  
normal.  
:-flies.
```

and together with B this entails $\neg H$:

```
bird.  
:-flies.
```

The principle of inverse entailment is typically employed to compute the bottom clause, which is the most specific clause covering the example under entailment. It can be computed by generating the set of all facts (true and false) that are entailed by $B \wedge \neg e$ and negating the resulting formula $\neg H$.

Synonyms

See also

logic of generality, entailment, bottom clause, inductive logic programming

References and recommended reading

1. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13(3-4):245286, 1995.

Bottom Clause

Definition

The bottom clause is a notion from the field of inductive logic programming. It is used to refer to the most specific hypothesis covering a particular example when learning from entailment. When learning from entailment, a hypothesis H covers an example e relative to the background theory B if and only if $B \wedge H \models e$, that is, B together with H entails the example e . The bottom clause is now the most specific clause satisfying this relationship w.r.t the background theory B and a given example e .

For instance, given the background theory B

```
bird :- blackbird.  
bird :- ostrich.
```

and the example e :

```
flies :- blackbird, normal.
```

the bottom clause is H

```
flies :- bird, blackbird, normal.
```

The bottom clause can be used to constrain the search for clauses covering the given example because all clauses covering the example relative to the background theory should be more general than the bottom clause. The bottom clause can be computed using inverse entailment.

Synonyms

Saturation, starting clause

See also

See also

logic of generality, entailment, inverse entailment, inductive logic programming

References and recommended reading

1. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13(3-4):245286, 1995.
2. L. De Raedt. *Logical and Relational Learning*. Springer. 2008.

Predicate invention

Definition

Predicate invention is used in inductive logic programming to refer to the the automatic introduction of new relations or predicates in the hypothesis language. Inventing relevant new predicates is one of the hardest tasks in machine learning, because there are so many possible ways to introduce such predicates and because it is hard to judge their quality. As one example, consider a situation where the predicates `fatherof` and `motherof` are known. Then it would make sense to introduce a new predicate that is true whenever `fatherof` or `motherof` is true. The new predicate that would be introduced this way corresponds to the `parentof` predicate. Predicate invention has been introduced in the context of inverse resolution.

Synonyms

See also

logic of generality, inductive logic programming.

1. L. De Raedt. Logical and Relational Learning. Springer. 2008.
2. S. Muggleton. Duce, an oracle based approach to constructive induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 287–292. Morgan Kaufmann, 1987.
3. S. Muggleton and W. Buntine. Machine invention of first order predicates by inverting resolution. In *Proceedings of the 5th International Workshop on Machine Learning*, pages 339–351. Morgan Kaufmann, 1988.