

## Introducing Inductive Logic Programming

- Inductive Logic Programming for concept learning
  - » basic definitions
  - » ILP as a search
  - » version space and generality relation
- Computational approaches
  - » top-down (general to specific)
  - » bottom-up (specific to general)
- Learning paradigms
  - » learning from interpretation
  - » learning from entailment

© Alessandra Russo

Unit 3 – Introducing ILP, slide 1

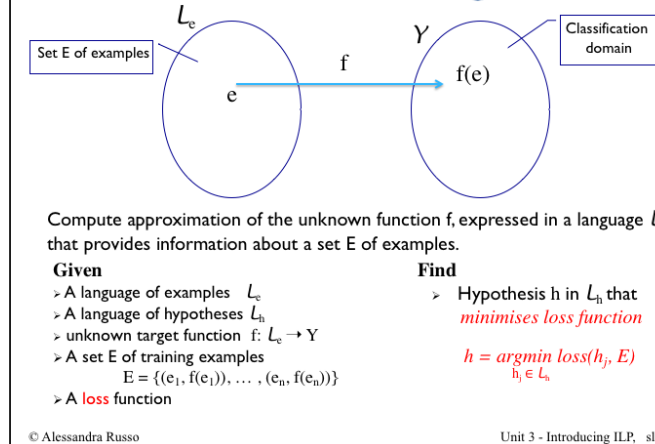
In this lecture we introduce the notion of Inductive Logic Programming. We will start from the task of concept learning as it relates more easily to the notions of data mining and classification, which you might have already encountered in other Machine Learning courses.

ILP can be seen as a search problem. In **concept learning** the task is to compute the definition of a concept, expressed in a given language (i.e. hypotheses space), that satisfies all examples labelled as positive and none of the examples labelled as negative in a given dataset. For instance, in the case of learning the concept of mutagenicity, the task was to find a definition that is able to determine which molecules in a given dataset are active mutagenic or which are not. How does this relate to a more general Machine Learning (ML) task? A ML problem is also a search problem. Its goal is to find solutions, in a given search space, that would minimise a *loss function*. In the first lecture we have stated that Inductive Logic Programming is a subset of Machine Learning, where prior knowledge and solutions are expressed in a declarative (logic-based) language. Being itself a Machine Learning, the ILP task can then be seen as search problem for an hypothesis, expressed in logic, that would *minimise a loss function*. So the question is how we can capture this aspect of search for solutions that minimises a loss function in terms of inductive logic inference? We will see that key to the search problem is the notion of “a more general than” relation between possible hypotheses. This notion (relation) can be used to prune the search and guide it towards solutions that are as close as possible to the “expected solution” and for which a loss function will have a minimal value.

In this lecture we will first introduce the notion of ILP as a search problem and see how its definition relates to the notions of *coverage* and *loss functions* used in Data Mining and other forms of Machine Learning. Within this context it is natural to think about different search mechanisms. We will then present, in general terms, key *search mechanisms* that have lead to the development of different ILP computational approaches, such as for instance bottom-up and top-down approaches. We will explore in the next lectures examples of ILP algorithms for each of these categories.

*NOTE: this set of slides is based on material by Nada Lavrac on “Introduction of ILP” and Hendrik Blocke on “From Machine Learning to Inductive Logic Programming”.*

## A Machine Learning Task



Let's assume that examples (or instances) are expressed in a language  $L_e$  and that the hypotheses, (pattern or function) that the search task has to compute, are expressed in a language  $L_h$ . In a ML task it is normally assumed the existence of an unknown target function  $f$ , expressed in  $L_h$  that maps elements from  $L_e$  to values in  $Y$ .  $Y$ , in this case, is a classification domain. The task requires also a set  $E$  of examples (e.g. labelled instances) of pairs of elements in  $L_e$  and values that the function should give (in  $Y$ ):

$$E = \{(e_1, f(e_1)), (e_2, f(e_2)), \dots, (e_n, f(e_n))\}$$

**The set  $E$  gives in essence instances of behavior of the unknown function.**

The goal of the learning task is to compute an hypothesis  $H$  that approximates the unknown function  $f$  as well as possible, which means that it minimises a notion of *loss*. The notion of loss is a measure (i.e. a *quality criterion*) of the quality of the learned hypothesis with respect to the given dataset  $E$ .

Minimise the loss function means computing the  $\operatorname{argmin} \operatorname{loss}(h_j, E)$ . In this case the set upon which the  $\operatorname{argmin}$  is computed is the set of possible hypotheses as defined by the language  $L_h$ . We are interested in finding the  $h$  such that the  $\operatorname{loss}(h, E)$  has the global minimum value. Different loss functions can be defined. In a simple classification problem, the domain  $Y = \{0, 1\}$  and a *loss function can be the error of classification over the training set  $E$* :

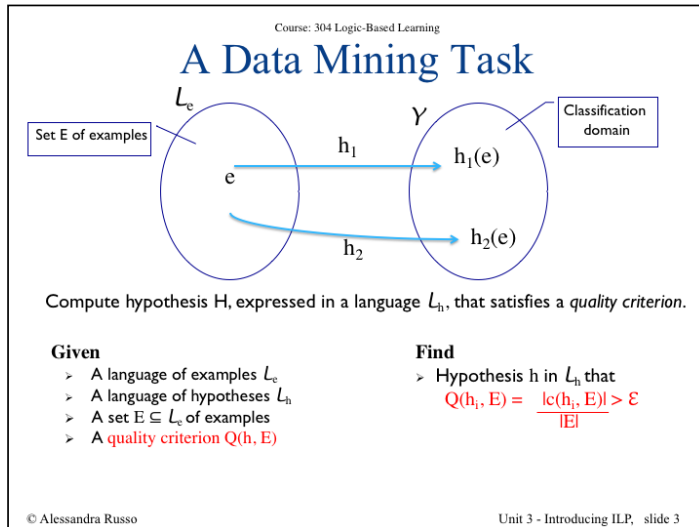
$$\operatorname{loss}(h, E) = (1/|E|) \times \sum (f(e_i) - h(e_i))$$

The loss function expresses the error made in classifying the training data  $E$ . Minimising the loss function means minimising the error made in classifying the training examples.

Different ML techniques assume different domain  $Y$ . In regression, for instance, the domain  $Y$  is the set of real numbers  $R$  and the loss function is the *mean square error*: Minimise the loss function means computing the least mean square error, i.e. minimise the average of the squares of the errors:

$$\operatorname{loss}(E, h) = (1/|E|) \times \sum (f(e_i) - h(e_i))^2$$

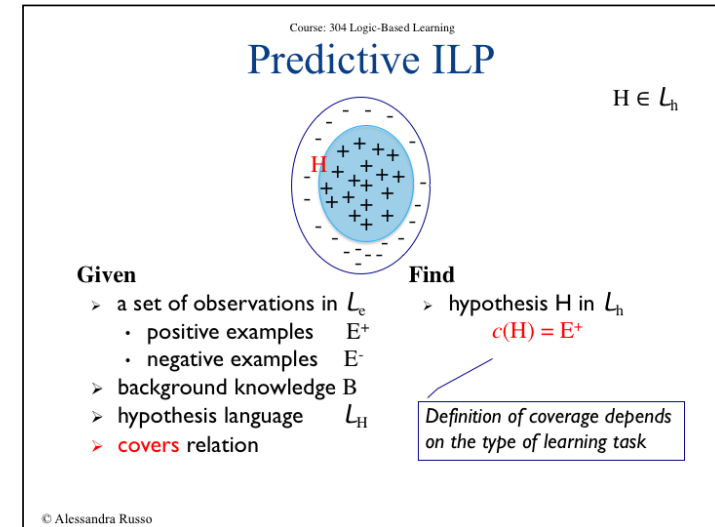
*Learning can therefore be seen as a function approximation problem.*



In a data mining task, the objective is also to *discover hypotheses, expressed in  $L_h$ , that satisfy a given quality criterion  $Q$* . For instance, in the case of mutagenicity, if the task is to learn the concept of active and inactive molecules, the domain  $Y$  could simply be  $Y = \{\text{active, inactive}\}$ , or  $Y = \{\text{true, false}\}$ . The quality criterion, in this case, can be expressed in terms of a notion of coverage, examples that the hypothesis defines as active (or assigns value 1). Using the notion of coverage as quality criterion is typical when **learning the definition of a concept**. We will refer to the set of examples covered by a given hypothesis  $h$ , with the symbol  $c(h)$  (or  $c(h, E)$ ). A typical definition of quality criterion in data mining is relative frequency:

$$Q(h, E) = |c(h, E)| / |E|.$$

So, a good hypothesis would be an hypothesis for which the relative frequency is above a given threshold value  $\varepsilon$ .



If we consider the Boolean values (true and false) of the classification domain and we superimpose the classification domain with the set  $E$  of examples, the subset of  $E$  with + elements refers to the example instances to which the learned function  $h_i$  assigns value 1 and the subset  $E$  with - elements refers the example instances to which the learned function  $h_i$  assigns value 0. We can see then that predictive ILP can be seen as the usual machine learning binary classification problem that applies to other predictive learning tasks such as decision trees, SVM, Bayesian Classifiers, etc.

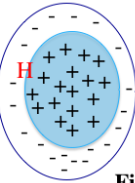
A *predictive learning task*, in fact, assumes a given (possibly empty) background knowledge, a set of observations (or instances) that include positive and negative examples, a language for the search space and a notion of **covers relation**. The outcomes are hypotheses expressed in the given search space language that correctly classify the observations by covering the positive examples and rejecting the negative examples. The notions *covers* and *rejects* depend on the particular type of learning task.

In ILP the languages  $L_e$  and  $L_h$  are normally from the same type of logic-based language. The notion of covers relation depends on whether the learning task is learning from entailment, learning from interpretation, induction of stable models, brave induction, cautious induction or learning from answer sets. We will refer to these different ILP paradigms as we go along in the course.

Course: 304 Logic-Based Learning

## Predictive ILP

$H \in L_h$



**Given**

- > a set of observations in  $L_e$ 
  - positive examples  $E^+$
  - negative examples  $E^-$
- > background knowledge  $B$
- > hypothesis language  $L_h$
- > **covers** relation
- > **quality** criterion

**Find**

- > hypothesis  $H$  in  $L_h$

$Q(h_i, E) = \max |c(h_i, E)|$

© Alessandra Russo

The covers relation can also be accompanied by a quality criterion. For instance, an ILP task may not necessarily be able to compute hypothesis that cover all given positive examples and none of the negative examples. This is for instance the case when data in  $E$  are noisy.

In this case an additional quality criterion can be used such as for instance maximise the number of positive examples covered, or **maximise the test accuracy**, also called precision test. This means the ratio of the number of true positive covered over the sum of the number of true positive covered and the number of false positive covered ( $TP/TP+FP$ ). In general, there can be different quality criteria to assess the goodness of a learnt hypothesis. To name some, these criteria include sensitivity (ratio of positive outcomes out of the real true positive set –  $TP/TP+FN$ ), specificity (ratio of negatives outcomes out of the real false negative set –  $TN/TN+FP$ ), precision (ratio of true positive out of the positive outcomes), and accuracy, which defines the proportion of trues results out of the total number of cases –  $TP+TN/TP+FN+TN+FP$ ). All these quality criteria are particularly important when we want to learn from positive and negative examples.

We will discuss these notions in more details later in the course, when we will refer to scoring function of hypotheses and other forms of heuristics that existing ILP algorithms adopt during the search for an hypothesis in order to compute the “best” hypothesis.

Course: 304 Logic-Based Learning

## ILP as search of program clauses

Languages  $L_e$  and  $L_h$  are logic-based languages.

*Searching an hypothesis in the hypothesis space means finding hypothesis that satisfies the covers relation and quality criterion (if needed).*

**Naïve generate-and-test algorithm**

For all  $H \in L_h$  do  
 if  $H$  satisfies the covers relation and maintains or improves the quality criterion, then output  $H$

Computationally expensive as the whole search space could in principle be explored.

**We need to structure the search process**

© Alessandra Russo

Unit 3 - Introducing ILP, slide 6

In many machine learning cases and data mining, the languages ( $L_e$ ) and ( $L_h$ ) are different. Logic-based learning, on the other hand, uses the same logical formalism for representing examples ( $L_e$ ), hypothesis ( $L_h$ ) and for expressing the notion of covers relation.

For this and the next unit we will assume that the type of hypothesis that we are interested to learn are (set of) definite clauses. Later we will extend the logical formalism to normal clauses and ASP.

Given a language  $L_h$ , which defines the hypothesis space, and given a covers relation, with possibly a quality criterion, learning can be seen as a search problem. The question is how to make this search problem computationally feasible. A very simplistic approach would be that of “generate-and-test”. The idea would be to enumerate the full set of hypotheses, as defined by the language  $L_h$  and pick each hypothesis, one by one, and test it with respect to the covers relation and the quality criterion. If the covers relation is satisfied and the quality criterion maintained or improved then the hypothesis is returned by the algorithm, otherwise a next hypothesis is considered.

This approach is clearly very naïve. It requires first that the set of hypothesis is enumerable and finite (for the algorithm to terminate) and moreover, it is very inefficient as the algorithm could in principle search the whole space.

What we need is identify some mechanisms that allow us to structure the search process, prune parts that are clearly irrelevant and adopt mechanisms that allow us to make the best choices so to get closer, in the search, to the hypothesis solutions. To understand how this can be done, let’s start with the case of concept learning and *version space model*.

Course: 304 Logic-Based Learning

## Concept Learning and Version Space

Infer the general definition of a concept given examples labelled as members or nonmember of that concept.

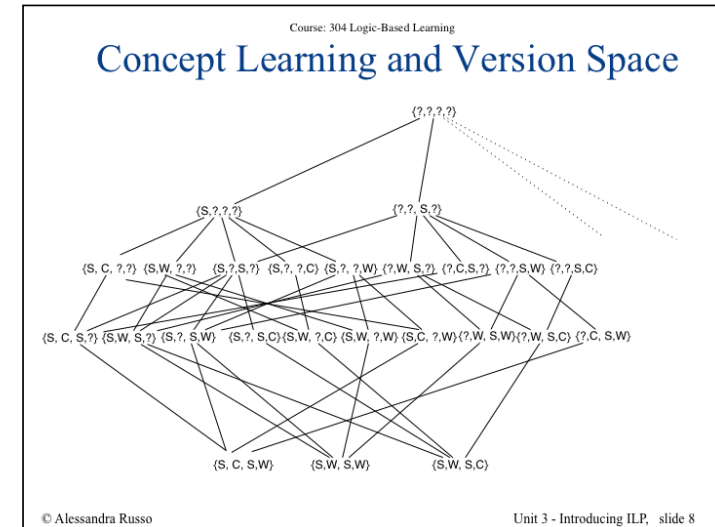
Examples (E)

Sky	Temp	Wind	Water	EnjoySport
Sunny	Warm	Strong	Warm	Yes
Sunny	Warm	Strong	Cold	Yes
Sunny	Cold	Weak	Cold	No
Sunny	Cold	Strong	Warm	Yes
Rainy	Cold	Strong	Cold	No

Concept to learn:  
What are the days my friend enjoys his favorite water sports?

© Alessandra Russo Unit 3 - Introducing ILP, slide 7

Concept learning is a type of logic-based learning that takes as instance space a set of negative and positive instances of the concepts that we are interested to learn. The unknown concept is therefore a subset of the set of instances. The representation language for the instance space and for the hypothesis to learn can be either defined in an extensional way (see table in the slide) or in an intentional way, i.e. through rules. In the above slide, the table includes part of the set of instances represented in an extensional manner. Those that have “yes” under the category “Enjoy Sport”, are positive examples of the concept that we want to learn and the rest are negative examples. In this case the concept that we want to learn is “What are the days my friend enjoys his favorite water sports?” You can think of an implicit representation for the above table: each row could be seen as a rule whose head predicate is the propositional variable *enjoySports* and body conditions are propositional letters corresponding to the possible values of the parameters (e.g. *sunny*, *rainy*, *cold*, etc.). The concept to learn is a subset of the instance space. So in this case the language of the hypothesis is the same as the language of the instance space. The “covers” relation  $c(h, E)$  is the set of examples (rows in the tables) that are covered by the hypothesis  $h$ . For instance, let’s assume that the hypothesis  $h$  is given by  $\text{Sky}=\text{Sunny}$ . You could easily see that in this case the hypothesis will cover not only the positive examples represented in rows 1, 2 and 4, but also the negative example represented in row 3. On the other hand, if the hypothesis was  $\{\text{Sky}=\text{Sunny and Wind}=\text{Strong}\}$  then only the positive examples in rows 1, 2 and 4 would be covered and the negative example in row 3 will no longer be covered. Alternative hypothesis of the form  $\{\text{Sky}=\text{Sunny, Temp}=\text{Warm, Wing}=\text{Strong}\}$  would cover the positive example expressed in row 2, which is already covered by the previous hypothesis. So this hypothesis is not needed if we had already found the previous hypothesis. Using an implicit representation, the above hypothesis would be written as (i)  $\text{sunny} \rightarrow \text{enjoySports}$ , (ii)  $\text{sunny} \wedge \text{strong} \rightarrow \text{enjoySport}$ , and (iii)  $\text{sunny} \wedge \text{warm} \wedge \text{strong} \rightarrow \text{enjoySport}$ . Hypothesis (ii) is said to subsume hypothesis (iii) as it covers already all the positive examples that hypothesis (iii) covers and possibly some more. Similarly, hypothesis (i) is said to subsume hypothesis (ii), and therefore hypothesis (iii).



Given the language of our hypothesis, the concept learning task defined in the previous slide induces a version space. This is the set of all hypothesis that are *consistent* with the given positive examples, in simple terms all hypothesis that might be **correct target concepts** for the given set of positive examples. The diagram given in this slide is only a partial definition of the full version space for the example given in the previous slide.

The hypothesis  $\{S, ?, ?, ?\}$ , which would correspond, in implicit form, to the rule  $\text{sunny} \rightarrow \text{enjoySports}$ , covers also cases  $\{S, C, S, C\}$  which is a negative example. So this hypothesis is too *general* and its level of precision (or test accuracy) would be  $\frac{3}{4}$  (3 positive example covered over a total of 4 positive and negative examples classified as *enjoySports*). Going down the version space the parent node becomes more specific. For instance  $\{S, W, ?, ?\}$  is more specific than  $\{S, ?, ?, ?\}$ , in the sense that it specifies the value of the second attribute when in the first case this was unspecified. For this second hypothesis the level of precision would be  $\frac{2}{2} = 1$  as it covers two true positive examples and does not have any false positive (i.e. negative examples covered). So this second hypothesis would in principle be preferred to the first hypothesis. The second hypothesis  $\{S, W, ?, ?\}$  corresponds, in implicit form, to the rule  $\text{sunny} \wedge \text{warm} \rightarrow \text{enjoySports}$ . In this case, by adding an extra condition (i.e. warm), we have been able to eliminate the coverage of a negative example, but at the same time we have also eliminated the coverage of a positive example (row 4 in the table). The second hypothesis is called a specialisation of the first hypothesis. We will see later in this lecture that specialisation is a mechanism that allows the elimination of covered unwanted negative examples. But the hypothesis should not be specialised too much as it risks to eliminate also the coverage of true positive examples.

What is important to understand here is the fact that, given an hypothesis language, we can express the version space of the possible hypothesis and navigate through this space in the search for the “best” hypothesis. How do we navigate from one node to another, using logical inference? And what do we mean by “best” hypothesis.

## Generality in Version Space

- Central to search over a version space is the notion of **generality**:  
 $h$  is more general than  $h'$  ( $h \succcurlyeq h'$ ) iff  $c(h', E) \subseteq c(h, E)$
- Version Space is limited by its
  - top** ( $\{ \text{all examples covered} \}$ )
  - bottom** ( $\{ \text{just positive examples covered} \}$ )
- Keeping the entire Version Space is not feasible. Most concept learners try to find one hypothesis in the version space.

Generality property can be used for search

- ❖ If  $h$  covers negative example, then any  $g \succcurlyeq h$  also covers negatives
- ❖ If  $h$  does not cover some positives, then any  $s \preccurlyeq h$  does not cover those positive examples either.

Central to any search algorithm over version spaces is the notion of **generality**. An hypothesis  $h$  is said to be more general than another hypothesis  $h'$  if and only if all (positive) examples covered by the second hypothesis are already included in (positive) examples covered by the first hypothesis. For instance considering again the version space given in the previous slide, the hypothesis  $\{S, ?, ?, ?\}$  is more general than the hypothesis  $\{S, W, ?, ?\}$ . We write  $\{S, ?, ?, ?\} \succcurlyeq \{S, W, ?, ?\}$ . In fact the examples covered by  $\{S, W, ?, ?\}$ , which are the first two rows in the table, are already covered by the hypothesis  $\{S, ?, ?, ?\}$ .

The *generality relation* over hypothesis gives the version space the shape of a lattice, whose bottom element is (are) the hypothesis (hypotheses) that covers just the given positive examples and the top node is the empty hypothesis that covers all examples (basically inconsistency – anything can be true and false). In principle, given an hypothesis language, we could generate the entire version space, i.e. the entire set of possible hypothesis, and search for a solution by navigating over this space. But this is not computationally feasible. You will not be asked to generate the version space.

We can search for the correct hypothesis without constructing the full version space, but just trying to search for the hypothesis using the underlying principle of generality. We could start from an hypothesis  $h$ . If  $h$  covers a negative example than any other hypothesis  $g$  that is more general than  $h$  will also cover the same negative example. Vice-versa, if an hypothesis  $h$  does not cover a positive example than any hypothesis  $g$  that specializes  $h$  will also not cover the same positive example. These two properties allow us to prune, or direct, the search along the right paths in the version space. For instance, the hypothesis  $\{S, C, ?, ?\}$  does not cover the first two rows (positive examples). Any other specialisation hypothesis of this one will also not cover the same positive examples. For instance  $\{S, C, S, ?\}$  will also not cover the first two rows in the table. So it is not necessary to move the search to its children nodes in the lattice.

NOTE: These two properties of the generality principle hold when the learning task is expressed within the context of definite clauses. We will see later in the course that when we want to learn normal logic programs (set of normal clauses) the search becomes much more complicated as the above properties no longer apply. We will look at other methods for searching hypothesis.

## ILP as search of program clauses

ILP learner can be described by

### ➤ Structure of the hypothesis space

based on generality relation:

$h$  is more general than  $h'$  iff  $\text{covers}(h') \subseteq \text{covers}(h)$

### ➤ Search strategy (or pruning strategy)

- Uninformed strategy (depth-first, random walk, etc.)
- Heuristic search

If  $h$  does not cover  $e$  then no specialisation  $h'$  covers  $e$   
*used with positive examples for pruning*  
 If  $h$  does cover  $e$  then all generalisations  $h'$  will also cover  $e$   
*used with negative examples for pruning*

### ➤ Heuristics

- for directing search
- for stopping search (quality criterion)

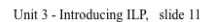
To define an ILP learner as a search of program clauses we can make use of 3 main components: (i) a notion of structure of hypothesis space, (ii) a search strategy and (iii) heuristics.

In the context of definite clauses, the structure of hypothesis space is characterised by a generality relation and the structure that this relation induces is that of a lattice.

Given the property of the structure, search strategies can be defined. In the case of generality relation the strategy takes into account that if an hypothesis in the lattice does not cover a example, then any of its specialisation hypothesis will also not cover the example. If it is a positive example that is not covered by a current hypothesis, then we can use this property to prune all the specialisations of this current hypothesis (as shown in the following few slides) and not search along this part of the lattice.

On the other hand, if a current hypothesis does cover a negative example then all its generalisations will also cover the same negative example. So this allows us to prune all its generalisation hypotheses as none of these hypotheses would have a better coverage.

The heuristic aspect of an ILP learner is what decides when to stop. In the case of learning with no noise in the examples, such stopping criteria is when an hypothesis is found that covers all the positive examples and none of the negative examples. We will see later a more formal definition of a learning task for definite clauses.



- 1) For each positive example  $e$ 
  - i. we would find in  $S$  those hypotheses that do not cover the example and apply one step generalisation
  - ii. we would find in  $G$  all those hypotheses that do not cover the positive example and throw them away.
- 2) For each negative example  $e$ 
  - i. we would find in  $S$  those hypotheses that cover it and throw them away.
  - ii. we would find in  $G$  all those hypotheses that cover  $e$  and apply a specialisation step.

Unit 3 - Introducing ILP, slide 12



## Generality of Theories

Let C and D be two definite clauses.

**C more general than D if and only if  $C \models D$**

e.g. "all fruit tastes good"  $\models$  "all apples taste good"

### Subsumption

Let C and D be two definite clauses.

**C subsumes D if and only if there exists a  $\theta$  such that  $C\theta \subseteq D$**

e.g.

$p(X, Y) \leftarrow r(Y, X)$  subsumes  $p(a, b) \leftarrow r(b, a)$  ( $\theta = \{X/a, Y/b\}$ )

$p(X, Y) \leftarrow r(Y, X)$  subsumes  $p(X, Y) \leftarrow r(Y, X), q(X)$

**If C subsumes D then  $C \models D$**

The generality relation between theories is defined in terms of entailment. A clause C is more general than another clause D if the first entails the second. This means that all the interpretations that makes C true, make also D true. If we think now in terms of coverage of examples, when a clause C covers an example means that C entails the example, i.e. all the interpretations that make C true, make also the example true. So if a clause D covers a negative example  $e^-$ , and a clause C entails D (i.e. C is more general than D), then C will cover the negative example  $e^-$  too. This is because for D to cover the negative example, all interpretation that satisfy D will also satisfy the example  $e^-$ . Since all interpretations of C are also interpretation of D, if  $e^-$  is true in the intersection of all the interpretations of D,  $e^-$  is also true in the intersection of all the interpretations of C and therefore  $e^-$  is covered by C.

Similarly if a clause C does not cover a positive example  $e^+$ , then also a clause D that is entailed by C will not cover the positive example. This is easy to see. If C does not cover  $e^+$  then there is some interpretation that satisfies C and does not satisfy  $e^+$ . This interpretation will also satisfy D, by definition of entailment, therefore D will not cover  $e^+$  either.

Hence, the two principles for searching hypotheses in a version space, which we have described in the previous slides, apply also in the case when the generality relation between two clauses is the entailment relation. **If an hypothesis is consistent (i.e. does not cover any negative example) then any specialisation of this hypothesis will also be consistent. Similarly if an hypothesis is complete, then every generalisation of this hypothesis will also be complete.**

But how to compute entailment relation? Logical entailment is in general undecidable. So to compute generality relation we use a notion of **subsumption** between clauses, which is decidable. A clause C is said to subsume a clause D if there is a substitution that, applied to C, makes it a subset of the clause D. Remember that although clauses are written as implication rules, they are sets of literals, so the subset operator makes sense.

It is also easy to show that if C subsumes D then C also logically entails D. In the above example of subsumption, it is easy to see that  $r(Y, X) \rightarrow p(X, Y) \models r(b, a) \rightarrow p(a, b)$ . Similarly for the second clause. So the generality relation between clauses can be defined in terms of subsumption.

## ILP as search of program clauses

ILP for definite clauses can be described by

> **Structure of the hypothesis space**

based on generality relation:

$h$  is more general than  $h'$  iff  $h \models h'$

> **Search strategy** (or pruning strategy)

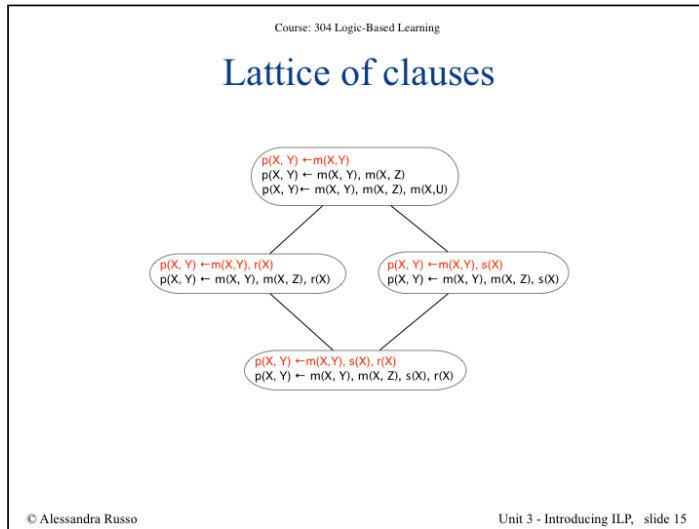
- $\theta$ -subsumption

Set of clauses H subsumes another set H' of clauses if and only if for all clauses  $c' \in H'$  there exists  $c \in H$  such that  $c\theta$ -subsumes  $c'$

**Two types of search traversal over the lattice of hypotheses:**

- specialisation
- generalisation

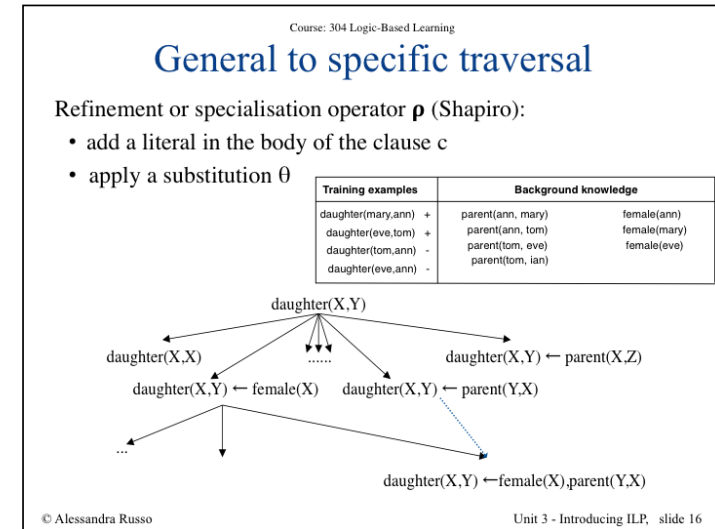
It has been shown that the subsumption relation over definite clauses defines a lattice structure. Details of proofs of this results are omitted from the course. Moving from a node to its parent node means considering a generalisation step, whereas moving from a node to one of its children nodes means considering a specialisation step.



An example of lattice structure given by the subsumption relation between clauses is given in this slide. At each node more than one clause is shown to give an example of clauses that are equivalent. For instance,  $p(X, Y) \leftarrow m(X, Y), m(X, Z)$  is equivalent to  $p(X, Y) \leftarrow m(X, Y)$ . It is possible to find a substitution ( $Z/Y$ ) to make the two clauses identical. The top clause at each of the nodes is often referred to as the “reduced” clause, a clause representative of an equivalence class.

Clauses at different nodes are instead generalisation (from child to parent) and specialisation (from parent to child) clauses. For instance  $p(X, Y) \leftarrow m(X, Y)$  is a generalisation of both  $p(X, Y) \leftarrow m(X, Y), r(X)$  and  $p(X, Y) \leftarrow m(X, Y), s(X)$ . Similarly,  $p(X, Y) \leftarrow m(X, Y), s(X), r(X)$  is a specialisation of both  $p(X, Y) \leftarrow m(X, Y), r(X)$  and  $p(X, Y) \leftarrow m(X, Y), s(X)$ .

Looking for consistent and complete hypothesis means traversing this lattice. This can be done by applying either a generalisation operator (moving upwards in the lattice) or specialisation operator (moving downwards in the lattice). In this first case we say that the search strategy adopted by the ILP learner is a top-down strategy. In the second case we say that the search strategy is a bottom-up strategy. Consequently, ILP learners for definite clauses have been classified into **top-down** and **bottom-up** learners.



A first example of refinement or specialisation operator was proposed by Shapiro in the early '80 which constituted the basis of his Model Inference System (MIS) learning approach. The basic idea is to start from a set of positive and negative examples of a new concept and compute a set of Horn clauses that correctly represent this concept by traversing a lattice (i.e. a special version space) of possible hypothesis by means of a refinement operator.

This operator essentially refines or specialises a clause by adding a literal to the body of the current clause or applying a substitution. For instance, navigating from  $daughter(X, Y)$  to  $daughter(X, X)$  is a result of refining the first clause by applying a substitution  $\{Y/X\}$ , instead moving from  $daughter(X, Y)$  to  $daughter(X, Y) \leftarrow parent(X, Z)$  is a refinement generating by adding a new body literal.

Consider for instance, the example given in the table above. The concept to learn is **daughter**. Starting from the top clause we check the coverage and see that it includes also the negative examples. So the refinement operator generate a specialisation of this clause. The lattice include various possible specialisations. One of these is the clause  $daughter(X, Y) \leftarrow parent(Y, X)$ . In this case we still cover one negative example (i.e.  $daughter(tom, ann)$ ). So a further refinement is required, which leads to the clause  $daughter(X, Y) \leftarrow parent(Y, X), female(X)$  which now covers all the positive examples and none of the negative one.



## Specific to general traversal

For bottom-up search:

- Plotkin's least general generalisation (lgg) of two clauses
- Muggleton's inverse resolution

Two different main operators have been developed in ILP that allow a bottom-up search over the space of hypotheses. These are Plotkin's least general generalisation (lgg) between two clauses and operation of inverse resolution proposed by Muggleton in the '80. The next few slides give a very brief overview of these two operators.

## Specific to general traversal

For bottom-up search:

- Plotkin's least general generalisation (lgg) of two clauses
- Muggleton's inverse resolution

Consider 2 clauses, compute the most specific clause that is more general than both of the given clauses.

- lgg of terms:  $\text{lgg}(a, b) = X, \text{lgg}(f(X), g(Z)) = W, \text{lgg}(f(a, b, a), f(c, c, c)) = f(X, Y, X).$
- lgg of literals:  $\text{lgg}(p(s_1, \dots, s_n), p(t_1, \dots, t_n)) = p(\text{lgg}(s_1, t_1), \dots, \text{lgg}(s_n, t_n))$
- lgg of clauses:  $\text{lgg}(c_1, c_2) = \{\text{lgg}(l_1, l_2) \mid l_1 \in c_1, l_2 \in c_2 \text{ and } \text{lgg}(l_1, l_2) \text{ is defined}\}.$

The Plotkin's least general generalisation operation takes two clauses and tries to compute a third clause that is the most specific of all the clauses that are generalisations of the given initial two clauses.

The lgg is defined first for any pair of terms. So lgg between any pair of constants is a new variable name. The lgg between two terms with the same function is given by a term that has the same function but as argument the lgg of the arguments of the given two terms.

For instance  $\text{lgg}(f(a), f(b)) = f(\text{lgg}(a, b)) = f(X).$

Now that we know what a lgg between two terms is, we can define the lgg between two predicates. This is only defined if the predicates are the same. The lgg is then given by the same predicate but with terms given by the lgg of respective pairs of terms.

We can then define the lgg between two clauses. This is given by the set of lgg of the literals that appear in these two clauses where such an lgg is defined. Consider for instance the two clauses:

$\{f(t, a), \neg p(t, a), \neg m(t), \neg f(a)\}, \{f(j, p), \neg p(j, p), \neg m(j), \neg m(p)\}$

The lgg is given by the set

$\{f(X, Y), \neg p(X, Y), \neg m(X), \neg m(Z)\}$

where

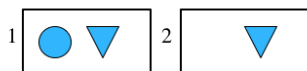
$\text{lgg}(t, j) = X$

$\text{lgg}(a, p) = Y$

$\text{lgg}(t, p) = Z$

## Specific to general traversal

Example:



```
pos(1).
contains(1, o1).
contains(1, o2).
triangle(o1).
points(o1, down).
circle(o2).

pos(2).
contains(2, o3).
triangle(o3).
points(o3, down).
```

Represent examples by clauses:

```
pos(1) ← contains(1, o1),
contains(1, o2), triangle(o1),
points(o1, down), circle(o2).
pos(2) ← contains(2, o3), triangle(o3),
points(o3, down).
```

Most specific clauses  
at the bottom of the  
lattice

```
lgg( pos(1) ← contains(1, o1), contains(1, o2), triangle(o1),
points(o1, down), circle(o2) ,
(pos(2) ← contains(2, o3), triangle(o3), points(o3, down) )
```

```
= pos(X) ← contains(X,Y), contain(X, Z), triangle(Y), points(Y, down)
```

## Specific to general traversal

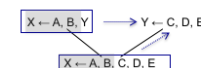
For bottom-up search:

- Plotkin's least generalisation (lgg) of two clauses
- Muggleton's inverse resolution

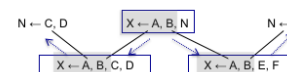
Absorption



Identification



Intra-construction



Inter-construction



Another mechanism for searching hypotheses from specific to general is the use of inverse resolution. Inverse resolution was initially proposed by Professor Muggleton as a mechanism for computing relative generality relation (i.e. generality relation conditional to a given background knowledge) between clauses.

The idea of inverse resolution is, as the name suggests, to compute clauses from given clauses that reflect an inverse operation of the standard resolution rules we have seen in the previous lectures. There are various inverse resolution operators and this slide gives a flavor of some of these in the context of propositional logic. Inverse resolution for first-order define clauses is slightly more complicated as it has to take into account unification between clauses. The second part of this course, taught by Professor Muggleton, might provide further details on this notion of inverse resolution. Briefly, in the propositional case, there are four main operations of inverse resolution.

**Absorption** can be applied to two clauses C1 and C2, such that the second clause C2 includes all the body conditions that define a predicate (Y) in the first clause C1. Then a new clause C3 can be generated from C2 by replacing the body literals in C2 that are used in C1 with the head of the rule C1. Essentially the new clause  $X \leftarrow A, B, Y$  is a generalisation of the first two clauses as it would allow the inference of X by using any other existing definition of Y.

**Identification** can be applied by identifying a set of literals that are common in two clauses and identify a single (leftover) literal (i.e. Y) with a (leftover) set of literals in the second clause (i.e. C, D, E) and generate a new clause that defines this left over literals in terms of the other leftover literals. This rule also generate new knowledge (new definition of Y) that can be used in the existing background knowledge to generate more consequences.

**Intra-construction** involve more than two clauses (i.e. inverse of two parallel steps of resolutions). It identifies a common set of literals in rules with the same head and defines a new concept using the literals that are not common in the used clauses.

**Inter-construction** involve more than two clauses (i.e. inverse of two parallel steps of resolutions). It identifies a common set of literals in rules and replace them with a new concept.

The latter two operations do not generalise the current knowledge but restructure it.

## Learning as a search

### Given

- a set of observations in  $L_e$ 
  - positive examples  $E^+$
  - negative examples  $E^-$
- background knowledge  $B$
- hypothesis language  $L_H$
- **covers**  $c(B, L_H, e)$

### Find

- A theory  $H$  in  $L_H$  that covers
  - $\forall e \in E^+ : B \cup H \models e$   
( $H$  is complete)
  - $\forall e \in E^- : B \cup H \not\models e$   
( $H$  is consistent)

Two basic types of search operators

- $\theta$ -subsumption-based
  - specialisation operator ( $\rho$  refinement)
  - generalisation operator (lgg of 2 clauses)
- inverse resolution

FOIL  
MIS  
DUNE  
CIGOL

Going back to the problem of how to compute logic-based learning as a search over a version space of theories, we have seen that different operations/approaches can be used when the generality relation between clauses in a version space is expressed as entailment and the hypothesis space is that of definite clauses.

In particular, from the point of view of how to search over the version space, we have identified two main search operators.

**The first one is based on  $\theta$ -subsumption.** The traversal can be done either top-down by specialising the clauses that cover negative examples (using Shapiro  $\rho$  refinement operator in his FOIL system), or bottom-up by computing more general clauses from given specific clauses (using the lgg operation proposed by Plotkin in his Model Inference System learner). **These types of approaches essentially have the characteristic of adding logic-based notion (such as  $\theta$ -subsumption) on top of an existing inductive learning approach (that is version space often used in data mining).**

The second operator is based on inverse entailment. This can essentially be seen as defining first a logic-based process (inverse resolution) as underlying mechanism and define induction on top of a richer logic-based inference mechanism.

On the right hand-side I have listed key learning systems that have been developed in the '80 using these different operations. The inverse resolution was proposed by Muggleton as part of a system called DUNE. Its subsequent extension to first-order logic gave rise to a system called CIGOL (LOGIC read in inverse order!).

Inverse resolution is also a bottom-up approach as it computes more general clauses from more specific one.

## Learning Settings

Different formalisation of concept learning in logic, depending of different representation of examples:

- Learning from interpretations
- Learning from entailment

Various forms of concept learning in logic have been developed. These include learning from interpretation, learning from entailment and very recently learning from answer sets. What differentiate these paradigms is the representation of the examples and the corresponding notion of covers relation. We will briefly describe what they are, as we will consider each one of them with related algorithms throughout the course.

## Learning from interpretation

Different formalisation of concept learning in logic, depending of different representation of examples:

- **Learning from interpretations**
  - **Example** = interpretation (set of facts)  $e$ 
    - contains a full description of the example
    - all information that intuitively belongs to the example, is represented in the example, not in background knowledge
  - **Background** = domain knowledge
    - general information concerning the domain, not concerning specific examples

A definite clause theory  $H$  is a solution if and only if  $e$  is a model of  $H$

In the paradigm of learning from interpretations, examples are assumed to be full interpretations. All the information relevant to an example is included in the example itself. Remember that any ground atom not included in an interpretation is assumed to be false. The background knowledge includes just domain knowledge not concerning with the example. A hypothesis (or solution) is therefore a clause theory that covers all the given positive examples. Coverage in this case mean that each positive example is a model of the hypothesis.

## Learning from interpretation

Examples

```
pos: {contains(1, o1), contains(1, o2), triangle(o1),
      points(o1, down), circle(o2), pos(1)}
pos: {contains(2, o3), triangle(o3), points(o3, down), pos(2)}
```

Background knowledge

```
polygon(X) ← triangle(X).
polygon(X) ← square(X).
```



Hypothesis

```
pos(X) ← contains(X,Y),
           triangle(Y),
           points(Y, down).
```

Consider a snapshot of the example given earlier that has to recognise specific patterns of geometrical shapes. In this case the first example contains an object that is a triangle pointing down and a circle. The second example contains just an object that is a triangle also pointed down. The background knowledge is independent from the examples in the sense that it does not define concepts that are used in the example. (this is because otherwise we would not be able to assume that the example is a full interpretation).

So the learned hypothesis in this case is a clause that accepts both the two positive examples as models.

## Learning from entailment

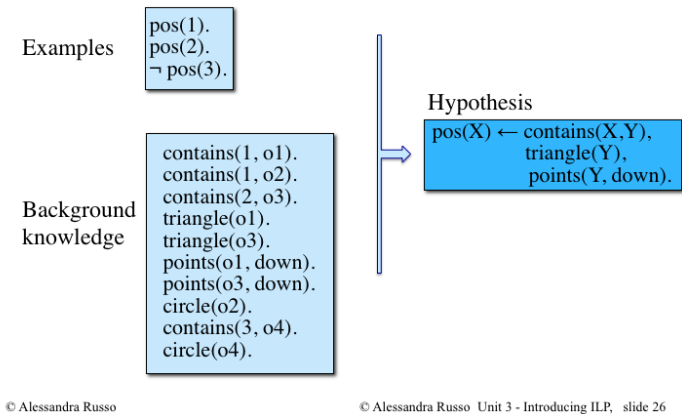
Different formalisation of concept learning in logic, depending of different representation of examples:

- **Learning from entailment**
  - Examples  $E = \langle E^+, E^- \rangle$  (*single example  $e$  is a ground fact*)
  - Background  $B$  = definite clause theory
  - FIND hypothesis  $H$  (definite clause theory) such that
    - $B \cup H \models e^+$  for all  $e^+ \in E^+$
    - $B \cup H \not\models e^-$  for all  $e^- \in E^-$

The learning from entailment is a completely different paradigm. Most of the results in Inductive Logic Programming have been based on the notion of learning from entailment. In the last few years new paradigms have also been introduced which we are going to see later in the course.

The learning from entailment is similar to what we have already informally introduced earlier. Examples in this case are single ground facts and background knowledge can be related to the given examples. The task is to find an hypothesis that, together with the background knowledge entails each positive example and does not entail any of the negative examples.

## Learning from entailment



The same example considered earlier can be formulated in terms of learning from entailment in the following way. We could include the information related to each pattern as part of the background knowledge. The hypothesis will then find the rule that correctly links the facts in the background knowledge with the examples.

Learning from entailment are in general more complex than the examples given here. Background knowledge is in general a set of (definite) clauses not just a set of facts but examples are assumed to be always a set of facts. The learning task may refer to learn just the concept that the examples expresses, or sometime concepts that do not appear in the example but appear in the background knowledge and that together with the background knowledge entail the examples.

## Summary

- Introduced ILP as **concept learning**
- Defined a notion of **generality relation**
- Generality in terms of **entailment** and then  **$\theta$ -subsumption**
- Search operators
  - **Specialisation** operators for **top-down** search
    - Shapiro's  $\rho$  refinement operator
  - **Generalisation** operators for **bottom-up** search
    - Plotkin's least general generalisation (lgg)
    - Muggleton's inverse resolution
- Two learning paradigms
  - **Learning from interpretation**
  - **Learning from entailment**